

Sunti Similarity Indexing and Searching

Lo scopo principale è quello di ottenere da un grande database immagini simili alla query, necessitiamo quindi di una **funzione di similarità** che misura la similarità tra oggetti e di **algoritmi di ricerca** su grandi database. Ovviamente per descrivere le immagini non prendiamo direttamente i pixels ma usiamo le feature che abbiamo visto nel modulo precedente.

Distance searching problem

In generale, quando vogliamo cercare qualcosa dobbiamo necessariamente definire il *tipo di dati*, *il metodo di comparazione* e *la formulazione della query*. Inoltre, è auspicabile avere *tecniche di indexing generali* che possono essere applicate a più problemi di ricerca specifici (*extensibility*).

Possiamo dividere i problemi di ricerca in:

- **Ricerca tradizionale:** che si basa sul trovare un risultato esatto, parziale o totale che sia. Inoltre, il dominio dei dati è sortable per numeri o stringhe.
- Ricerca prospettiva (**Perspective search**): andiamo a cercare risultati basandosi su metriche non esatte come ad esempio similarità, dissimilarità, prossimità e distanza. Il dominio in questo caso non è sortable, infatti, abbiamo dati rappresentati per esempio tramite la Hamming distance o color histograms.

Metric Space

Dato un dominio D e una misura della distanza d , prendiamo $X \subseteq D$ contenente n elementi (il nostro db) e vogliamo strutturare i dati in modo che le query di prossimità siano eseguite in modo efficientemente. Per fare questo possiamo utilizzare il **metric space** come astrazione della ricerca, in esso si può utilizzare la distanza per effettuare la ricerca ma non abbiamo coordinate per identificare gli oggetti nello spazio, non possiamo quindi partizionare i dati nello spazio. La differenza con il vector space è che in esso conosciamo le posizioni dei dati. Utilizziamo il metric space perché non conosciamo le coordinate quindi non possiamo utilizzare il vector space e non abbiamo altre possibilità, in più le performance per casi speciali sono buone ed infine abbiamo un'alta extensibility.

Il metric space $M=(D,d)$ è composto dal dominio dei dati D e dalla funzione della distanza d : $D \times D \rightarrow \mathbb{R}$ (chiamata anche metric function o metric), prende due oggetti e mi restituisce un numero reale.

Il metric space ha le seguenti **proprietà**:

Non negativity $\forall x, y \in D, d(x, y) \geq 0$

Symmetry $\forall x, y \in D, d(x, y) = d(y, x)$

Identity $\forall x, y \in D, x = y \Leftrightarrow d(x, y) = 0$

Triangle inequality $\forall x, y, z \in D, d(x, z) \leq d(x, y) + d(y, z)$

Posso sostituire la proprietà di identità con le seguenti due:

(p3) reflexivity $\forall x \in D, d(x, x) = 0$

(p4) positiveness $\forall x, y \in D, x \neq y \Rightarrow d(x, y) > 0$

Otteniamo quindi

(p1) non negativity $\forall x, y \in D, d(x, y) \geq 0$

(p2) symmetry $\forall x, y \in D, d(x, y) = d(y, x)$

(p3) reflexivity $\forall x \in D, d(x, x) = 0$

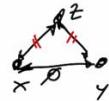
(p4) positiveness $\forall x, y \in D, x \neq y \Rightarrow d(x, y) > 0$

(p5) triangle inequality $\forall x, y, z \in D, d(x, z) \leq d(x, y) + d(y, z)$

Pseudo Metric

Se la proprietà *positiveness* p4 non è verificata otteniamo uno **pseudo metric** space, cioè ho oggetti differenti che sono a distanza zero, se consideriamo questi due oggetti come lo stesso allora torniamo nel metric space.

- | To be proved $d(x, y) = 0 \Rightarrow \forall z \in \mathcal{D}, d(x, z) = d(y, z)$
- | Since $d(x, z) \leq d(x, y) + d(y, z)$
 $d(y, z) \leq d(x, y) + d(x, z)$
- | We get $d(x, y) = 0 \Rightarrow \begin{cases} d(x, z) \leq d(y, z) \\ d(y, z) \leq d(x, z) \end{cases} \Rightarrow d(x, z) = d(y, z)$



Quasi metric

Se la proprietà *symmetry* p2 non è verificata abbiamo un **quasi metric** space, un esempio pratico sono strade a senso unico che collegano un punto a ad un punto b, quindi la distanza tra questi due punti in un senso e nell'altro sarà differente.

La trasformazione nel metric space avviene al verificarsi di

$$d_{\text{sym}}(x, y) = d_{\text{asym}}(x, y) + d_{\text{asym}}(y, x)$$

Super metric

Chiamato anche ultra metric, prevede un vincolo più stretto sulla *disuguaglianza triangolare*

$$\forall x, y, z \in \mathcal{D}: d(x, z) \leq \max \{d(x, y), d(y, z)\}$$

Le tre distanze sopra sono i lati di un triangolo, il vincolo mi dice che non possiamo avere un lato più grande di tutti gli altri quindi almeno due lati devo essere lunghi uguali (triangolo isoscele).

Metric distance measures

Distinguiamo funzioni:

- Discrete: funzioni che ritornano solo un piccolo e predefinito set di valori
- Continue: funzioni nelle quali la cardinalità del set di valori ritornato è molto grande o infinito.

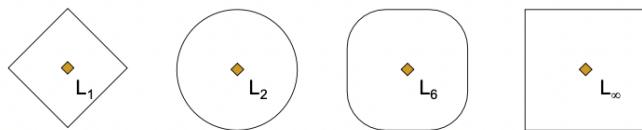
Minkowsky distance

Chiamata anche L_p metrics, viene definita su vettori di dimensione n:

$$L_p[(x_1, \dots, x_n), (y_1, \dots, y_n)] = \sqrt[p]{\sum_{i=1}^n |x_i - y_i|^p}$$

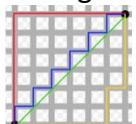
- L_1 – Manhattan (City-Block) distance
- L_2 – Euclidean distance
- L_∞ – maximum (infinity) distance

$$L_\infty = \max_{i=1}^n |x_i - y_i|$$



A sx casi speciali di Minkowsky

Manhattan distance: la distanza tra i due punti è la stessa indipendentemente dai percorsi rosso blu o giallo



Quadratic form distance

Nella distanza euclidea andiamo a comparare celle di vettori nella stessa dimensione (prima cella con prima, ecc), può essere però necessario comparare anche celle in posizioni differenti (prima cella con terza), questo metodo si chiama **cross talk** (es color histograms) e viene applicato usando una matrice M nxn semidefinita positiva

$$d_M(\vec{x}, \vec{y}) = \sqrt{(\vec{x} - \vec{y})^T \cdot M \cdot (\vec{x} - \vec{y})}$$

Nel caso in cui la matrice M = diag(w₁, ..., w_n) abbiamo la distanza euclidea pesata:

$$d_M(\vec{x}, \vec{y}) = \sqrt{\sum_{i=1}^n w_i (x_i - y_i)^2}$$

Vedi slide 21/22 per esempio.

Edit distance

Chiamata anche *Levenshtein distance*, compara due stringhe x e y misurando il numero minimo di operazioni atomiche per trasformare la stringa x in y.

- **insert** character c into string x at position i

$$\text{ins}(x, i, c) = x_1 x_2 \cdots x_{i-1} c x_i \cdots x_n$$

- **delete** character at position i in string x

$$\text{del}(x, i) = x_1 x_2 \cdots x_{i-1} x_{i+1} \cdots x_n$$

- **replace** character at position i in x with c

$$\text{replace}(x, i, c) = x_1 x_2 \cdots x_{i-1} c x_{i+1} \cdots x_n$$

Se i pesi (costi) di inserimento e rimozione sono differenti, l'edit distance non è simmetrica, in questo caso non siamo nel metric space.

- Example: $w_{\text{insert}} = 2$, $w_{\text{delete}} = 1$, $w_{\text{replace}} = 1$

$$d_{\text{edit}}(\text{"combine"}, \text{"combination"}) = 9$$

replacement e → a, insertion t,i,o,n

$$d_{\text{edit}}(\text{"combination"}, \text{"combine"}) = 5$$

replacement a → e, deletion t,i,o,n

Non so dove si trova la stringa nello spazio e non posso deciderlo, però posso calcolarne la distanza.

Jaccard's coefficient

Viene utilizzato per misurare la distanza tra set, ad esempio misurare la differenza in persone tra due città.

$$d(A, B) = 1 - \frac{|A \cap B|}{|A \cup B|} \quad \begin{matrix} \text{Cardinality of intersection /} \\ \text{Cardinality of union} \end{matrix}$$

Se i due set non hanno oggetti in comune allora la distanza sarà 1.

La **Tanimoto distance** viene per vettori e corrisponde alla Jaccard's in caso di vettori binari

$$d_{TS}(\vec{x}, \vec{y}) = 1 - \frac{\vec{x} \cdot \vec{y}}{\|\vec{x}\|^2 + \|\vec{y}\|^2 - \vec{x} \cdot \vec{y}}$$

$\vec{x} \cdot \vec{y}$ is the scalar product

$\|\vec{x}\|$ is the Euclidean norm

Hausdorff distance

Anch'essa misura la distanza tra set, compara gli elementi tramite la distanza d_e .

Measures the extent to which each point of the "model" set A lies near some point of the "image" set B and vice versa.

Two sets are within Hausdorff distance r from each other if and only if any point of one set is within the distance r from some point of the other set.

$$d_p(x, B) = \inf_{y \in B} d_e(x, y), \quad \text{Distance between object and set and viceversa}$$

$$d_p(A, y) = \inf_{x \in A} d_e(x, y), \quad \text{viceversa}$$

$$d_s(A, B) = \sup_{x \in A} d_p(x, B), \quad \text{Between sets}$$

$$d_s(B, A) = \sup_{y \in B} d_p(A, y).$$

$$d(A, B) = \max \{d_s(A, B), d_s(B, A)\}.$$

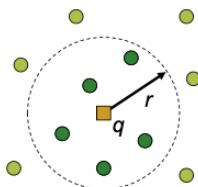
Similarity Queries

Similarity Range Query

Cerco i punti che sono distanti dalla query al più r .

$$R(q, r) = \{x \in X \mid d(q, x) \leq r\}$$

Es. Tutti i musei distanti al max 2km dal mio hotel



Nearest Neighbor Query

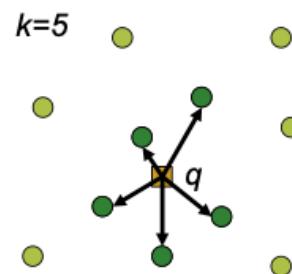
the nearest neighbor query

- ◻ $NN(q) = x$
- ◻ $x \in X, \forall y \in X, d(q, x) \leq d(q, y)$

$k=5$

k -nearest neighbor query

- ◻ $k-NN(q, k) = A$
- ◻ $A \subseteq X, |A| = k$
- ◻ $\forall x \in A, y \in X - A, d(q, x) \leq d(q, y)$



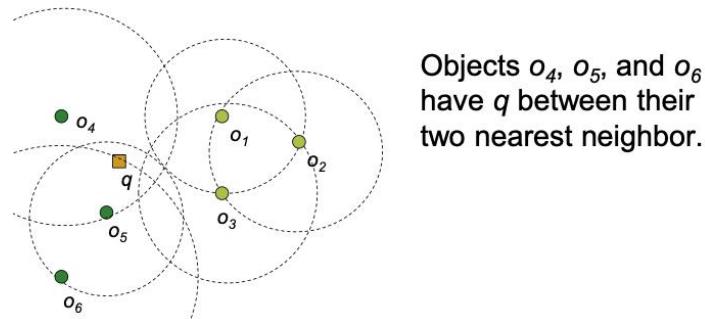
Reverse Nearest Neighbor

Cerco quegli oggetti che hanno la query tra i loro kNN.

$$kRNN(q) = \{R \subseteq X, \forall x \in R : q \in kNN(x) \wedge$$

$$\forall x \in X - R : q \notin kNN(x)\}$$

Es. Tutti gli hotel con uno specifico museo tra i loro kNN.



Necessitiamo quindi di tutti i kNN di tutti gli oggetti per vedere chi ha la query tra i propri.

Similarity Queries

Misuriamo la similarità tra due dataset, utilizzando μ come soglia di distanza.

Per ogni punto dei due dataset viene calcolata la distanza, se questa è inferiore a μ allora la coppia viene aggiunta al risultato della query.

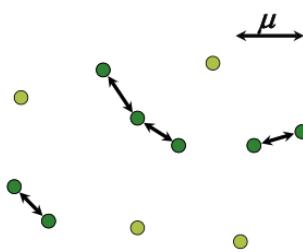
$$X \subseteq \mathcal{D}, Y \subseteq \mathcal{D}, \mu \geq 0$$

$$J(X, Y, \mu) = \{(x, y) \in X \times Y : d(x, y) \leq \mu\}$$

Es. le coppie di hotel e musei che distano al massimo 5 minuti.

Similarity self join

Uguale a prima solo che prendiamo un solo dataset $\Leftrightarrow X=Y$



Combined Query

Possiamo combinare diversi tipi di query:

- **Range + NN:** i k elementi più simili e dentro un certo range, quindi il risultato può avere meno di k elementi.

$$kNN(q, r) = \{R \subseteq X, |R| \leq k \wedge \forall x \in R, y \in X - R : d(q, x) \leq d(q, y) \wedge d(q, x) \leq r\}$$

- **NN + Similarity Join:** cerchiamo i k più vicini che hanno una distanza minima tra di loro

Complex Query

Andiamo a cercare più features contemporaneamente, ad es trovare i migliori match di oggetti di **forma tonda** e di **colore rosso**. Non è garantito che il miglior match con feature prese singolarmente sia anche il miglior match delle features combinate.

The A0 Algorithm:

Per ogni predicato si prendono gli oggetti in ordine discendente di similarità uno alla volta fino a che non abbiamo un resultset temporaneo che contiene **k** elementi, in pratica prendiamo i migliori di ogni predicato poi i secondi migliori e così via finchè non si arriva a k. Una volta ottenuto il resultset temporaneo si considerano tutti i prediciati e si stabilisce il rank finale dei risultati, ad esempio dando dei pesi maggiori se dei prediciati hanno più importanza.

Basic Partitioning Principles

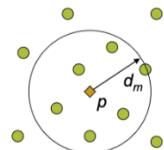
Dato un set di oggetti contenuti in un metric space definiamo tre tecniche di partizione:

- Ball partitioning
- Generalized hyperplane
- Excluded middle partitioning

Ball Partitioning

Inner set: $\{x \in X \mid d(p,x) \leq d_m\}$

Outer set: $\{x \in X \mid d(p,x) > d_m\}$

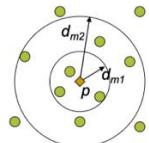


Multi-way ball partitioning

Inner set: $\{x \in X \mid d(p,x) \leq d_{m1}\}$

Middle set: $\{x \in X \mid d(p,x) > d_{m1} \wedge d(p,x) \leq d_{m2}\}$

Outer set: $\{x \in X \mid d(p,x) > d_{m2}\}$

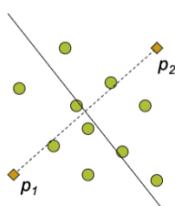


Generalized Hyperplane

La linea continua è una linea di equidistanza, cioè la distanza tra i due pivot è la medesima.

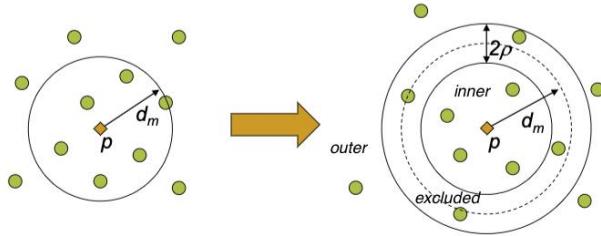
$\{x \in X \mid d(p_1, x) \leq d(p_2, x)\}$

$\{x \in X \mid d(p_1, x) > d(p_2, x)\}$



Excluded middle partitioning

Inner set: $\{ x \in X \mid d(p,x) \leq d_m - \rho \}$
 Outer set: $\{ x \in X \mid d(p,x) > d_m + \rho \}$



Excluded set: otherwise

A cosa serve questo metodo? Dato che non si conosce le posizioni perché siamo nel metric space, usiamo la distance distribution per vedere se ci sono zone più dense, esse dovranno essere messe all'interno dell'excluded set per applicare altre partizioni più specifiche.

Principles of Similarity Query Execution

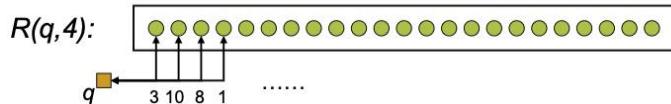
Il **costo** per rispondere ad una query è definito da:

- Partitioning principle
- Query execution algorithm

Basic Strategies

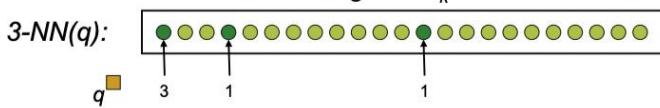
■ Sequential organization & range query $R(q,r)$

- All database objects are consecutively scanned and $d(q,o)$ are evaluated.
- Whenever $d(q,o) \leq r$, o is reported on result



■ Sequential organization & k -NN query $3\text{-NN}(q)$

- Initially: take the first k objects and order them with respect to the distance from q .
- All other objects are consecutively scanned and $d(q,o)$ are evaluated.
- If $d(q,o_i) \leq d(q,o_k)$, o_i is inserted to a correct position in answer and the last neighbor o_k is eliminated.

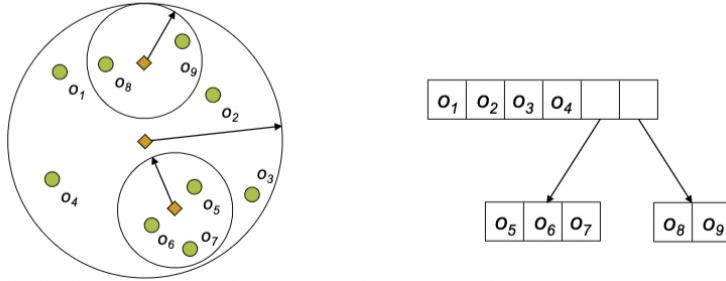


Ball Region Organization

Invece di usare il sequential scanning andiamo ora ad usare un approccio **gerarchico** tramite l'uso delle ball region, definiamo un insieme di oggetti G e una bounding region $R(G)$ che contiene tutti gli elementi di G . Ogni elemento appartiene ad un solo insieme G e per ogni G abbiamo un solo root node.

Using ball regions

- Root node organizes four objects and two ball regions.
- Child ball regions has two and three objects respectively.



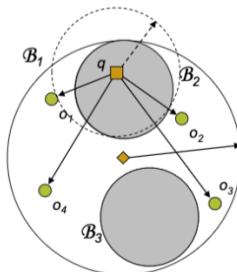
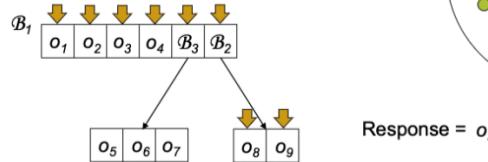
Se applichiamo una **range query** al solito si procede come segue

Given $Q=R(q,r)$:

- Start at the root.
- In the current node $N=(G, R(G))$, process all elements:
 - object element $o_j \in G$:
 - if $d(q, o_j) \leq r$, report o_j on output.
 - non-object element $N'=(G', R(G')) \in G$
 - if $R(G')$ and $R(Q)$ intersect, recursively search in N' .

$R(q,r)$:

- Start inspecting elements in \mathcal{B}_1 .
- \mathcal{B}_3 is not intersected.
- Inspect elements in \mathcal{B}_2 .
- Search is complete.



Response = o_8, o_9

Nearest Neighbor Search Algorithm

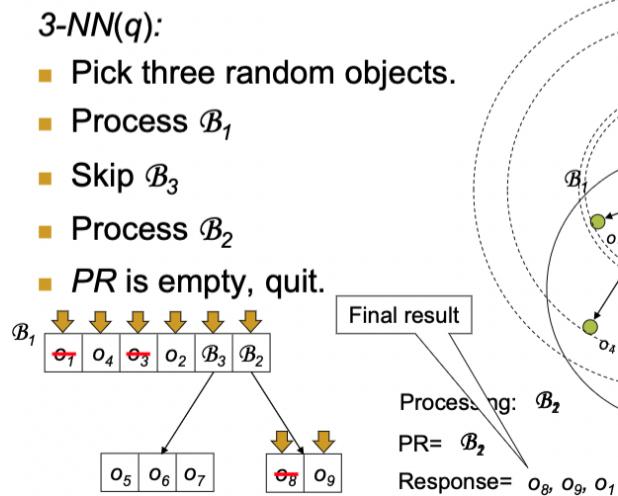
In questo caso il range della query non è definito perché cerchiamo "i più vicini", si parte con un raggio molto grande che comprende tutto il data set che viene poi ridotto di volta in volta successivamente all'analisi degli oggetti e alla valutazione dei NN.

Manteniamo anche una coda di priorità PR in cui vengono inserite le regioni che devono essere analizzate sortate in ordine decrescente di importanza (ipotizzo che una buona politica potrebbe essere prima le regioni con il pivot più vicino alla query).

Inizialmente partiamo prendendo tre oggetti random da tutto il data set, mettiamo il root node nella PR e scorriamo i suoi oggetti per vedere se ne troviamo qualcuno più vicino rispetto ai tre random, in tal caso andremo a diminuire il raggio della query.

Se il raggio attuale diventa piccolo a tal punto da non intersecare una ball, come B3 nell'esempio, allora non controllerò gli elementi al suo interno.

Poi analizzo le ball nella PR, cioè B2, e vedo che o9 è più vicino e lo prenderò nel final result.

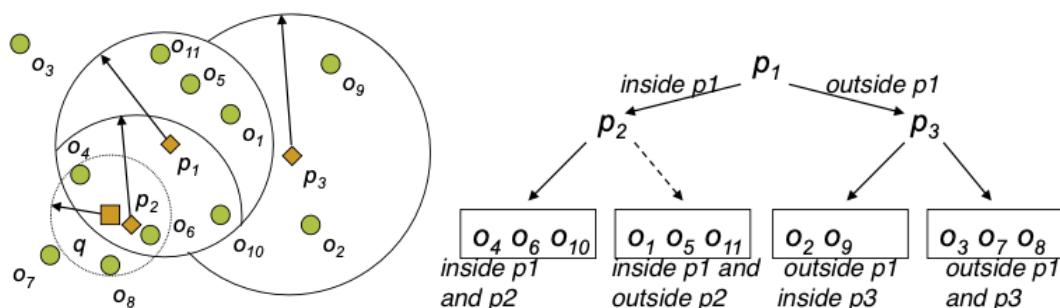


Policies to Avoid Distance Computations

Nel metric space misurare la distanza è dispendioso per questo motivo limitare il numero delle distanze da valutare velocizzare l'esecuzione di una query di similarità. Possiamo adottare delle strategie di pruning come:

- object-pivot
- range-pivot
- pivot-pivot
- double-pivot
- pivot-filtering

Introduciamo un index structure costruito sui 11 oggetti in figura, applicando il ball-partitioning: per costruire l'indice parto da p1 e mi chiedo chi sta dentro e chi fuori da p1, poi prendo p2 che sta dentro p1 e mi chiedo chi sta dentro sia p1 che p2 e chi sta fuori. Poi lo stesso anche per p3 che è fuori da p1.

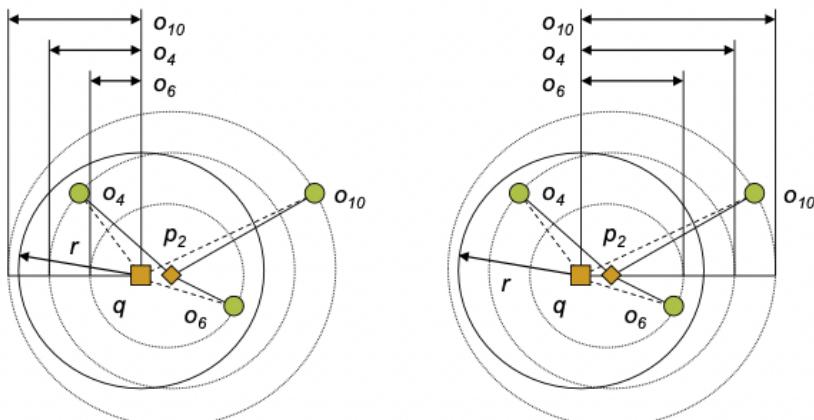


Quando applichiamo una range query $R(q,r)$ dobbiamo per prima cosa vedere con quale ball si interseca la query, per farlo dobbiamo calcolare 3 distanze $d(q,p1)$, $d(q,p2)$, $d(q,p3)$, vengono intersecati p3 (un pochino), p2 e p1, non interseco inside p1 e outside p2 quindi non considero la seconda foglia. Le altre distanze da calcolare sono 8 e sono le distanze tra gli oggetti delle altre foglie e la query.

Object-Pivot constraint

Viene solitamente applicato ai nodi foglia, spighiamo questo concetto tramite un esempio. Prendiamo l'index structure dell'immagine precedente e assumiamo che la foglia più a sinistra venga visitata. Per capire se gli oggetti presenti nella foglia appartengono o meno al resultset andrebbero calcolate le distanze fra la query q e ogni oggetto presente nella foglia. Le distanze fra p_2 e ogni oggetto presente nella foglia sono già state calcolate durante la fase di creazione della struttura, in più conosciamo anche la distanza fra la query e p_2 $d(p_2, q)$. Conoscendo queste distanze possiamo **stimare** la distanza fra la query e ad esempio o_{10} definita come $d(q, o_{10})$. Essendo in un metric space non possiamo conoscere la posizione esatta di un oggetto, ma solo la distanza fra due oggetti. Con questo concetto non possiamo dire dove si trova o_{10} con esattezza, sappiamo però che si può trovare in ogni punto della circonferenza tratteggiata. Detto questo possiamo trovare i due casi limite mostrati nella figura sotto.

Quando o_{10} si trova all'estrema destra di p_2 si otterrà la distanza massima dalla query, viceversa, quando o_{10} si trova a sinistra di p_2 otterremo la distanza minima dalla query.



Lower bound is $|d(q, p_2) - d(p_2, o_{10})|$

□ If greater than query radius, o_{10} cannot qualify

Upper bound is $d(q, p_2) + d(p_2, o_{10})$

□ If less than query radius, o_{10} directly qualifies!

I due limiti sono calcolati come mostrato nella figura precedente, ci possono aiutare a includere o a escludere automaticamente un oggetto dal resulset senza dover calcolare la distanza fra la query e l'oggetto in questione. In particolare, se il limite superiore è più piccolo del raggio della query allora l'oggetto viene preso sicuramente, viceversa, se il limite inferiore è maggiore del raggio della query, allora l'oggetto sarà sicuramente escluso. Se non possiamo verificare queste due condizioni allora sarà necessario calcolare la distanza fra l'oggetto e la query.

In maniera più generale la distanza fra la query e un oggetto è necessariamente compresa fra i due limiti come mostrato nella figura successiva.

Given a metric space $\mathcal{M}=(\mathcal{D}, d)$ and three objects

$q, p, o \in \mathcal{D}$, the distance $d(q, o)$ can be constrained:

$$LB \leq |d(q, p) - d(p, o)| \leq UB$$

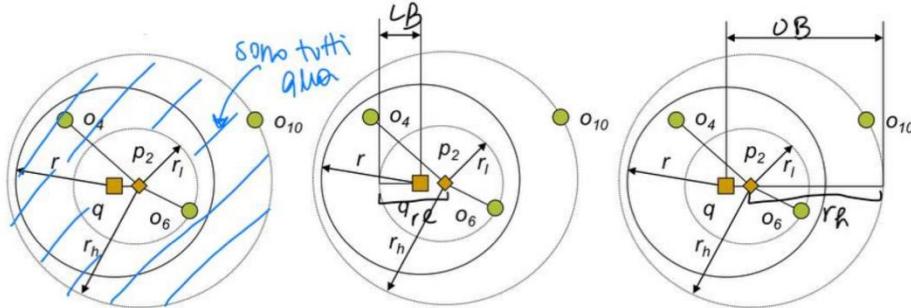
$$d(q, p) - d(p, o) \leq d(q, o) \leq d(q, p) + d(p, o)$$

Range-Pivot constraint

Le strutture spesso non salvano le distanze fra un pivot e tutti gli elementi di una foglia, per ogni foglia vengono salvati due valori corrispondenti alla distanza massima e a quella minima fra il pivot e gli oggetti della foglia. Questo range viene descritto come: $[r_l, r_h]$.

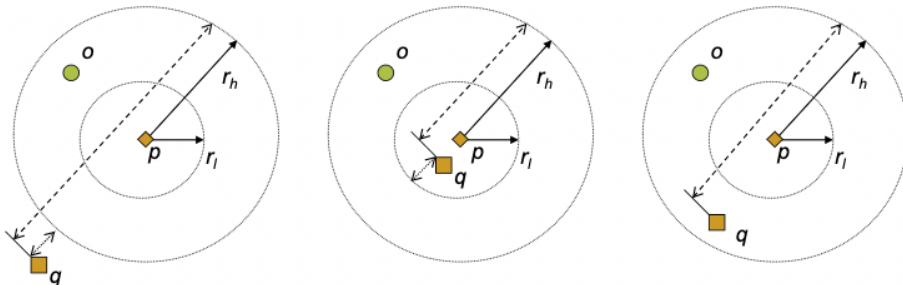
Tramite questo range possiamo decidere se accedere o meno alla foglia.

Dato che $d(q, p_2)$ è nota possiamo definire i limiti per questo caso specifico. In particolare, se il limite inferiore risulta più grande del raggio della query allora **nessun** oggetto della foglia può far parte del result set, se invece il limite superiore risulta essere più piccolo del raggio della query allora **tutti** gli oggetti della foglia vanno inseriti nel resultset.



- Lower bound is $r_l - d(q, p_2)$
 - if greater than the query radius r , no object can qualify
- Upper bound is $r_h + d(q, p_2)$
 - if less than the query radius r , all objects qualify!

Il caso che abbiamo analizzato è solo uno dei tre possibili. Può capitare che la query sia anche nelle posizioni 1 e 3, in queste posizioni cambiano i limiti che dobbiamo considerare. Nel terzo caso il limite inferiore è 0 perché un oggetto può essere coincidente con la query.



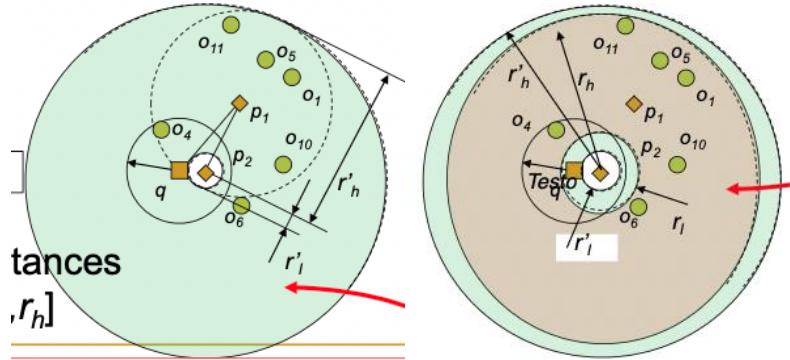
- Given a metric space $\mathcal{M}=(\mathcal{D},d)$ and objects $p, o \in \mathcal{D}$ such that $r_l \leq d(o,p) \leq r_h$. Given $q \in \mathcal{D}$ with known $d(q,p)$. The distance $d(q,o)$ is restricted by:

$$\max \{d(q,p) - r_h, r_l - d(q,p), 0\} \leq d(q,o) \leq d(q,p) + r_h$$

Pivot-pivot Constraint

Quello che vorremmo fare è capire a priori se entrare o meno nel sottoalbero p_2 (cioè la query non contiene nessun oggetto nelle foglie di p_2), evitando di calcolare la distanza tra la query e p_2 e tra la query e tutti gli oggetti.

1. Dato che sappiamo le distanze $d(p_1, p_2)$ e $d(q, p_1)$ possiamo applicare l'**object-pivot** constraint per stimare la distanza $d(q, p_2) \in [r'_l, r'_h]$ (diciamo in questo modo che la query può stare in qualsiasi punto dell'area verde).



2. Possiamo quindi applicare il **range-pivot** constraint per ottenere r_l e r_h che corrispondono alla distanza minima e massima nel quale possono stare gli oggetti di p_2 : $d(\mathbf{0}, \mathbf{p}_2) \in [r_l, r_h]$ (ogni oggetto può stare nell'area marrone).

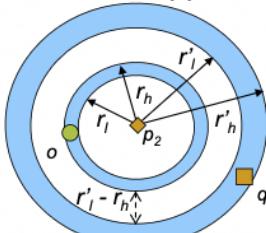
Con questi due vincoli possiamo stabilire LB e UB tra query e oggetti:

- LB = 0 perché la query può stare nel verde e l'oggetto sul marrone quindi possono stare nella stessa posizione
- • UB = $r_h + r'_l$ che è il caso in cui q sia sul bordo verde e o sul bordo marrone dal lato opposto

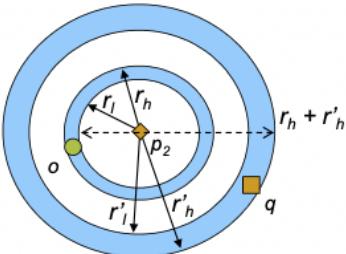
Se i due dischi non si intersecano possiamo avere due casi differenti:

1. $[r_l, r_h]$ è più piccolo di $[r'_l, r'_h]$ (disco che contiene gli oggetti è interno al disco che contiene la query). In tal caso:

- a. LB = $r'_l - r_h$ come si nota facilmente da sotto



- b. UB = $r_h + r'_l$



2. Il secondo caso è l'inverso $[r'_l, r'_h]$ è più piccolo di $[r_l, r_h]$ e quindi:

- a. LB = $r_l - r'_h$
- b. UB = $r_h + r_h'$

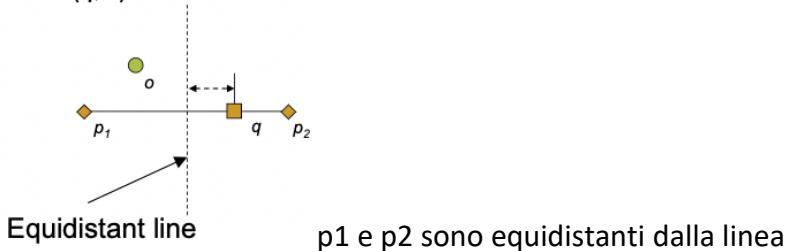
Given a metric space $\mathcal{M}=(\mathcal{D},d)$ and objects $q,p,o \in \mathcal{D}$
such that $r_l \leq d(o,p) \leq r_h$ and $r'_l \leq d(q,p) \leq r'_h$. The
distance $d(q,o)$ can be restricted by:

$$\max\{r'_l - r_h, r_l - r'_h, 0\} \leq d(q,o) \leq r'_h + r_h$$

Double Pivot Distance Constraint

Anche qua l'idea è di non calcolare la distanza tra q e o , ma di trovare delle stime.

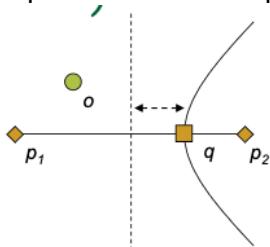
Precedentemente i constraints usavano un solo pivot nel ball partitioning, adesso applichiamo l'hyper-plane generalizzato usando due pivots. Non si può definire un UB su $d(q,o)$ perché avendo un piano si può andare all'infinito.



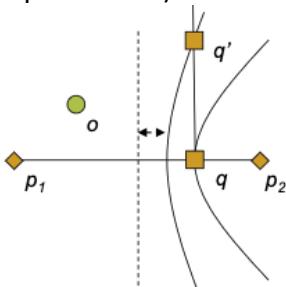
Per quanto riguarda il LB, se abbiamo q e o in **sottospazi differenti** allora il LB è quando o è sulla linea verticale e q sulla linea che collega p_1 e p_2

- $\text{LB} = (d(q,p_1) - d(q,p_2))/2$ (qualsiasi sia la posizione di q tra p_2 e la linea di equidistanza)

L'iperbole mostra la posizione di q tenendo costante il LB.



Se invece vogliamo spostare q verso l'alto (manteniamo quindi la distanza tra q e la linea di equidistanza) il LB sarà più piccolo.



Se invece abbiamo che q e o sono nello **stesso sottospazio** il LB è 0 perché possono coincidere. Come si può vedere dalla figura successiva, la distanza tra un oggetto è la query sta fra i limiti.

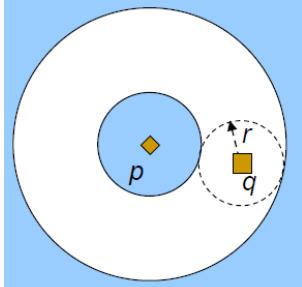
Given a metric space $\mathcal{M}=(\mathcal{D},d)$ and objects

$o, p_1, p_2 \in \mathcal{D}$ such that $d(o, p_1) \leq d(o, p_2)$. Given a query object $q \in \mathcal{D}$ with $d(q, p_1)$ and $d(q, p_2)$. The distance $d(q, o)$ can be lower-bounded by:

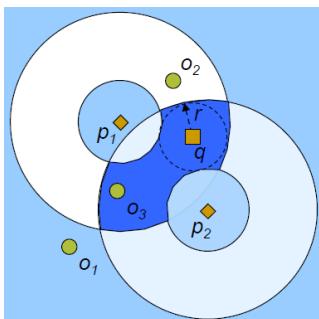
$$\max\left\{\frac{d(q, p_1) - d(q, p_2)}{2}, 0\right\} \leq d(q, o)$$

Pivot Filtering

Si basa sul concetto di utilizzare più di una volta la tecnica dell'**object pivot**. Usando solo un pivot e $|d(p,o) - d(p,q)| > r$ posso eliminare tutti gli oggetti che hanno il LB maggiore del raggio della query.

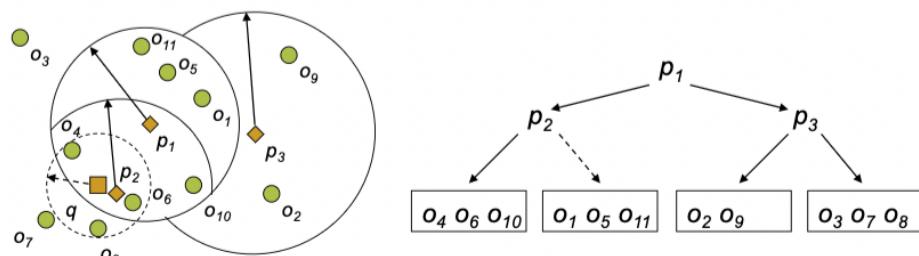


Aumentando il numero di pivot ad esempio a due, ottengo che solo gli oggetti nella zona blu scura devono essere analizzati, mentre tutti gli altri possono essere scartati.



Constraint and Explanatory

- **Range query $R(q,r) = \{o_4, o_6, o_8\}$**
 - Sequential scan: 11 distance computations
 - No constraint: 3+8 distance computations



Calcolo le 3 distanze tra query e pivot, scarto la seconda foglia e poi calcolo le distanze tra q e tutti gli 8 oggetti rimasti.

Object Pivot

Range query $R(q,r) = \{o_4, o_6, o_8\}$

- Only object-pivot in leaves: 3+2 distance computations
 - o_6 is included without computing $d(q,o_6)$
 - $o_{10}, o_2, o_9, o_3, o_7$ are eliminated without computing.

Ricordiamo che sappiamo le distanze tra i pivot e tutti gli oggetti.

Per prima cosa calcolo la distanza tra i pivot e la query e scarto la seconda foglia perché q non interseca inside p1 e outside p2.

Poi applico l'**object pivot** calcolando i LB e UB per ogni oggetto e così facendo scelgo sicuramente o_6 perché il suo UB è minore del raggio della query e scarto a priori $o_{10}, o_2, o_9, o_3, o_7$. Non mi resta che calcolare $d(q, o_4)$ e $d(q, o_8)$.

Range Pivot

Range query $R(q, r) = \{o_4, o_6, o_8\}$

- Only range-pivot: 3+6 distance computations
 - o_2, o_9 are pruned.
- Only range-pivot +pivot-pivot: 3+6 distance computations

Scartiamo al solito la seconda foglia, tramite il **range-pivot** andiamo a scartare o_2 e o_9 perché il raggio della query non interseca il Lb.

Range query $R(q, r) = \{o_4, o_6, o_8\}$

- Assume: objects know distances to pivots along paths to the root.
- Only pivot filtering: 3+3 distance computations (to o_4, o_6, o_8)
- All constraints together: 3+2 distance computations (to o_4, o_8)

Principles of approximate similarity search

L'idea è quella di non restituire il risultato calcolato in maniera matematicamente perfetta come avviene con gli **exact similarity search**. Questo perché la similarità è un concetto soggettivo in quanto ogni individuo può dare maggiore importanza ad un dettaglio piuttosto che ad un altro. Dato che non può esistere un risultato che accontenti tutti, si può pensare di approssimare il risultato. Questa approssimazione potrebbe consistere, ad esempio con una range query, nel non restituire tutti i risultati con distanza minore del raggio, ma solo alcuni. Questo ci porta chiaramente ad un aumento notevole delle performance al costo di perdere accuratezza nel risultato. Questo metodo viene usato ad esempio in sistemi interattivi, l'utente preferisce query che vengono eseguite velocemente piuttosto che query lente con il risultato matematicamente esatto. Si ottiene un valore anche migliore di **due ordini di grandezza**.

Una strategia può essere quella di trasformare lo spazio di lavoro; tuttavia, non andremo ad analizzare questo tipo di strategie.

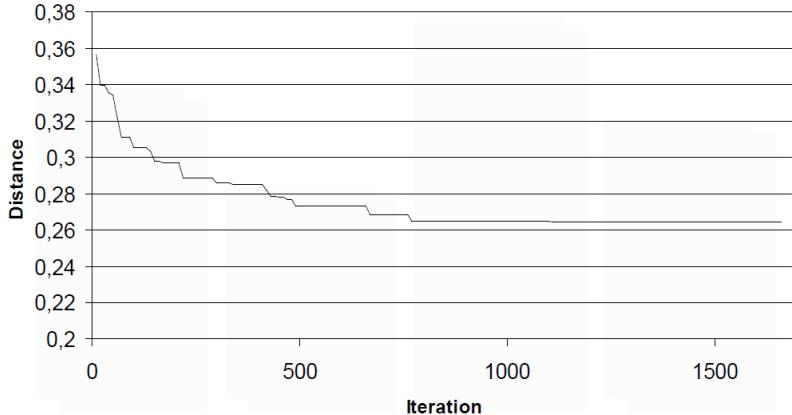
Andremo invece ad analizzare le tecniche che permettono di ridurre i subset di dati che dovranno essere analizzati. Esistono due strategie possibili:

- **Early termination strategies:** l'algoritmo si ferma dopo che tutti i dati necessari sono stati recuperati.
- **Relaxed branching strategies:** le regioni che si intersecano con la regione della query possono essere scartate secondo delle specifiche regole. Ad esempio, se la regione di intersezione è molto piccola allora è altamente probabile che non contenga nessun risultato utile.

Early termination strategies

Gli algoritmi di approximate similarity search usano una particolare regola di stop, l'algoritmo di ferma quando ci si accorge che la possibilità di migliorare il risultato è molto bassa. L'ipotesi che sta alla base della scelta di questa regola è che si ottiene un buon risultato **dopo poche approssimazioni**, gli step successivi servono solamente a migliorare di poco il risultato portando però un elevato costo.

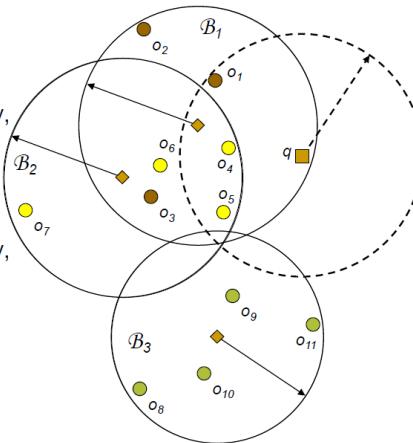
L'immagine successiva rappresenta il grafico di un KNN, sull'asse delle y è riportata la distanza mentre sull'asse delle x rappresenta le iterazioni. Si tiene traccia della distanza minima degli oggetti nel risultato, si può notare che dopo 500 iterazioni il miglioramento è quasi nullo, si potrebbe interrompere l'algoritmo a 500 iterazioni senza perdere troppo.



Relaxed Branching strategies

Abbiamo già parlato del ragionamento alla base di questo approccio, vediamo con un esempio pratico per capire meglio.

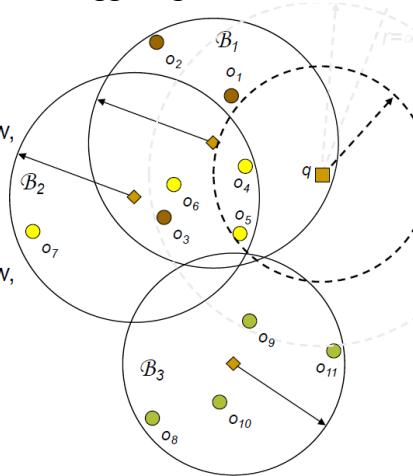
- Given a range query:
- Access \mathcal{B}_1
 - Report o_1
 - If early termination stopped now, we would loose objects.
- Access \mathcal{B}_2
 - Report o_4, o_5
 - If early termination stopped now, we would not loose anything.
- Access \mathcal{B}_3
 - Nothing to report
 - A relaxed branching strategy may discard this region – we don't loose anything.



La regione \mathcal{B}_3 sarà scartata in quanto la porzione che interseca la query è molto piccola, infatti non contiene nessun risultato utile.

Abbiamo visto un esempio con una Range query, vediamo cosa succede con una query di tipo KNN. Supponiamo di voler trovare i 2 oggetti più vicini alla query. Ricordiamo che questo tipo di query non ha il raggio, si parte con il raggio uguale a infinito.

- Given a 2-NN query:
- Access \mathcal{B}_1
 - Neighbors: o_1, o_3
 - If early termination stopped now, we would loose objects.
- Access \mathcal{B}_2
 - Neighbors: o_4, o_5
 - If early termination stopped now, we would not loose anything.
- Access \mathcal{B}_3
 - Neighbors: o_4, o_5 – no change
 - A relaxed branching strategy may discard this region – we don't loose anything.



Analizzando la prima zona otteniamo o1 e o3 come valori più vicini, il raggio è settato a o3. Accedendo alla seconda zona si trova che o4 e o5 sono i più vicini, la terza zona non cambia il risultato e quindi sarà probabilmente scartata.

Measures of performance

Dopo aver preso in considerazione queste due strategie è necessario capire come si possano misurare le performance. Si deve tenere di conto di due fattori: **il miglioramento in efficienza e l'accuratezza del risultato**. Di solito questi due fattori formano un trade-off, per ottenere un buon algoritmo di approximate search si vuole un buon incremento in efficienza mantenendo alta l'accuratezza del risultato.

Una prima misura di performance è **Improvement in Efficiency(IE)** calcolata come

$$IE = \frac{Cost(Q)}{Cost^A(Q)}$$

Dove la funzione di costo indica il numero di accessi al disco oppure il numero di distanze che vengono calcolate. un valore di IE=10 indica che l'esecuzione con approssimazione è circa 10 volte più veloce della stessa esecuzione senza approssimazione.

Un altro metodo per misurare le performance è quello di usare Precision e Recall

- Accuracy can be quantified with *Precision (P)* and *Recall (R)*:

$$P = \frac{|S \cap S^A|}{|S^A|}, \quad R = \frac{|S \cap S^A|}{|S|}$$

- S – qualifying objects, i.e., objects retrieved by the precise algorithm
- S^A – actually retrieved objects, i.e., objects retrieved by the approximate algorithm
- They are very intuitive but in our context
 - Their interpretation is not obvious & misleading!!!
- For approximate range search we typically have $S^A \subseteq S$
 - Therefore, precision is always 1 in this case
- Results of $k\text{-NN}(q)$ have always size k
 - Therefore, precision is always equal to recall in this case.
- Every element has the same importance
 - Loosing the first object rather than the 1000th one is the same.

Di seguito ci sono due esempi di precision e recall calcolati su query di tipo KNN

- Suppose a $10\text{-NN}(q)$:
 - $S=\{1,2,3,4,5,6,7,8,9,10\}$
 - $S^{A1}=\{2,3,4,5,6,7,8,9,10,11\}$ the object 1 is missing
 - $S^{A2}=\{1,2,3,4,5,6,7,8,9,11\}$ the object 10 is missing
- In both cases: $P = R = 0.9$
 - However S^{A2} can be considered better than S^{A1} .
- Suppose $1\text{-NN}(q)$:
 - $S=\{1\}$
 - $S^{A1}=\{2\}$ just one object was skipped
 - $S^{A2}=\{10000\}$ the first 9,999 objects were skipped
- In both cases: $P = R = 0$
 - However S^{A1} can be considered much better than S^{A2} .

Si può notare che cambiamenti importanti non fanno variare precisione e recall, questo tipo di misure non ci è molto utile.

Un altro metodo che possiamo usare è quello di utilizzare la **relative error on distances(ED)**. Questa misura può essere definita come nella seguente immagine.

It compares the distances from a query object to objects in the approximate and exact results

$$ED = \frac{d(o^A, q) - d(o^N, q)}{d(o^N, q)} = \frac{d(o^A, q)}{d(o^N, q)} - 1$$

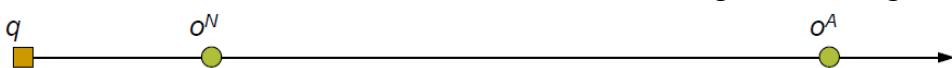
where o^A and o^N are the approximate and the actual nearest neighbors, respectively.

Si compara la distanza fra la query e l'oggetto approssimato con la distanza fra la query e l'oggetto esatto. Questa formula può anche essere generalizzata per ogni oggetto di una query KNN nel seguente modo.

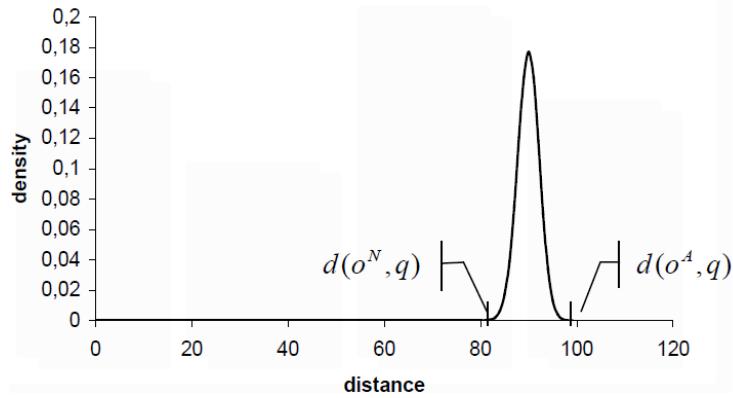
$$ED_j = \frac{d(o_j^A, q)}{d(o_j^N, q)} - 1$$

Questo approccio ha uno svantaggio notevole, non prende in considerazione la distribuzione delle distanze. Per capire meglio quanto questo svantaggio possa influire facciamo due esempi.

Nel primo esempio abbiamo che la differenza fra la query/oggetto esatto e query/oggetto approssimato è molto grande, questo produrrà un valore di ED molto elevato anche se si è saltato solo un elemento del database come mostrato nella seguente immagine.



Nel secondo esempio consideriamo che tutti gli oggetti hanno la stessa distanza dalla query, prendere l'oggetto più lontano rispetto a quello più vicino produrrà un piccolo valore di ED anche se abbiamo effettivamente saltato la maggior parte del dataset.



L'ultima misura che prendiamo in considerazione è **Error on Position (EP)**. In altre parole, la discrepanza in rank fra il risultato esatto e quello approssimato. Questo tipo di misura è definita come segue.

Obtained using the *Sperman Footrule Distance*

(SFD):

$$SFD = \sum_{i=1}^{|X|} |S_1(o_i) - S_2(o_i)|$$

$|X|$ – the dataset's cardinality

$S_i(o)$ – the position of object o in the ordered list S_i

La funzione S ritorna la posizione dell'oggetto o .

SFD viene usato per calcolare la correlazione fra due liste che hanno la stessa cardinalità, per le liste che non soddisfano questa condizione utilizziamo **Induced Footrule Distance(IFD)** definita come segue.

$$IFD = \sum_{i=1}^{|S^A|} |OX(o_i) - S^A(o_i)|$$

OX – the list containing the entire dataset ordered with respect to q .

S^A – the approximate result ordered with respect to q .

Si può aggiungere un fattore di normalizzazione ottenuto moltiplicando le cardinalità delle due liste per far ricadere IFD in $[0,1]$ dove 1 indica l'errore maggiore e 0 indica l'assenza di errori.

- Position in the approximate result is always smaller than or equal to the one in the exact result.
 - S^A is a sub-lists of OX
 - $S^A(o) \leq OX(o)$
- A normalisation factor $|S^A| \cdot |X|$ can also be used
- The *error on position (EP)* is defined as

$$EP = \frac{\sum_{i=1}^{|S^A|} (OX(o_i) - S^A(o_i))}{|S^A| \cdot |X|}$$

Advance Issues

Statistics on Metric Dataset

Ci sono delle statistiche caratteristiche dei dataset che formano la base per l'ottimizzazione delle performance dei database. Queste informazioni statistiche sono utilizzate per definire il costo del modello e per accedere a strutture di tuning (utili per ottimizzare i parametri), degli esempi di informazioni statistiche sono:

- Istogrammi di frequenze
- Distribuzione dei dati, in caso di dati rappresentati nel vector space.

Dato che queste statistiche richiedono di sapere le coordinate degli oggetti non possiamo usarli nel metric space dove possiamo affidarci solo alle distanze non conoscendo la posizione degli oggetti.

Per questo motivo andiamo ad utilizzare statistiche utili per applicare le tecniche di similarity searching nel metric space:

- Distance density e distance distribution
- Homogeneity of viewpoints
- Proximity of ball regions

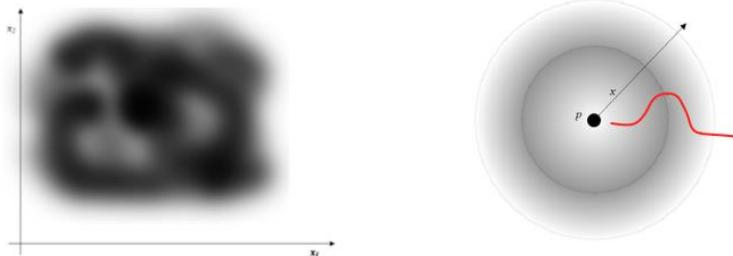
Data Density vs Distance Density

Data density: può essere applicato solo nei vector space dal momento che questa caratteristica indica come sono disposti gli oggetti nello spazio, di conseguenza dobbiamo conoscere le coordinate degli oggetti.

Distance density: può essere applicato nei metric space perché questo indice si basa sulla distanza fra gli oggetti, le coordinate infatti non sono necessarie. Serve solo una funzione utile a calcolare la distanza.

Nell'immagine seguente viene mostrata la differenza tra questi due approcci, nel primo si possono notare le regioni più dense. Utilizzando invece la distance density possiamo solo conoscere le frequenze delle distanze, per questo motivo si formano delle regioni circolari, non conoscendo la posizione esatta dell'oggetto dobbiamo cercare su tutta la circonferenza.

- Data density
- Distance density from the object p



Possiamo rappresentare la distance density dell'esempio tramite una gaussiana nel grafico distanza/densità cioè rappresenta la **distance distribution**.

Possiamo definirla rispetto ad un oggetto p (Questa distribuzione viene chiamata **viewpoint**):

$$F_{D_p}(x) = \Pr\{D_p \leq x\} = \Pr\{d(p, o) \leq x\}$$

Dove D_p è una variabile random che corrisponde alla distanza $d(p, o)$ dove o è un punto casuale del metric space.

La distance density di un oggetto p è ottenibile derivando la distribuzione di distanza.

La distance distribution valuta la probabilità tra un punto p fissato e un oggetto o randomico, possiamo generalizzare questa formula ottenendo l'**overall distance distribution che** è invece la probabilità di distanza tra oggetti

$$F(x) = \Pr\{d(o_1, o_2) \leq x\}$$

Dove o_1 e o_2 sono oggetti random nel metric space e x è la soglia scelta.

Homogeneity of Viewpoints

Per **viewpoint** si intende la distance distribution da un punto p, ovviamente al cambiare di p cambierà anche il viewpoint perché le distanze tra gli oggetti saranno distribuite in modo differente.

Il viewpoint sarà differente anche dalla overall distance distribution dato che in questa non andiamo a fissare un punto ma sono entrambi casuali ed è caratterizzata da tutte le distanze possibili nel dataset.

Tuttavia se il dataset è omogeneo rispetto alla probabilità di distanza allora possiamo usare la overall dd al posto di usare i view point perché le distanze sono omogenee. Possiamo dire che un dataset è omogeneo se la discrepanza tra i viewpoints (tutti) è piccola.

Definiamo quindi un indice che ci permette di stabilire se un dataset è omogeneo o meno e di conseguenza poter usare l'overall dd:

Homogeneity of viewpoints (HV) per un metric space $M=(D,d)$

$$HV(M) = 1 - \operatorname{avg}_{p_1, p_2 \in D} \delta(F_{p_1}, F_{p_2})$$

Dove p_1 e p_2 sono oggetti randomici e la discrepanza tra due viewpoints viene così definita

$$\delta(F_{p_i}, F_{p_j}) = \operatorname{avg}_{x \in [0, d^+]} |F_{p_i}(x) - F_{p_j}(x)|$$

Dove F_{p_i} è il viewpoint di p_i

Dati due punti casuali p_i e p_j , quello che facciamo è per ogni distanza possibile x (da 0 a d^+ che è la distanza massima) andiamo a calcolare la differenza tra le distance distribution e ne facciamo la media (seconda formula). Ripetiamo questo procedimento per ogni possibile coppia di punti nel dataset e ne facciamo la media totale (prima formula), che poi andremo a sottrarre a 1. Più questa media sarà piccola più la discrepanza generale media fra i punti sarà piccola, in tal caso possiamo definire questo dataset come omogeneo.

Di conseguenza più $HV \approx 1$ più il dataset è omogeneo, possiamo quindi utilizzare l'overall distance distribution.

Proximity of ball region

La prossimità tra due regioni è la misura che viene usata per stimare il numero di oggetti presenti nell'intersezione, questa misura viene usata in:

- Region splitting for partitioning: dopo aver diviso una regione le nuove devono avere meno oggetti in comune possibili
- Disk allocation: aumento di performance distribuendo dati su più dischi
- Approximate search: aiuta a decidere se si deve accedere ad una regione in base alla probabilità che l'intersezione con la query contenga oggetti o meno.

Nello spazio euclideo è facile da ottenere, si deve calcolare la data distribution e farne l'integrale sull'intersezione per trovare la data density.

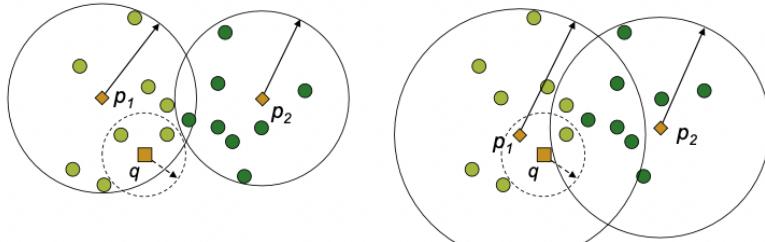
Nel metric space dato che non si possono utilizzare le coordinate non si può sfruttare la data distribution, infatti, le uniche statistiche che abbiamo sono la distance density/distribution.

Vediamo come ci può essere utile nei casi elencati prima:

Partitioning

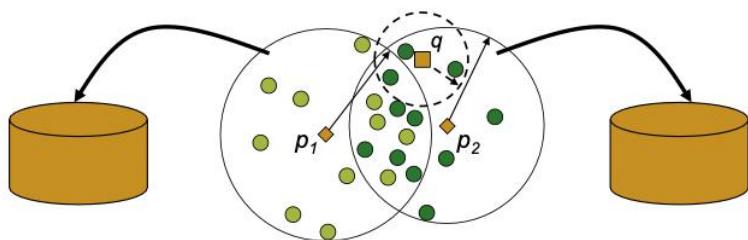
Solitamente la query segue la data distribution, cioè si troverà in zone dense; il nostro obiettivo è partizionare i dati in modo da evitare o limitare il più possibile l'intersezione tra le regioni, così facendo avremo meno probabilità che la query comprenda anche parte dell'intersezione che corrisponderebbe a controllare entrambe le regioni.

- Low overlap (left) vs. high overlap (right)



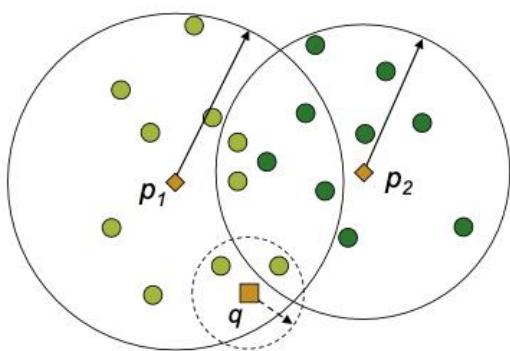
Data allocation

Se nell'intersezione abbiamo molti oggetti come in figura la query è molto probabile che cada nell'intersezione perché segue la distribuzione dei dati, allora conviene separare le due regioni su due dischi diversi in modo da poter eseguire la query in parallelo sui due dischi ed ottenere il risultato più velocemente.



Approximate Search

Se riduciamo l'intersezione abbiamo meno probabilità che la query trovi oggetti rilevanti nell'intersezione, di conseguenza, non dobbiamo accedere ad entrambe le regioni ma solo ad una.



Riprendiamo ora il concetto di prossimità, date due regioni R_1 e R_2 la misura di **proximity** è definita nel seguente modo.

$$prox(\mathcal{R}_1, \mathcal{R}_2) = \Pr\{d(p_1, o) \leq r_1 \wedge d(p_2, o) \leq r_2\}$$

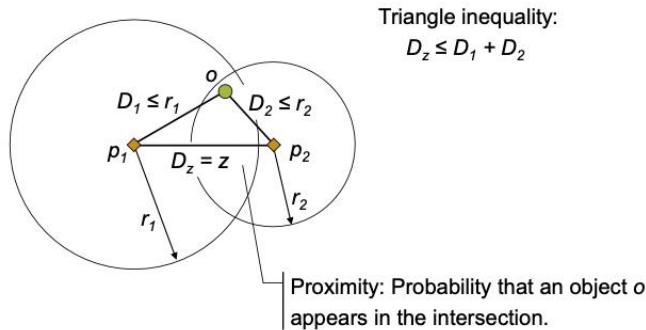
che equivale a calcolare la probabilità che un oggetto qualunque si trovi dentro entrambe le regioni.

Nei dataset reali la distribuzione di distanza non dipende da oggetti specifici ma dalla visione intera del ds, inoltre, dato che in situazioni reali abbiamo anche un alto indice di omogeneità possiamo definire la **overall proximity**:

$$prox_z(r_1, r_2) = \Pr\{d(p_1, o) \leq r_1 \wedge d(p_2, o) \leq r_2 | d(p_1, p_2) = z\}$$

cioè per ogni coppia di regioni possibili, considero quelle i cui centri distano z , e restituisco la probabilità che un oggetto si trovi all'interno dell'intersezione tra le due regioni.

Nell'overall proximity si deve soddisfare la diseguaglianza triangolare tra i due punti p_1 e p_2 e l'oggetto, vedremo successivamente il motivo ed a che cosa ci servirà.



Vediamo come calcolare effettivamente questa prossimità. Definiamo $D_1 = d(p_1, o)$, $D_2 = d(p_2, o)$ e $D_z = d(p_1, p_2)$, la **overall proximity** può essere calcolata come segue

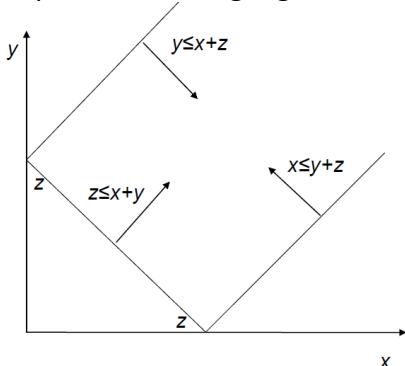
$$prox_z(r_1, r_2) = \int_0^{r_1} \int_0^{r_2} f_{D_1, D_2 | D_z}(x, y | d_z) dy dx$$

La funzione per la probabilità condizionata congiunta non è conosciuta per i metric space generici, per risolvere questo problema possiamo utilizzare la probabilità congiunta $f_{D_1, D_2}(x, y)$. Questa probabilità è più semplice da calcolare perché se le due variabili sono indipendenti può essere calcolata come il prodotto delle due densità. Se poi il dataset è omogeneo possiamo utilizzare le **overall densities**.

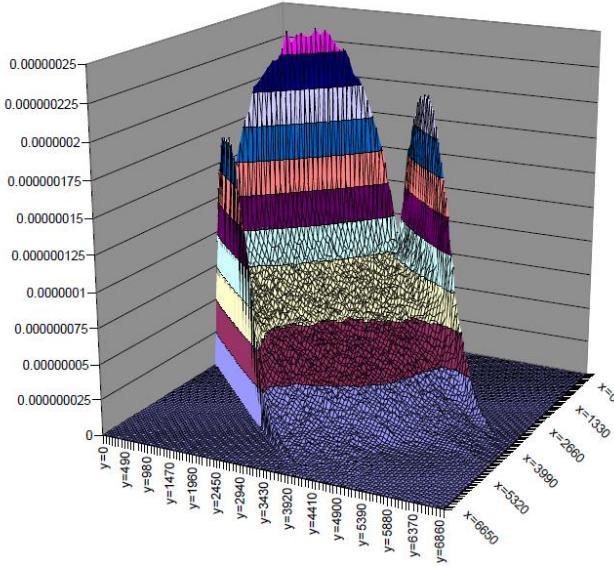
$$f_{D_1 D_2}(x, y) = f_{D_1}(x) \cdot f_{D_2}(y) \quad f_{D_1 D_2}(x, y) = f(x) \cdot f(y)$$

In altre parole eliminiamo il vincolo della distanza fra i due centri uguale a z .

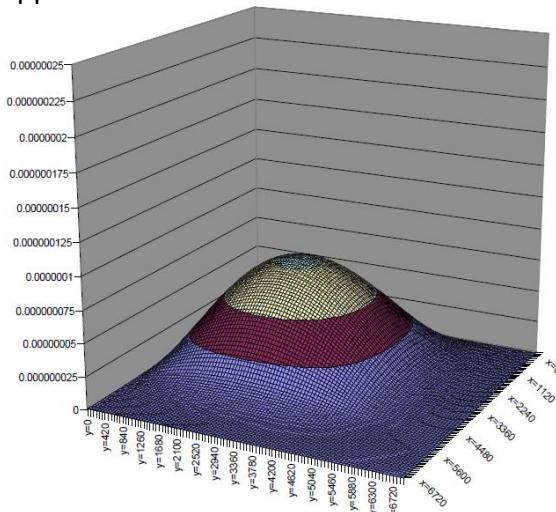
La probabilità congiunta condizionata implica che non tutte le triplette di x , y e z siano accettabili, esse devono soddisfare la diseguaglianza triangolare $z \leq x + y$. Fissando quindi un certo z (la distanza fra i due centri) tutti i punti fuori dalla regione avranno probabilità 0 perché non rispettano la diseguaglianza triangolare.



In maniera grafica possiamo costruire la probabilità di ogni coppia di punti x e y fissato z , questo è mostrato nella seguente figura.



Per ottenere questo risultato l'idea che abbiamo avuto è stata quella di usare solo la probabilità congiunta senza quindi considerare il vincolo di distanza z fra i due centri. Sappiamo calcolare la densità congiunta essendo una semplice moltiplicazione fra densità per ottenere la seguente rappresentazione



Per ottenere la probabilità congiunta condizionata è necessario fissare un valore di z e forzare tutti i valori al di fuori della regione accettabile verso il bordo, per questo motivo si creano i picchi del grafico mostrato sopra.

Chiaramente stiamo facendo un'approssimazione, ma grazie a questo metodo siamo in grado di calcolare la probabilità che due regioni contengano un elemento.

Performance prediction

La distance distribution può anche essere usata per fare delle stime sulle performance dei metodi si similarity search. In particolare, possiamo:

- Stimare il numero di subset ai quali accediamo
- Stimare il numero di distanze che devono essere calcolate
- Stimare il numero di oggetti che devono essere recuperati, ad esempio da una range query

Supponiamo che il dataset generale sia partizionato in m subset e ogni subset è racchiuso da una ball region con un centro e un raggio.

Data quindi una Range query $R(q, r_q)$ sappiamo che dovremo accedere ad una regione i se la regione si interseca con la query, quindi se la distanza fra i due centri è minore o uguale alla somma dei due raggi. Grazie a questa definizione possiamo definire anche la probabilità che si debba accedere ad una qualsiasi regione nel seguente modo:

$$\Pr\{d(q, p) \leq r + r_q\} = F_q(r + r_q) \approx F(r + r_q)$$

La probabilità può anche essere vista anche come la distribuzione delle distanze considerando la query come viewpoint e la somma dei raggi come distanza, chiaramente se il dataset è omogeneo si può usare la **overall distance distribution**. Quindi in questo modo otteniamo la probabilità che si acceda ad una regione.

Date queste definizioni possiamo calcolare il **numero di regioni** alle quali accediamo semplicemente sommando la probabilità di accedere ad ogni regione

$$subsets(R(q, r_q)) \approx \sum_{i=1}^m F(r_i + r_q)$$

l'unico vincolo è quello di essere in grado di mantenere in memoria i raggi di **tutte** le regioni.

In maniera simile si può stimare il **numero di distanze che devono essere calcolate**, possiamo infatti moltiplicare la probabilità di accedere ad una determinata regione per il numero di oggetti che quella regione contiene. In pratica, se accediamo ad una regione si devono calcolare le distanze per tutti gli oggetti appartenenti a quella determinata regione. Si può usare la seguente formula, ricordandosi però che è necessario salvarsi anche tutte le cardinalità.

$$distances(R(q, r_q)) \approx \sum_{i=1}^m |\mathcal{R}_i| F(r_i + r_q)$$

Se invece volessimo stimare il **numero di oggetti** presenti nel risultato di una range query dovremmo semplicemente usare la distribuzione delle distanze usando il raggio della query, questo mi indica la probabilità che un oggetto randomico del database abbia distanza dal centro della query minore del raggio.

$$objects(R(q, r_q)) \approx n \cdot F(r_q)$$

Come detto prima abbiamo bisogno di una struttura per mantenere tutti i raggi e le cardinalità, questo potrebbe diventare insostenibile con l'aumentare del dataset. Per risolvere questo problema si possono usare delle approssimazioni, si dividono le regioni in una struttura ad albero staticamente e poi dobbiamo solo salvare **due indici per livello**, il numero di regioni per ogni livello e il raggio medio per ogni livello.

Grazie questa approssimazione le formule viste sopra diventano le seguenti:

$$subsets(R(q, r_q)) \approx \sum_{l=1}^L M_l F(ar_l + r_q)$$

rapp: > medio per livello
al max accedo a M_l regioni per livello

$$distances(R(q, r_q)) \approx \sum_{l=1}^L M_{l+1} F(ar_l + r_q)$$

Per quanto riguarda il numero di regioni alle quali accediamo moltiplichiamo la probabilità si trovi a distanza inferiore della somma tra il raggio della query e il raggio medio del livello per il numero di regioni presenti in quel livello, dobbiamo applicare questo ragionamento per ogni livello e fare la somma.

Per quanto riguarda il numero delle distanze da calcolare dobbiamo fare attenzione, **il numero di oggetti presenti in una regione equivale al numero di figli**; quindi, per capire quante distanze dobbiamo calcolare basta moltiplicare la probabilità di accedere ad una regione per il numero di nodi presenti al livello successivo.

Fino ad ora abbiamo analizzato solo le **range query**, per quanto riguarda le query di tipo K-NN possiamo dire che l'algoritmo ottimo accede alle regioni alle quali accederebbe una range query che ha come raggio la distanza del k-esimo vicino di q (viene indicato con o_k ed è il vicino più lontano). Tuttavia, la distanza fra il centro della query e il k-esimo vicino non è nota a priori.

Prima di tutto dobbiamo definire la probabilità che il k-esimo vicino di q abbia una certa distanza x , per fare questo usiamo la seguente formula che sfrutta la distribuzione della distanza

$$F_{o_k}(x) = \Pr\{d(q, o_k) \leq x\} = 1 - \sum_{i=0}^{k-1} \binom{n}{i} F(x)^i (1 - F(x))^{n-i}$$

Di conseguenza la densità f_{o_k} è la derivata della distribuzione F_{o_k} .

Come conseguenza di quello che abbiamo appena detto, il numero di regioni alle quali dobbiamo accedere è pari all'integrale da 0 alla distanza massima delle regioni per la range query moltiplicate la probabilità che il k-esimo vicino si trovi ad una certa distanza, chiaramente l'integrale è svolto lungo r.

$$subsets(kNN(q)) \approx \int_0^{d^+} subsets(R(q, r)) f_{o_k}(r) dr$$

distanza max
prob che il k-esimo abbia r come distanza

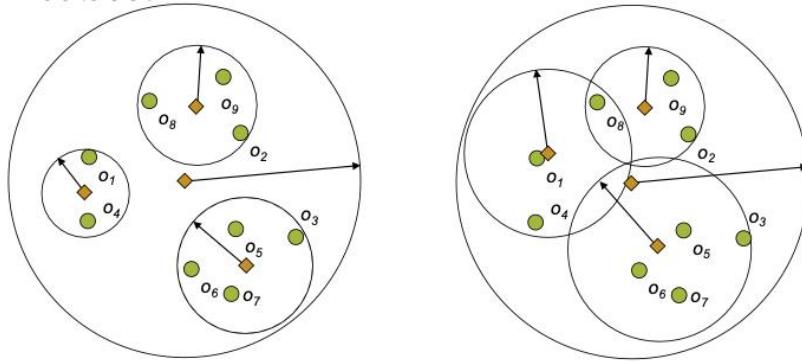
In maniera simile, il numero di distanze da calcolare seguirà la seguente formula

$$distances(kNN(q)) \approx \int_0^{d^+} distances(R(q, r)) f_{o_k}(r) dr$$

Tree Quality Measure

Vogliamo poter dire se abbiamo una buona data structures o no, le strutture (index) con cui spesso andiamo a definire un database sono organizzate ad albero. Possiamo costruire differenti strutture ad albero a partire dallo stesso data set, nell'esempio della figura sottostante vediamo due possibilità: la prima con le regioni ben distinte mentre la seconda le regioni si sovrappongono. Inoltre, il primo albero è più compatto e l'occupazione dei nodi figli è più alta mentre nel secondo è bassa e l'albero è meno compatto.

Ovviamente è meglio il primo ma il nostro obiettivo è definire un modo per valutare la qualità dell'albero.



Fat Factor

La misura si basa sull'intersezione delle regioni

$$overlap = \frac{|R_1 \cap R_2|}{|R_1 \cup R_2|}$$

È un ratio tra l'intersezione tra gli oggetti delle regioni diviso l'unione, se l'intersezione contiene tutti gli oggetti overlap=1. Un buon albero ha un basso overlap.

La **misura** va a contare il numero totale di accessi richiesti per rispondere alla query su tutti gli oggetti del database. Se l'intersezione di due regioni R_1 e R_2 contengono un oggetto o , allora, entrambi i nodi corrispondenti devono essere acceduti per rispondere alla query.

$$fat(T) = \frac{I_c - nh}{n} \cdot \frac{1}{m-h}$$

I_c sono il numero totale di nodi acceduti quando eseguiamo n query con range 0 (exact match query), cioè la query è un punto. In altre parole prendo n query diverse con raggio 0 che indicano n punti nello spazio, I_c è il numero totale di nodi acceduti per rispondere a tutte le query.

Nell'albero migliore non abbiamo intersezione e il numero di accessi per una query è pari al numero di livelli (h) che compongono l'albero (un nodo acceduto per ogni livello), quindi in totale abbiamo nh accessi.

Nel peggior caso tutte le regioni si sovrappongono quindi per ogni query si devono accedere tutte le m regioni (nodi), quindi un totale di nm accessi.

Riassumendo

An ideal tree needs to access just one node per level.

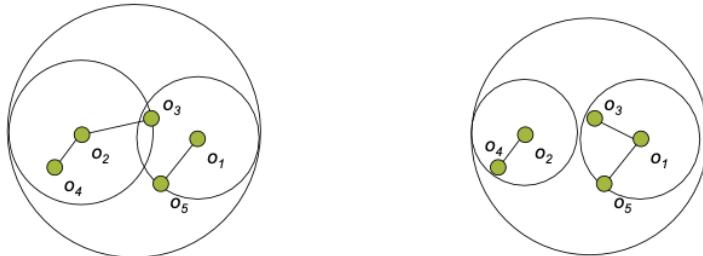
◻ $fat(T_{ideal}) = 0 \rightarrow I_c = nh$

The worst tree always access all nodes.

◻ $fat(T_{worst}) = 1 \rightarrow I_c = nm$

Esempio:

- Two trees organizing 5 objects:
 - $n=5 \quad m=3 \quad h=2$
 - $I_C=11$
 - $\text{fat}(T)=0.2$
- $I_C=10$
- $\text{fat}(T)=0$



Per il calcolo del primo I_C andiamo su ogni punto e contiamo il numero di nodi che si deve accedere, per $o_4 \ o_2 \ o_5 \ o_1$ è 2 ognuno mentre per o_3 è 3 perché si deve accedere alle due regioni più piccole e a quella grande che comprende tutto. Quando calcoliamo il fat factor vediamo che non è esattamente 0 quindi si può migliorare ancora.

Appunto nel secondo esempio con le regioni che non si intersecano il fat factor è 0, cioè il caso migliore.

Some consideration about Fat Factor

- Solo le range queries sono prese in considerazione, le kNN queries sono un caso speciale di range quindi possiamo utilizzarle.
- La distribuzione delle exact match query segue la distribuzione dei data object, in generale ci si aspetta che le query saranno più probabilmente in zone dense.
- Weak point: non viene considerato il numero di nodi presenti nell'albero, quindi un albero grande con un basso fat-factor è migliore rispetto ad uno piccolo con un fat-factor un po' più grande (se si prende nell'immagine sopra a dx un albero dove ad ogni regione è assegnato un oggetto soltanto il fat-factor sarà 0 ma non ho un buon albero perché non partiziono nulla).

Per ovviare a quest'ultimo problema usiamo il Relative Fat Factor.

Relative Fat Factor

Non usiamo il numero reale di livelli h ma usiamo un numero ottimale h_{\min} , e invece di usare il numero reale di regioni m usiamo l'ottimo m_{\min}

$$rfat(T) = \frac{I_C - nh_{\min}}{n} \cdot \frac{1}{m_{\min} - h_{\min}}$$

I_C – total number of nodes accessed

C – capacity of a node in objects

Minimum height: $h_{\min} = \lceil \log_C n \rceil$

Minimum number of nodes: $m_{\min} = \sum_{i=1}^{h_{\min}} \left\lceil \frac{n}{C^i} \right\rceil$

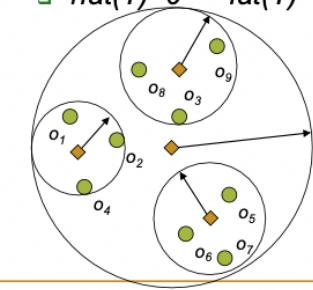
Esempio:

- Two trees organizing 9 objects:

- $n=9$ $C=3$ $h_{min}=2$ $m_{min}=4$

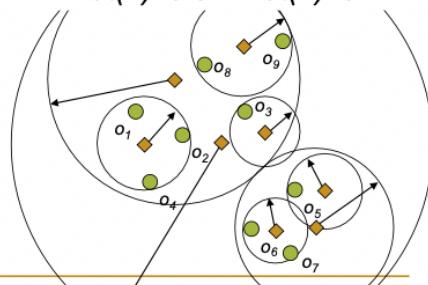
- Minimum tree

- $I_C=18$ $h=2$ $m=4$
- $rfat(T)=0$ $fat(T)=0$



- Non-optimal tree

- $I_C=27$ $h=3$ $m=8$
- $rfat(T)=0.5$ $fat(T)=0$



Nonostante il fat factor sia 0 per entrambi il relative fat factor è differente perché nell'esempio di destra abbiamo più nodi di quelli realmente necessari (cioè il caso a sx) e di conseguenza il relative fat factor sarà più alto.

Tree Quality Measures: Conclusion

- Absolute fat-factor

- $0 \leq fat(T) \leq 1$
- Region overlaps on the same level are measured.
- Under-filled nodes are not considered.
- Can this tree be improved?*

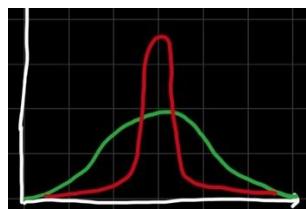
- Relative fat-factor

- $rfat(T) \geq 0$
- Minimum tree is optimal
- Overlaps and occupations are considered.
- Which of these trees is more optimal?*

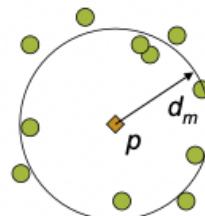
Choosing Reference Points

Tutte le index structure, tranne i naïve, hanno bisogno di pivot (reference objects), essi sono essenziali per partizionare gli oggetti e fare ricerca con pruning (non cerco tutto ma taglio quando posso evitando di fare ricerche inutili).

Quello che vogliamo capire è **come scegliere i pivot** perché a seconda di come li scegliamo le performance vengono influenzate. Infatti, prendendo la distance density da un pivot se essa è alta e stretta (rosso nell'immagine) allora avrò molti punti circa alla stessa distanza e quando applico una query dovrò accedere ad entrambi i subset (dentro e fuori dalla ball, vedi dopo). Quindi quello che voglio è una distribuzione gaussiana più larga (verde) dove gli oggetti sono distribuiti su distanze diverse.

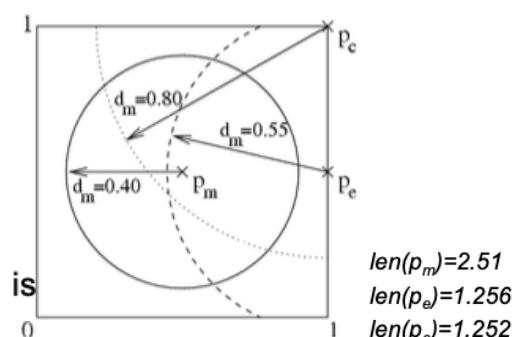


Nell'esempio sotto vediamo una rappresentazione della distance density rossa, i punti circa alla stessa distanza d_m , quando vado ad applicare una query è molto probabile che debba accedere ai due subset, sia dentro che fuori dalla ball.

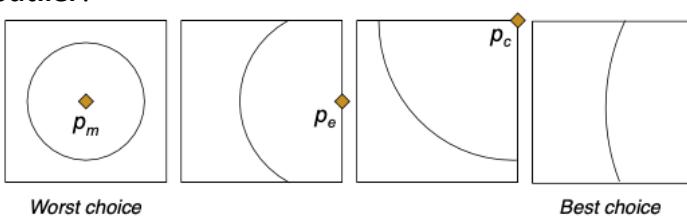


Vediamo ora un esempio per capire meglio, prendiamo un quadrato di dimensione unitaria uniformemente distribuito (stessa probabilità in ogni punto di trovare un oggetto), e consideriamo tre possibili posizioni di pivot: centro, angolo e bordo; consideriamo invece come raggio d_m la mediana (cioè quel raggio tale per cui metà oggetti sono dentro la ball e metà sono fuori).

Si può intuire che *la probabilità di accedere ad entrambi i subset è proporzionale alla lunghezza del bordo della ball*, vediamo con le lunghezze riportate a fianco che la migliore scelta è il pivot nell'angolo mentre la peggiore è metterlo nel centro.

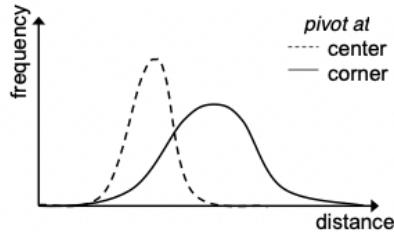


Per ottenere il bordo più corto possibile dobbiamo prendere un punto fuori dallo spazio, cioè un **outlier!**



p_o

Come detto precedentemente possiamo definire la bontà di un pivot confrontando la distance density, scegliendo il pivot nel corner infatti la densità è più schiacciata rispetto al pivot centrale che è più alta e stretta.



Quindi i pivot buoni dovrebbero essere outlier dello spazio oppure oggetti vicini al bordo dello spazio. Scegliere i pivot non è però semplice perché di solito richiede una complessità quadratica o cubica, spesso infatti vengono scelti random anche se questa scelta è banale e non ottimizzata.

Heurist to find One Pivot

Dato che non c'è una definizione per corner nel metric space, consideriamo come corner un oggetto lontano dagli altri.

Algoritmo per un outlier:

1. Prendi un oggetto random
2. Calcola la distanza dall'oggetto preso a tutti gli altri
3. Prendi l'oggetto più lontano come pivot

Questo non ci garantisce la migliore scelta possibile ma ci aiuta a scegliere un pivot migliore di uno preso random, ma ci porta ad un guadagno di performance circa del 5%-10%.

Choosing more pivots

Scegliere più pivot è più complicato perché essi dovrebbero stare circa equamente distanti.

Algoritmo per scegliere m pivots:

1. Prendi $3m$ oggetti random dagli n del set
2. Prendi un oggetto, calcola le distanze con tutti gli altri e prendi come primo pivot il più lontano.
3. Il secondo pivot viene scelto come l'oggetto più lontano dal primo
4. Il terzo è l'oggetto più lontano dai pivot precedenti, cioè quello che massimizza il minimo $\min(d(p_1, p_3), d(p_2, p_3))$. Cioè voglio massimizzare la distanza tra il pivot che cerco e il pivot a lui più vicino.
5. E così via fino ad ottenere m pivots.

Questo algoritmo richiede $O(3m^2)$ calcoli di distanza, per un piccolo valore di m possiamo ripetere l'algoritmo più volte prendendo diversi $3m$ oggetti di partenza, andando poi a scegliere il migliore.

Vediamo nel prossimo paragrafo come valutare il "miglior set".

Efficiency Criterion

Dato un set di pivot $P = \{p_1, \dots, p_n\}$, la **pivoted distance** (lower bound di d) viene definita come

$$d_P(o, o') = L_\infty(\Psi(o), \Psi(o')) = \max_i |d(p_i, o) - d(p_i, o')|$$

Dove $\Psi(o) = (d(o, p_1), \dots, d(o, p_n))$

Dato che $d_p(o, o') \leq d(o, o')$, una buona scelta di P dovrebbe massimizzare il lower bound d_p .

Per ottenerlo dobbiamo massimizzare la **media della pivoted distance** μ_{dp} .

Dati due set di pivot P_1 e P_2 , **P1 sarà meglio se $\mu_{dp1} > \mu_{dp2}$** .

Dato un set $P = \{p_1, \dots, p_n\}$ andiamo a stimare μ_{dp} per P:

1. At random choose l pairs of objects $\{(o_1, o'_1), (o_2, o'_2), \dots, (o_l, o'_l)\}$ from database $X \subseteq \mathcal{D}$
2. Map all pairs into the pivoted space of the set of pivots P
 $\Psi(o_i) = (d(p_1, o_i), d(p_2, o_i), \dots, d(p_t, o_i))$
 $\Psi(o'_i) = (d(p_1, o'_i), d(p_2, o'_i), \dots, d(p_t, o'_i))$
1. For each pair (o_i, o'_i) compute their pivoted distance:
 $d_i = L_\infty(\Psi(o_i), \Psi(o'_i))$.
2. Compute μ_{dp} as the mean of d_i : $\mu_{dp} = \frac{1}{l} \sum_{1 \leq i \leq l} d_i$

Selects further pivots “on demand”

- Based on efficiency criterion μ_{dp}

Algorithm:

1. Select a sample set of m objects.
2. $P_1 = \{p_1\}$ is selected from the sample as μ_{dp1} is maximum.
3. Select another sample set of m objects.
4. Second pivot p_2 is selected as: μ_{dp2} is maximum where $P_2 = \{p_1, p_2\}$ with p_1 fixed.
5. ...

Total cost for selecting k pivots: $2lmk$ distances

- Next step would need $2lm$ distance, if distances d_i for computing μ_{dp} are kept.

Summary on Choosing Reference Points

Le regole ad ora sono:

- I pivot buoni sono quelli **distanti dagli altri oggetti** nel metric space
- I pivot buoni sono **distanti tra loro**

L'euristica a volte può fallire, ad esempio un dataset con il coefficiente di Jaccard, il principio di outlier seleziona un pivot p tale che $d(p, o)=1$ per ogni altro oggetto o del database. Tale pivot è inutile per partizionare e filtrare.

Exact similarity searching methods

Ball partitioning methods

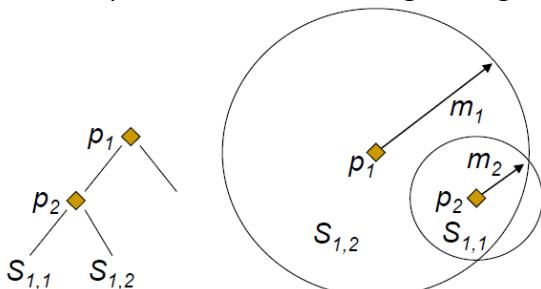
Si costruisce il **Vantage point tree**, vantage point è un altro nome per chiamare i pivot/reference object. Per costruire questo albero si usa il ball partitioning dividendo ricorsivamente il dataset X. Ad ogni step si sceglie un pivot dal dataset X, poi si calcola la **mediana m**. Questa distanza rappresenta il raggio che, se applicato al punto p scelto, divide il dataset in due parti uguali, metà degli oggetti sarà interno alla regione e l'altra metà no.

In maniera più formale vediamo come viene diviso un dataset:

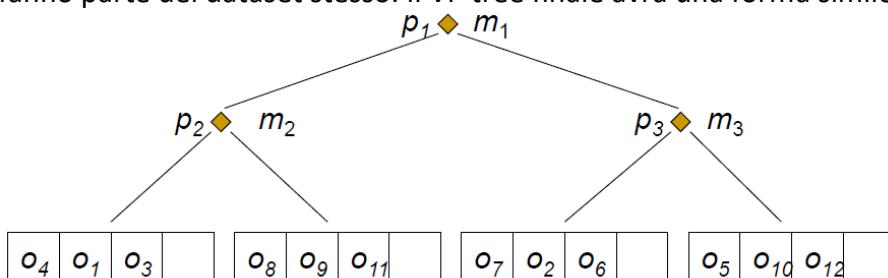
$$S_1 = \{x \in X - \{p\} \mid d(x, p) \leq m\}$$

$$S_2 = \{x \in X - \{p\} \mid d(x, p) \geq m\}$$

Il segno di uguaglianza serve per assicurare il bilanciamento. Questa azione è ricorsiva, quindi alla seconda iterazione si andrà a partizionare ad esempio S_1 . Otterremo quindi un albero bilanciato simile a quello mostrato nella figura seguente.



Uno o più oggetti possono essere posizionati in una foglia. Si crea quindi un albero binario bilanciato, la struttura è statica e i pivot che servono per partizionare il dataset sono punti che fanno parte del dataset stesso. Il VP tree finale avrà una forma simile a quella mostrata in figura.



Vediamo come possiamo valutare una **range query** utilizzando un VP tree, data la query $R(q, r)$ si comincia ad esplorare l'albero dalla radice e poi si valuta nodo per nodo, ogni nodo (p_i, m_i) devo analizzare tre casi:

- Se $d(q, p_i) \leq r$: inserisco p_i nell'output
- Se $d(q, p_i) - r \leq m_i$: esploro il sottoalbero sinistro
- Se $d(q, p_i) + r \leq m_i$: esploro il sottoalbero destro

In altre parole, esploro il sottoalbero sinistro se la query interseca l'interno della regione, altrimenti esploro il sottoalbero destro.



In maniera simile possiamo applicare una query di tipo kNN, l'unica differenza è che dobbiamo usare due variabili di supporto dato che in questo tipo di query abbiamo il raggio variabile iterazione per iterazione.

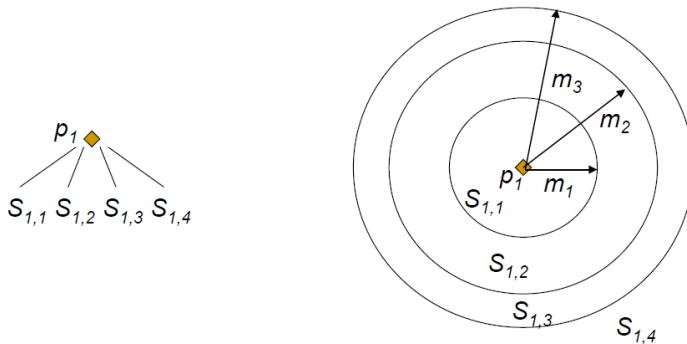
Inizializziamo quindi $d_{NN}=d_{MAX}$ e $NN = \text{null}$, seguiamo gli stessi principi per esplorare l'albero e per ogni nodo avremo i tre casi:

- Se $d(q, p_i) \leq d_{NN}$: setto $d_{NN}=d(q, p_i)$ e $NN=p_i$
- Se $d(q, p_i) - d_{NN} \leq m_i$: esploro il sottoalbero sinistro
- Se $d(q, p_i) + d_{NN} \geq m_i$: esploro il sottoalbero destro

L'esempio appena illustrato riguarda la ricerca dell' 1-NN, se volessimo espandere la ricerca a K vicini basta rendere le due variabili (d_{NN} e NN) due vettori ($d_{NN}[k]$ e $NN[k]$) che sono ordinati in base alla distanza dei punti dalla query q .

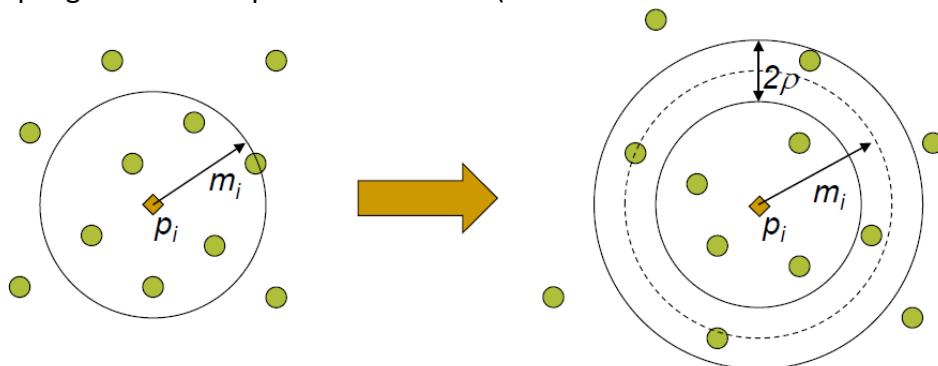
Multi-Way Vantage Point Tree

Eredita tutte le caratteristiche di un normale VP tree, cambia il modo di partizionare. Ogni nodo non viene più diviso in maniera binaria ma abbiamo la possibilità di individuare diverse partizioni come possiamo vedere nell'esempio successivo.

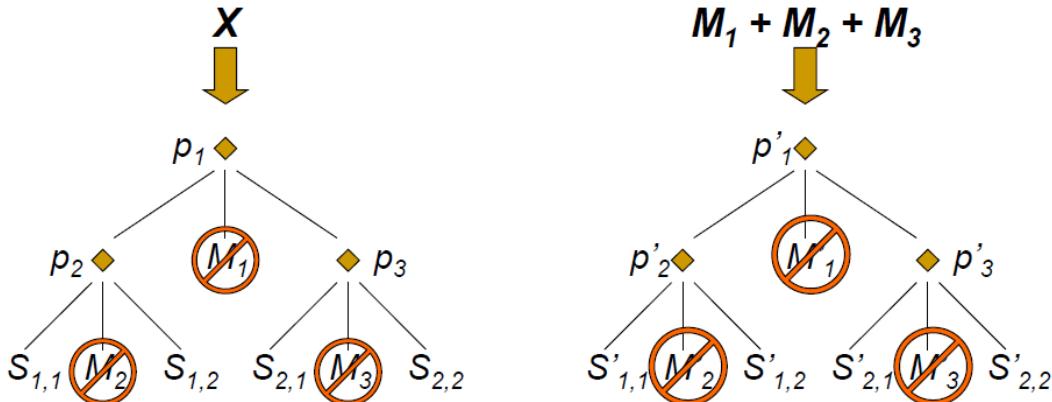


Vantage Point Forest (VPF)

Si tratta di un'ulteriore variazione dell'algoritmo di base, si utilizza il concetto di soglia per creare una partizione centrale che poi viene esclusa dal processo di costruzione dell'albero. Nella figura si nota che la corona circolare centrale contenente i 4 oggetti viene esclusa per essere poi processata in un secondo momento. Questa tecnica può essere utile, ad esempio, se abbiamo una narrow distribution nella corona, ciò ci permette di analizzare gli altri punti con un normale albero e poi gestire i molti punti nella corona (dato che la distribuzione è narrow in essa).



Dopo aver partizionato lo spazio in sottoregioni, le zone escluse vengono unite e valutate attraverso un secondo albero che adotta mediane totalmente differenti, si applica lo stesso ragionamento e quindi con le zone escluse dal secondo albero se ne può formare un terzo e così via, si formerà quindi una foresta di alberi.

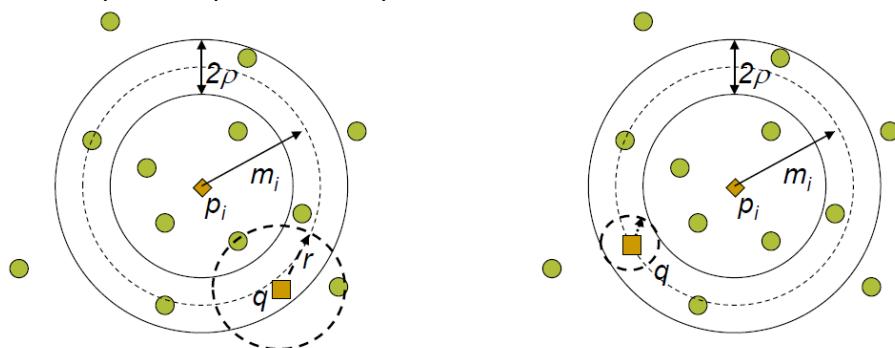


Per analizzare una range query utilizzando questo approccio si parte dal primo albero sempre dal root e si analizza nodo per nodo, ad ogni nodo ci possono capitare i casi illustrati nella seguente figura, grazie ai quali si può capire come dobbiamo esplorare l'albero e se dobbiamo analizzare anche quello successivo oppure no.

- if $d(q, p_i) \leq r$
- if $d(q, p_i) - r \leq m_i - \rho$
- if $d(q, p_i) + r \geq m_i - \rho$
- if $d(q, p_i) + r \geq m_i + \rho$
- if $d(q, p_i) - r \leq m_i + \rho$
- if $d(q, p_i) - r \geq m_i - \rho$ and
 $d(q, p_i) + r \leq m_i + \rho$

- report p_i
- search the left sub-tree
- search the next tree !!!
- search the right sub-tree
- search the next tree !!!
- search only the next tree !!!

Se ad esempio la query intersecasse tutte e tre le partizioni ad un determinato nodo dovremmo esplorare le due parti dell'albero attuale e dovremmo inoltre esplorare il VP tree successivo, se invece la query dovesse intersecare solo la zona esclusa dell'albero attuale allora possiamo interrompere l'esplorazione e passare direttamente all'albero successivo.



Exploiting pre-computed distances

Quando si crea la struttura inserendo un oggetto per volta si deve calcolare alcune distanze, l'idea è quella di salvarsi queste distanze in modo da sfruttarle quando vengono analizzate le query.

AESA

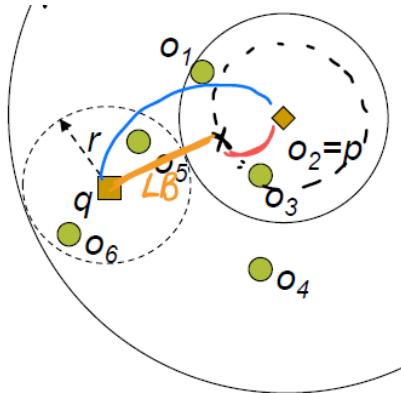
L'idea alla base è evitare più calcoli di distanze possibili scartando in base ai LB e UB.

AESA sta per *approximating and eliminating search algorithm*, si salva la matrice delle distanze fra tutti gli oggetti del db, la matrice sarà quindi $n \times n$, da momento che le distanze sono simmetriche si può salvare solo metà della matrice $\frac{n(n-1)}{2}$.

	O_1	O_2	O_3	O_4	O_5	O_6
O_1	0	1.6	2.0	3.5	1.6	3.6
O_2	1.6	0	1.0	2.6	2.6	4.2
O_3	2.0	1.0	0	1.6	2.1	3.5
O_4	3.5	2.6	1.6	0	3.0	3.4
O_5	1.6	2.6	2.1	3.0	0	2.0
O_6	3.6	4.2	3.5	3.4	2.0	0

Con questo approccio ogni punto del database può essere considerato un pivot.

Vediamo adesso come processare una range query $R(q, r)$. Per prima cosa si sceglie un oggetto casuale come pivot p , si calcola al distanza del pivot dalla query $d(q, p)$. Ricordiamo che $|d(q, p) - d(p, o)|$ è un lower bound per la distanza reale $d(q, o)$ di un oggetto casuale o dalla query. Tramite il lower bound si possono filtrare i punti, in particolare se $LB > r$ il punto non sarà sicuramente incluso nel risultato.



Il LB ci dà il caso migliore parlando del posizionamento di un oggetto rispetto alla query, se questa approssimazione è maggiore del raggio della query (es. se prendiamo o_3) allora siamo sicuri di poter scartare tale oggetto.

Successivamente viene scelto un nuovo pivot, per scegliere il pivot ottimale dovremmo prendere quello più vicino alla query. L'idea di base è quella di non calcolare distanze, quindi per scegliere il nuovo pivot si utilizza il LB. Dato che per ogni oggetto nel db abbiamo il valore di LB il nuovo pivot sarà quello che ha il LB minore. Si ripete il processo di filtraggio per eliminare altri oggetti. Si continua nella scelta dei pivot fino a che il numero di oggetti rimanenti è accettabile oppure fino a quando tutti gli oggetti sono stati scelti come pivot. Come ultima cosa si deve controllare le distanze fra gli oggetti rimanenti e la query per formare il risultato. Chiaramente si può anche usare l'upper bound per determinare gli oggetti che sicuramente sono nel dataset.

LAESA

Il problema di AESA è quello di essere quadratico rispetto al numero di oggetti, questo porta al metric space a non poter essere salvato per intero in memoria. Si usa LAESA per provare a risolvere questo problema, l'idea è quella di salvare le distanze solo con gli oggetti che possono avere la possibilità di essere pivot. Solo un sotto insieme di m elementi possono diventare pivot e non tutti gli elementi come nell'approccio precedente. I pivot scelti dovrebbero essere molto

distanti fra loro. Per filtrare prima vengono usati tutti gli m pivot disponibili e successivamente vengono valutate le distanze fra la query e gli oggetti che non sono stati scartati.

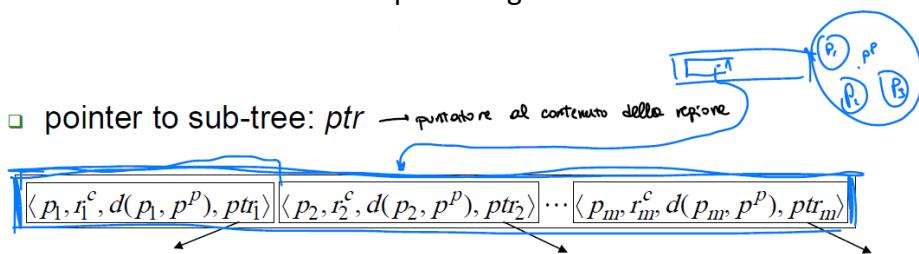
The M-tree

Vediamo adesso un esempio di una struttura utilizzata per salvare le distanze e in generale la struttura del metric space. L'M-tree è una struttura dinamica, quindi permette inserimento e rimozione di oggetti, è disk oriented nel senso che ogni nodo ha una dimensione fissata (spesso la dimensione del blocco del disco). L'albero è costruito in maniera **bottom-up**.

Tutti i dati sono salvati nelle foglie mentre nei nodi interni sono presenti i puntatori ai sotto alberi successivi e contengono anche altre informazioni utili.

Vediamo adesso cosa è salvato in un nodo interno dell'albero. Un **nodo interno** contiene una entry per ogni sottoregione, **ogni entry** contiene:

- Il pivot p della regione
- Il raggio della regione
- La distanza fra p e p^P che corrisponde al pivot della regione padre
- Il puntatore al nodo interno che contiene tutte le informazioni relative alle regioni contenute all'interno di questa regione

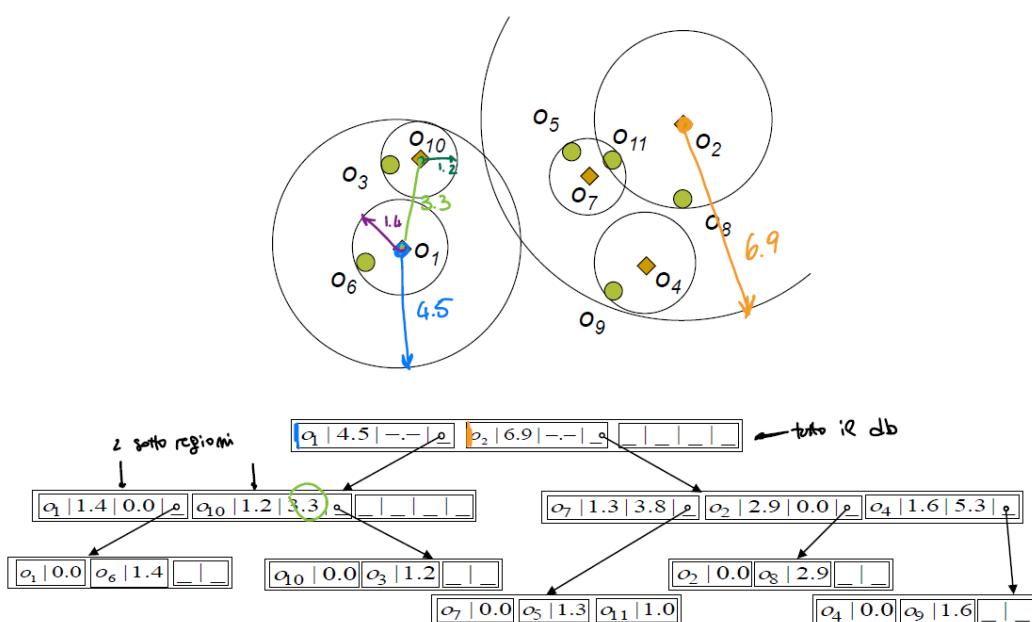


Nell'immagine possiamo capire meglio il senso del puntatore al prossimo nodo.

Passando al nodo foglia, ogni **leaf node** contiene delle entry contenenti i dati. In particolare, ogni entry corrisponde ad una coppia:

- Un oggetto o
- La distanza fra o e il pivot della regione padre: $d(o, o^P)$

Nella prossima immagine possiamo vedere un esempio di un M-Tree totalmente costruito insieme alla struttura visiva del metric space.



M-Tree Insert

Per aggiungere un oggetto o_N si deve scendere ricorsivamente nell'albero per individuare la foglia che più si addice a contenere il nuovo oggetto.

Ad ogni step si accede al sottoalbero che ha il pivot p e che:

- Non richiede aumentare il raggio della regione: $d(o_N, p) \leq r^c$. In altre parole, si sceglie il subtree che contiene il nuovo oggetto. In caso di parità si sceglie il subtree più vicino all'oggetto.
- **OPPURE**, si sceglie il subtree che richiede l'aumento minimo del raggio della regione.

Quando si raggiunge una foglia, **se la foglia non è piena**, si inserisce l'oggetto. Se la foglia risulta piena è necessario che la foglia venga divisa. Vediamo come possiamo fare la **divisione di una foglia**.

Per prima cosa definiamo **S** il set di entry presenti nella foglia che deve essere divisa, prendiamo due pivot p_1 e p_2 dal set S. Partizioniamo S in S_1 e S_2 tramite i pivot appena scelti. Salviamo S_1 nella foglia che doveva essere divisa in principio e allochiamo una nuova foglia per S_2 . Dobbiamo aggiungere una entry ricorsivamente lungo tutto l'albero dal momento che è stata creata una nuova foglia, se il nodo che deve essere diviso è la radice allora si crea lo split e si aggiunge una nuova radice che fa da padre alle due partizioni.

- if N is root
 - allocate a new root and store there entries for p_1, p_2
- else (let N^p and p^p be the parent node and parent pivot of N)
 - replace entry p^p with p_1
 - if N^p is full then **Split**(N^p, p_2)
 - else store p_2 in node N^p

In figura sono mostrati gli aggiustamenti da fare lungo l'albero ricorsivamente partendo dalla foglia aggiunta.

Ci sono diverse politiche per scegliere i due pivot, nell'immagine seguente ne sono mostrate alcune. Quello che conta è che tutte possono essere applicate in due modi:

- Confirmed: si usa il pivot originale e se ne sceglie solo uno nuovo
- Uncorffimed: si selezionano due nuovi pivot

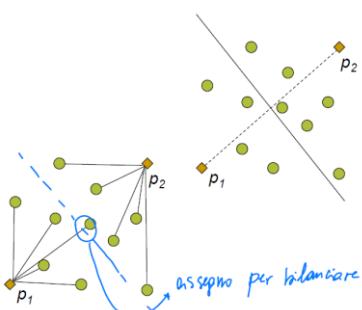
RANDOM – select pivots p_1, p_2 randomly

m_RAD – select p_1, p_2 with minimum $(r_1^c + r_2^c)$

mM_RAD – select p_1, p_2 with minimum $\max(r_1^c, r_2^c)$

M_LB_DIST – let $p_1 = p^p$ and $p_2 = o_i \mid \max_i \{ d(o_i, p^p) \}$

Quando facciamo lo split fra i due pivot possiamo semplicemente usare un hyperplane che divide gli oggetti in base alla distanza, in questo modo però le partizioni non sono bilanciate. Se vogliamo ottenere un bilanciamento utilizziamo un vincolo rilassato, gli oggetti che hanno distanza più o meno uguale rispetto ai due pivot vengono inseriti in una o l'altra partizione in modo da garantire il bilanciamento.

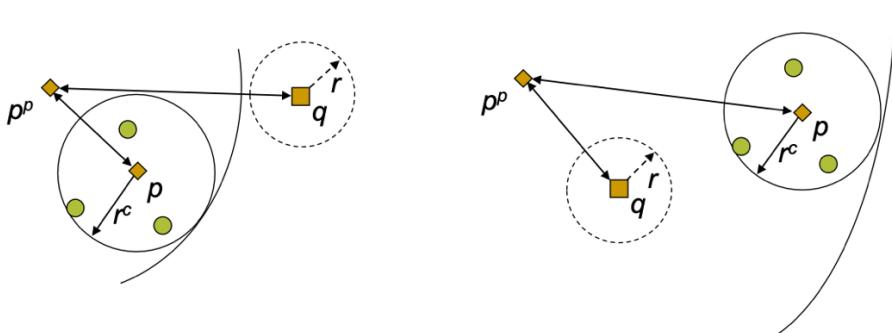


M-Tree: Range Search

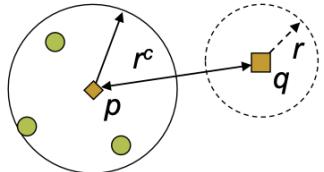
Adesso vogliamo applicare una range query all'M-tree, l'idea alla base è se c'è una sovrapposizione tra la query e la regione allora devo controllare, altrimenti non ci entro tanto sicuramente la query non può prendere nessun punto all'interno di quella regione.

Per vedere se c'è intersezione basta calcolare la distanza tra il centro della query e della regione e vedere se è maggiore o minore della somma dei due raggi. In alcuni casi però possiamo risparmiarci il calcolo della distanza, sfruttando $d(p,pp)$ che è salvata nell'entry dell'albero.

Ricapitolando, data una query $R(q,r)$, attraversiamo l'albero e per ogni entry $\langle p, r^c, d(p,pp), \text{ptr} \rangle$ e facciamo pruning del sottoalbero se $|d(q,pp) - d(p,pp)| - r^c > r$ (come nell'immagine, evitandoci il calcolo della distanza),



altrimenti calcoliamo $d(q, p)$ e facciamo pruning del sottoalbero se $d(q,p) - r^c > r$:



Andiamo poi a cercare ricorsivamente in tutte le entrate non-pruned.

Quando arriviamo ad una **leaf node** per ogni entry $\langle o, d(o,op) \rangle$:

- Ignoriamo l'entry se $|d(q,op) - d(o,op)| > r$ (sappiamo già le due distanze, la prima è calcolata quando abbiamo deciso di entrare nella regione, la seconda è scritta nella entry)
- Altrimenti calcoliamo $d(q,o)$ e controlliamo se $d(q,o) \leq r$

M-Tree: k-NN Search

La differenza con il range search è che qui il raggio non è fisso ma viene incrementato ad ogni step.

Data una query k-NN(q)

Usiamo una **priority queue** che salva le regioni che con il raggio attuale vengono intersecate, scartando tutte le altre, ovviamente ogni volta che inseriamo un nuovo oggetto questa coda deve essere aggiornata. In particolare la priority queue:

- Salva i puntatori ai sottoalberi dove si possono trovare oggetti, cioè quelle regioni che intersecano la query
- Vengono salvate le regioni in base alla distanza minima tra la query e il bordo più vicino delle regioni, più formalmente: considerando un entry $E = \langle p, r^c, d(p,pp), \text{ptr} \rangle$ la coppia $\langle \text{ptr}, d_{\min}(E) \rangle$ viene salvata
- Dove $d_{\min}(E) = \max\{d(p,q) - r^c, 0\}$

Bulk-Loading Algorithm

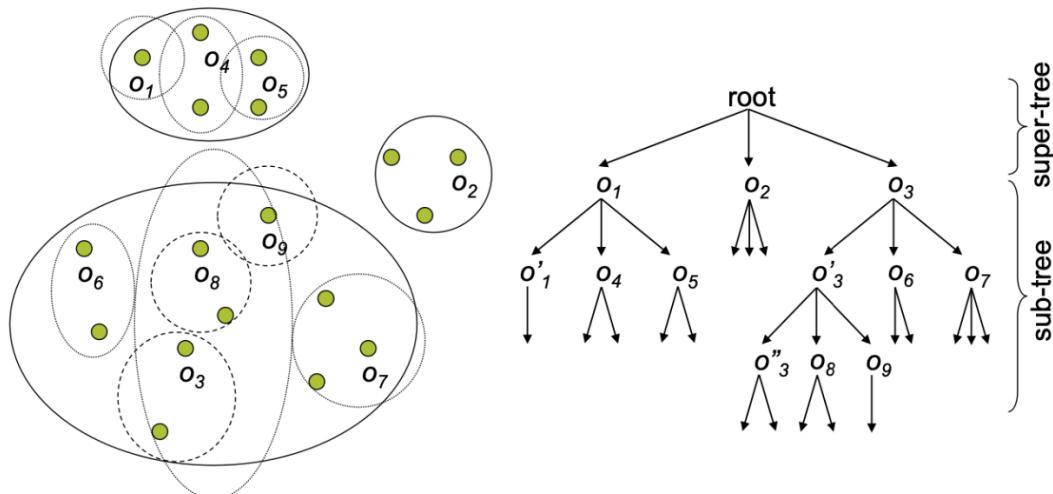
È un miglioramento nella costruzione (inserimento) dell'M-tree, è **necessario** però conoscere l'intero dataset fin dall'inizio.

Il **Bulk-Loading** algorithm si divide in due fasi:

- Prima fase: crea l'M-tree
- Seconda fase: rifinisce (ottimizza) l'albero non bilanciato

Dato un dataset $X=\{o_1, \dots, o_n\}$ e un numero di entries per nodo pari a m , selezioniamo un numero di l di pivot $P=\{p_1, \dots, p_l\}$ da X dove $l=m$, ed assegniamo ogni oggetto di X al pivot più vicino creando l subset P_1, \dots, P_l .

Applichiamo **ricorsivamente** questo procedimento ai l subset che abbiamo trovato fino ad arrivare alle foglie, cioè quando il numero di oggetti in un subset è al massimo m (numero max entry per nodo), poi creiamo il root node e ci attacchiamo tutti i subset trovati, in figura un esempio.

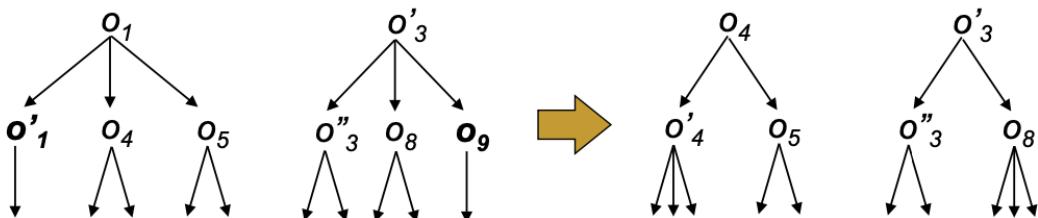


Abbiamo però dei **problem**i a seconda della scelta dei pivots, infatti se:

- Abbiamo regioni sparse allora il sottoalbero è piccolo e corto
- Abbiamo regioni dense allora il sottoalbero è profondo (vedi il sotto albero di o_3)

Tutto ciò porta ad avere un M-tree **sbilanciato**, il ribilanciamento avviene nella seconda fase dove applichiamo due tecniche per aggiustare il set di pivot P :

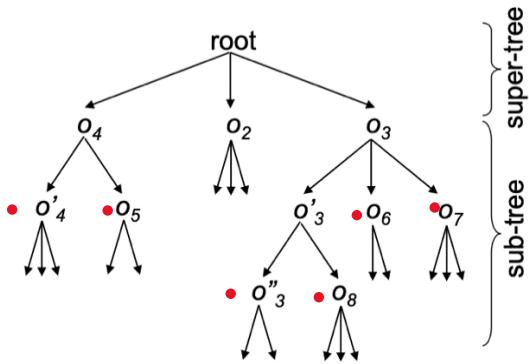
- **Under-filled node** (nodi che hanno molte entry libere, come o'_1 e o_9 sopra): riassegnamo questi oggetti ad altri pivot e eliminiamo i corrispondenti pivot da P .



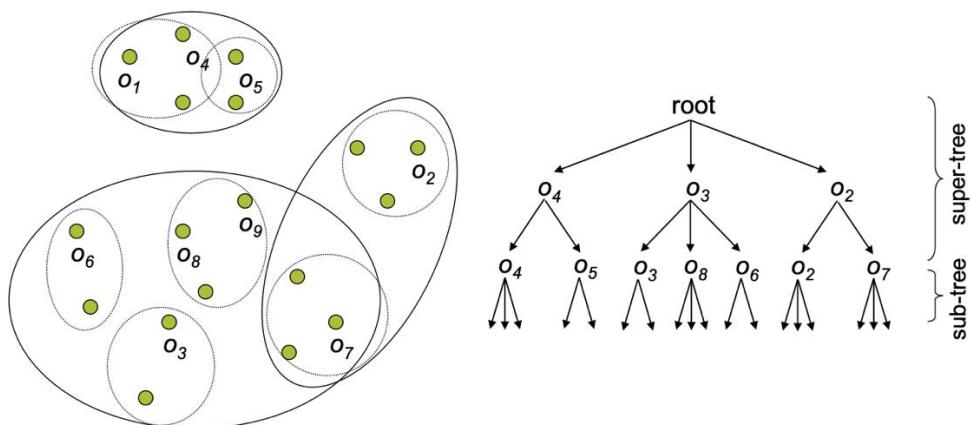
Vediamo che o'_1 ha una sola entry occupata quindi eliminiamo il pivot o_1 , scegliamo come pivot o_4 ed assegnamo ad o'_1 l'oggetto o_1 .

Anche o_9 ha una sola entry occupata quindi lo spostiamo sotto o_8 (il nodo più vicino e con spazio).

Ottenendo una struttura più **compatta**:



- **Deeper subtrees:** analizziamo i sottoalberi più profondi, rispetto alla root abbiamo o_3 e o_4 più profondi rispetto a o_2 , quindi prendiamo questi due sottoalberi e li dividiamo in nuovi sottoalberi con root in tutti i nodi più profondi o'_4 , o_5 , o''_3 , o_8 , o_6 , o_7 (segnati in rosso sopra). A questo gli aggiungiamo tutti a P scartando o_3 e o_4 , praticamente li stiamo mettendo tutti allo stesso livello. Adesso $P = \{o_2, o'_4, o_5, o''_3, o_8, o_6, o_7\}$.



Adesso abbiamo un albero bilanciato, perché tutti i path dal root alle foglie hanno lo stesso numero di livelli.

Optimization of Bulk-Loading

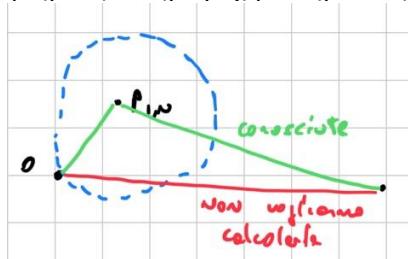
Sfruttando come al solito i LB possiamo ridurre dei costi di computazione nelle chiamate ricorsive dell'algoritmo.

Dopo la fase iniziale abbiamo le distanze $d(p_j, o_i)$ per tutti gli oggetti $X = \{o_1, \dots, o_m\}$ e tutti i pivot $P = \{p_1, \dots, p_l\}$, assumendo il processo ricorsivo su P_1 si prende un nuovo set di pivot $\{p_{1,1}, \dots, p_{1,l'}\}$ dividendo P_1 . Per decidere i nuovi pivot dobbiamo misurare le distanze tra i nuovi pivot e tutti gli oggetti appartenenti a P_1 , possiamo sfruttare il **lower bound** sulla distanza $d(p_{1,j}, o_i)$:

$$|d(p_{1,j}, o_i) - d(p_{1,N}, p_{1,j})| \leq d(p_{1,j}, o_i).$$

Se il lower bound è più grande della distanza dal pivot più vicino fino ad ora $p_{1,N}$, cioè:

$$|d(p_{1,j}, o_i) - d(p_{1,N}, p_{1,j})| > d(p_{1,N}, o_i) \text{ allora possiamo evitare di calcolare } d(p_{1,j}, o_i).$$



Sfruttando i LB riduciamo i costi dell'11%, quando sfruttiamo le distanze già calcolate precedentemente da un singolo pivot, e dell'20% quando usiamo le distanze con più pivot.

Multi-Way Insertion Algorithm

Un'altra estensione dell'algoritmo di inserimento dell'M-tree è il **multi-way**, l'obiettivo è quello di andare costruire alberi più compatti **riducendo il costo di ricerca** (sia I/O che CPU). Questo algoritmo può gestire anche **dataset dinamici**, cioè che non devono necessariamente essere disponibili in anticipo ma si può aggiungere nuovi elementi dinamicamente, a discapito però di un costo di inserimento leggermente maggiore.

Nell'inserimento *single-way* si va a visitare solamente un ramo del root (un solo path nell'albero), quello più vicino all'oggetto da inserire o quello che porta ad un incremento minimo del raggio della regione, questo però non è sempre la scelta più conveniente e per questo motivo è stato introdotto il **multi-way** che ci permette di seguire simultaneamente più path sull'albero andando a considerare tutte le foglie che intersecano l'oggetto da inserire, scegliendo poi la migliore.

Quando vogliamo **inserire** un nuovo oggetto o_N :

- Facciamo una *point query* $R(o_N, 0)$, che restituisce tutti i leaf node che contengono il punto o_N
- Per tutte le leaf visitate andiamo a calcolare la distanza tra o_N e i pivot delle leaf, andando a scegliere quello più vicino
- Se nessuna leaf è visitata, cioè quando l'oggetto da inserire non è intersecato da nessuna regione, allora procedi con l'inserimento *single-way*

Il **costo di inserimento** di I/O è più alto del 25%, perché dobbiamo esaminare più nodi, ed anche il costo di CPU è maggiore (dobbiamo calcolare più distanze).

Il **costo di ricerca** invece:

- Diminuisce del 15% gli accessi al disco
- Il costo in CPU è circa lo stesso per le *range query*
- Diminuisce del 10% il calcolo delle distanze per le *k-NN query*

The Slim Tree

È un'altra estensione dell'M-tree, migliora la velocità di inserimento, del node splitting e migliora anche l'utilizzo dello spazio creando alberi più compatti. Tutto questo è possibile grazie ad un'euristica nella **scelta del nodo** per l'inserimento, ad un nuovo algoritmo di **node splitting** e una procedura per il **post-processing** per rendere gli alberi più compatti.

Insertion

Partendo dal root node, per ogni step:

- Troviamo un nodo che copre l'oggetto da inserire
- Se nessun nodo lo copre allora scegliamo quello che ha il pivot più vicino, invece l'M-tree avrebbe scelto quello che permette il minor allargamento
- Se più nodi coprono l'oggetto allora selezioniamo il nodo che occupa meno spazio (la regione è più piccola), invece l'M-tree avrebbe scelto il nodo con il pivot più vicino

Stiamo quindi dando la priorità ai *nodi più piccoli*, quelli che hanno più spazio, scartando quelli già abbastanza pieni che potrebbero poi portare a splitting. In tal modo si riduce anche la dimensione dell'albero.

Dei risultati sperimentali mostrano che abbiamo costi di I/O più bassi e circa lo stesso numero di distanze calcolate, tutto questo sia per la procedura di *costruzione dell'albero* che per l'*esecuzione della query*.

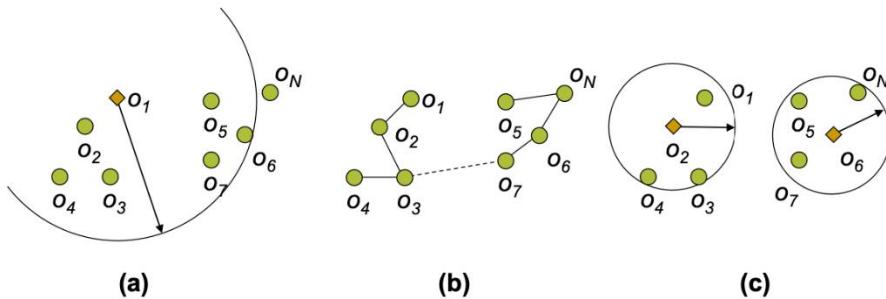
Node Splitting

Fare splitting quando un nodo è pieno ha un alto costo, fino ad ora il meglio che abbiamo è l'algoritmo **mM_RAD_2** che ha complessità $O(n^3)$ usando $O(n^2)$ calcoli di distanze.

Lo Slim Tree splitting si basa sul minimum spanning tree (MST) che ha complessità $O(n^2 \log n)$ usando $O(n^2)$ calcoli di distanze. L'MST assume di avere un grafo full connected e va a scegliere gli archi che hanno costo minimo prendendo tutti i nodi, quindi dati n oggetti avrà $n(n-1)$ edges (distanze tra gli oggetti).

Splitting policy alla base di MST:

- Costruiamo il MST sul grafo
- Eliminiamo l'edge più grande
- I due sotto-grafi formano nuovi nodi
- Per ogni nodo scegliamo come pivot quello che ha distanza minima da tutti gli altri appartenenti allo stesso gruppo



- (a) the original Slim Tree node
- (b) the minimum spanning tree
- (c) the new two nodes

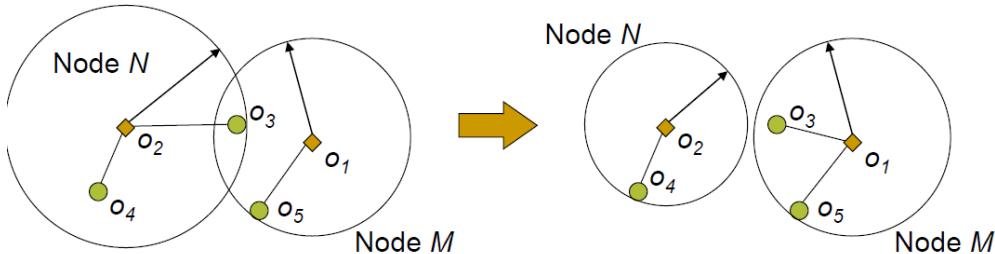
Non è garantito di avere uno split bilanciato, a tal proposito c'è una variante che forza di avere split bilanciati andando a dividere i gruppi prendendo il set di archi più lunghi e scegliamo quello che bilancia di più.

Degli esperimenti provano che costruendo l'albero con MST è almeno **40 volte** più veloce rispetto al mM_RAD_2, l'esecuzione della query invece non viene migliorata significativamente.

Adesso vediamo il post processing del Slim Tree, l'algoritmo di slow down.

Slim-Down Algorithm

È una procedura di postprocessing dello Slim Tree, serve per ridurre il **fat factor** dell'albero. Per fare questo l'idea è quella di ridurre l'intersezione fra i nodi ad un determinato livello. Si deve quindi minimizzare il numero di nodi visitati durante lo svolgimento di una **point query**, nell'esempio sotto si applica $R(o_3, 0)$.



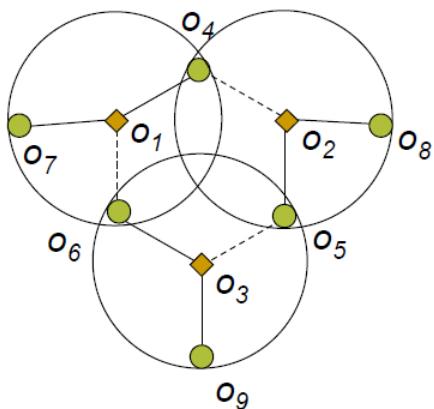
Svolgendo la point query su o_3 si accede a entrambi i nodi, per risolvere questo problema si sposta o_3 nel nodo M e, conseguentemente a questo il raggio del nodo N viene ridotto questo perché o_3 era l'oggetto responsabile del raggio di N, quando questo oggetto viene spostato il raggio si adatta quindi all'oggetto più lontano.

In maniera più precisa l'algoritmo funziona nel seguente modo, per ogni nodo N al livello delle foglie si svolgono i seguenti due passaggi:

1. Si individua l'oggetto o più lontano dal pivot di N
2. Si cerca un nodo M che copre anche lui l'oggetto o. Se M non è pieno, quindi se non si necessita di applicare lo split, si sposta l'oggetto o da N a M. Il raggio di N viene quindi aggiornato.

Questi due passaggi sono applicati a tutti i nodi ad un determinato livello.

In teoria si potrebbero formare dei loop infiniti, come nell'esempio in figura, questo può essere evitato semplicemente aggiungendo un numero fisso di cicli che può eseguire l'algoritmo.



Si è testato che si riduce il costo di I/O di almeno il 10%. Questo approccio è molto utile per evitare di applicare lo split a un nodo pieno, per evitarlo si può spostare un oggetto ad un nodo ad un altro per liberare uno spazio per l'inserimento di un nuovo nodo.

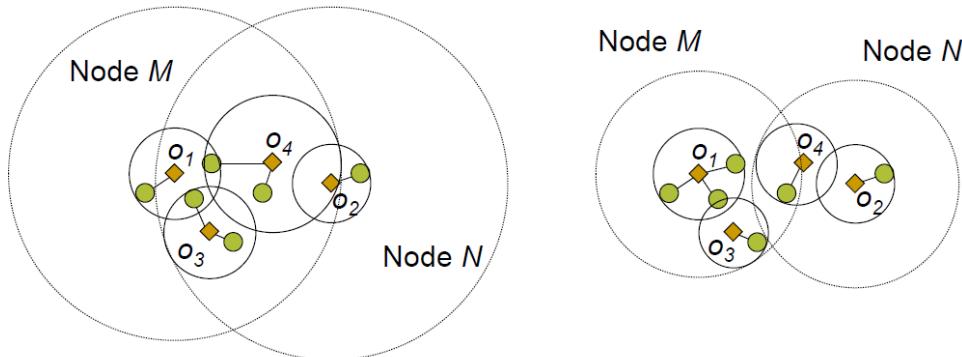
Generalized Slim-Down Algorithm

Si estende il concetto spiegato nella sezione precedente per poter ottimizzare tutti i nodi dell'albero e non solo le foglie. Dato che non si devono più muovere oggetti singoli ma regioni intere, necessitiamo anche i raggi r^c di ogni regione. L'algoritmo generalizzato parte ottimizzando prima le foglie e poi ottimizza livello per livello fino alla radice.

In maniera più specifica, per ogni entry $E = \langle p, r^c, \dots \rangle$ ad un certo livello che non sia quello delle foglie si eseguono i seguenti passi:

1. Si considera la regione (p, r^c)
2. Si esplora l'albero per ottenere tutti i nodi che contengono completamente la regione presa in esame
3. Dal set appena ottenuto si sceglie il nodo M che ha il parent pivot più vicino rispetto al parent pivot della regione alla quale è assegnata la ball region presa in esame. In altre parole, si sposta la regione nel nodo che ha il centro più vicino alla regione
4. Se questo nodo M esiste, si sposta la entry E da N a M
5. Se possibile, si riduce il raggio di N

Nella seguente immagine è riportato un esempio pratico dell'algoritmo generalizzato applicato.



■ Leaf level:

- move two objects from o_3 and o_4 to o_1 – shrink o_3 and o_4

■ Upper level:

- originally node M contains o_1, o_4 and node N contains o_2, o_3
- swap the nodes of o_3 and o_4

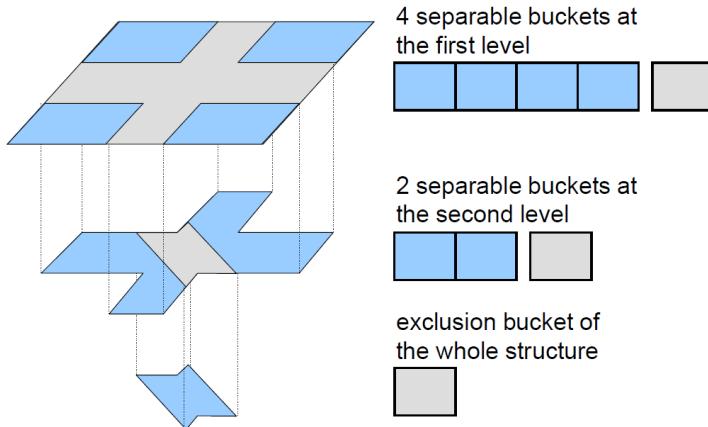
Si parte ottimizzando il livello delle foglie, si spostano gli oggetti più lontani e si ottengono o_1 , o_2 , o_3 e o_4 come sono mostrati a destra. Si passa successivamente al livello superiore, all'inizio M contiene o_1 e o_4 mentre il nodo N contiene o_3 e o_2 . In questo livello si scambiano o_3 e o_4 per ottimizzare.

Hash based methods

Quello che volgiamo creare è una funzione di hash che restituisca lo stesso valore se due oggetti sono simili o vicini nello spazio.

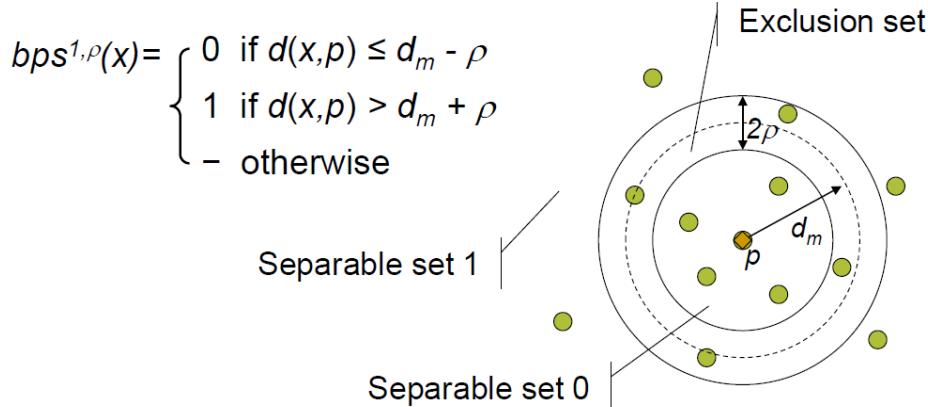
Distance Index (D-Index)

È una struttura ibrida perché per definire questa funzione di hash useremo tecniche basate sul pivot filtering e utilizzeremo la partizione dello spazio, più precisamente l'esclusione della zona centrale. Si utilizzerà la funzione $\rho - split$ tipica della partizione per esclusione della regione centrale. Questo index è una struttura multilivello e, per ogni livello, viene definita una funzione hash differente. Nel primo livello si partitiona l'intero dataset, nei livelli successivi si prende la zona esclusa dalla partizione del livello precedente e vi si applica una nuova funzione di hash, una volta giunti all'ultimo livello, la regione esclusa dall'ultimo partizionamento andrà a formare l'**exclusion bucket**. Vediamo con un esempio pratico il funzionamento.



Nel primo livello si formano quattro bucket che contengono tutti gli oggetti che producono lo stesso risultato di hashing. Vediamo adesso nel dettaglio come si possono creare questi bucket.

Si usa l'excluded middle partitioning, prendendo una ball region come funzione di hash si possono identificare due bucket e una regione esclusa.



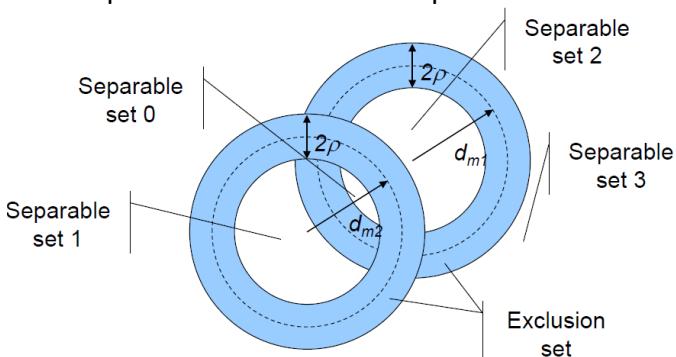
Tutti gli oggetti interni avranno come valore di hash 0, tutti quelli esterni avranno valore 1 mentre gli oggetti appartenenti alla regione interna non avranno nessun valore di hash. Il problema di questa funzione è che si possono **creare solo due bucket per livello**, noi vorremmo crearne molti di più.

Questo tipo di partizionamento è chiamato *binary mapping*, possiede alcune proprietà interessanti. Prima di tutto notiamo che la funzione ha l'indice 1, questo significa che è la funzione di primo ordine. Vediamo le proprietà:

- **Separable Property:** se due oggetti x e y hanno due valori di hash differenti, allora la loro distanza sarà sicuramente maggiore di 2ρ . Questo perché appartengono alle due zone differenti e la loro distanza minima è appunto data dal doppio della soglia.
- **Symmetry property:** dati due margini ρ_1 e ρ_2 tali che $\rho_2 \geq \rho_1$, prendendo due oggetti x e y se x viene classificato con il margine ρ_2 mentre y non viene classificato col margine ρ_1 allora la loro distanza sarà maggiore della differenza fra i margini.

$$bps^{1,\rho}(x) \neq -, bps^{1,\rho}(y) = - \Rightarrow d(x,y) > \rho_2 - \rho_1$$

Vediamo come si può risolvere il problema della creazione di soli due bucket, proviamo a utilizzare contemporaneamente due rho-split function come nell'esempio seguente.



Si nota che vengono formati quattro bucket, questo è possibile grazie ai due risultati delle due funzioni di hash. Si interpretano come numeri binari e si ottengono i 4 split.

Nell'esempio appena fatto abbiamo combinato solo due rho-split di primo livello, generalmente se ne possono combinare quanti vogliamo. In maniera più generale, una combinazione di n rho-split del primo ordine ci restituisce partizioni che vanno da 0 a 2^n-1 , mentre perché un oggetto risulti non classificabile basta che un partitioner lo classifichi come tale.

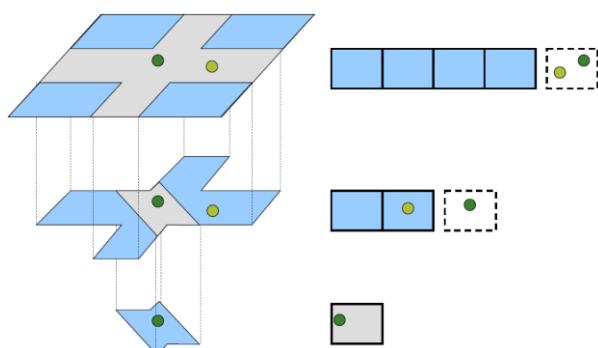
$$bps^{n,\rho}: \mathcal{D} \rightarrow \{0..2^n-1, -\}$$

$$bps^{n,\rho}(x) = \begin{cases} - & \text{if } \exists i, bps_i^{1,\rho}(x) = - \\ b & \text{all } bps_i^{1,\rho}(x) \text{ form a binary number } b \end{cases}$$

Chiaramente, come visto prima, i numeri vanno visti come binari per ottenere l'id del set. Anche utilizzando più rho-binary split insieme si mantengono le due proprietà viste in precedenza.

D-index:Insertion

Vediamo adesso come si inseriscono e come si comprende dove salvare gli oggetti tramite il seguente esempio. Abbiamo definito come si trovano le aree che dividono lo spazio in bucket.



Si può notare che i due oggetti ricadono nella zona esclusa al primo livello di partizionamento; quindi, non appartengono a nessun bucket. Passando al secondo livello si nota che l'oggetto verde chiaro cade in una zona classificabile si va quindi a salvare l'oggetto nel bucket corrispondente, l'oggetto verde scuro cade nuovamente nella regione esclusa. Arrivando all'ultimo livello, quello formato solo da un bucket, l'oggetto verde scuro viene salvato qua.

Di seguito è mostrato l'algoritmo di inserzione che abbiamo appena spiegato tramite l'esempio.

- $Dindex^P(X, m_1, m_2, \dots, m_h)$
 - h – number of levels,
 - m_i – number of binary functions combined on level i .
- Algorithm – insert the object o_N :

```
for i=1 to h do
  if  $bps^{m_i, P}(o_N) \neq -1$  then
     $o_N \rightarrow$  bucket with the index  $bps^{m_i, P}(o_N)$ .
    exit
  end if
end do
 $o_N \rightarrow$  global exclusion bucket.
```

L'oggetto viene inserito accedendo solo a un bucket, mentre con gli altri metodi era necessario passare in rassegna tutta la struttura per capire il nodo giusto dove salvare il nuovo oggetto.

Tuttavia, questo algoritmo necessita di calcolare $\sum_{i=1}^j m_i$ distanze, dove j indica il livello nel quale si è trovato il bucket giusto; queste distanze servono perché è necessario classificare l'oggetto tramite l'insieme di binary function a ogni livello.

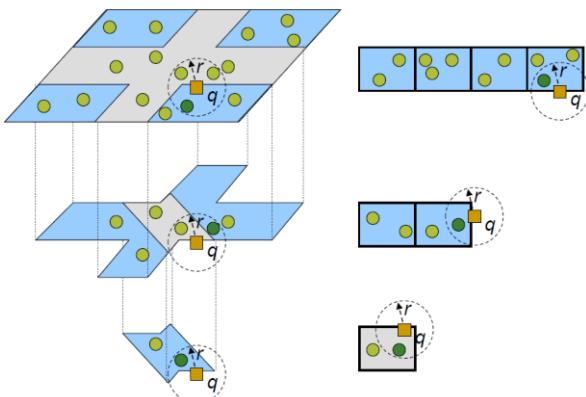
D-index: Range Query

Vediamo adesso come applicare una range query usando questo indice. Per eseguire questo algoritmo il raggio della query **dove** deve essere necessariamente minore del margine, quindi, data una query $R(q, r)$ si cerca nel bucket che contiene il centro della query per ogni livello. Tutti gli oggetti che hanno distanza dal centro minore del raggio vengono riportati come output.

Given a query $R(q, r)$ with $r \leq \rho$:

```
for i=1 to h do
  search in the bucket with the index  $bps^{m_i, 0}(q)$ .
end do
search in the global exclusion bucket.
```

Vediamo meglio con un esempio visivo. Nel primo livello il centro della query è contenuto nell'ultimo bucket, si accede a quello e si cercano i punti che stanno all'interno della regione. È importante notare che si applica una ball region pura senza margine per accedere ad un bucket per ogni livello. Si ripete questo procedimento per ogni livello per recuperare tutti gli oggetti che appartengono al risultato.



Si accede ad un singolo bucket per ogni livello, il costo di esecuzione della query è fissato a $h+1$ accessi dove h è il numero di livelli è l'accesso extra è dovuto al bucket delle esclusioni finali. Volendo il numero di bucket che dobbiamo accedere può essere ridotto con le seguenti due regole:

- Se la regione della query si trova nella zona di esclusione allora si può cercare direttamente nel livello successivo
- Se la regione della query si trova in una regione classificabile allora possiamo interrompere la ricerca perché tutti gli oggetti che devono essere trovati si trovano in quel livello

D-Index: Advanced Range Query

Utilizziamo una versione generale dell'algoritmo visto sopra per poter eliminare il vincolo che il raggio debba essere minore o uguale al margine. L'algoritmo si divide in tre parti principali:

- Il primo *if*: controlla se la query sta interamente in un separabile bucket in modo da evitare di scendere ai livelli successivi. Per fare questo controllo si aumenta in margine di una quantità pari al raggio della query e, se il centro è sempre all'interno di una zona classificabile, allora tutta la regione della query è contenuta in una regione classificabile. In conseguenza a questo non è necessario scendere nei livelli successivi dell'index.
- Il secondo blocco *if*: controlla se la query ha raggio più piccolo del margine e successivamente controlla se la regione della query è completamente contenuta nel margine, se questo accade allora si può direttamente passare al livello successivo. Per fare questo si restringe il margine di una quantità pari al raggio e si prova a classificare il centro della query, se questo risulta non classificabile allora si può passare direttamente al livello successivo, altrimenti si deve accedere prima al bucket e successivamente passare al livello successivo perché la query interseca sia la zona classificabile che quella esclusa.
- Blocco *else*: questo blocco identifica una query con raggio maggiore del margine, potrebbe quindi intersecare tutte e tre le regioni insieme. Tramite la funzione G si ottengono gli indici dei bucket che si intersecano con la query e che quindi devono essere controllati per trovare il risultato.

```

for  $i = 1$  to  $h$ 
  if  $bps^{m_i, \rho+r}(q) \neq -$  then          (exclusively in the separable bucket)
    search in the bucket with the index  $bps^{m_i, \rho+r}(q)$ .
    exit                                (search terminates)
  end if
  if  $r \leq \rho$  then                  (the search radius up to  $\rho$ )
    if  $bps^{m_i, \rho-r}(q) \neq -$  then      (not exclusively in the exclusion zone)
      search in the bucket with the index  $bps^{m_i, \rho-r}(q)$ .
    end if
  else                                (the search radius greater than  $\rho$ )
    let  $\{i_1, \dots, i_n\} = G(bps^{m_i, r-\rho}(q))$ 
    search in the buckets with the indexes  $i_1, \dots, i_n$ .
  end if
end for
search in the global exclusion bucket.

```

La funzione G ritorna una serie di indici corrispondenti ai bucket che devono essere controllati, per fare questo controlla il centro della query con $r-\rho$ come margine, in questo modo se ottengo che il punto non è classificabile allora la regione interseca tutte e tre le regioni, altrimenti, se ottengo una classificazione la regione interseca quella ritornata e al più quella esclusa. Per questo motivo, quando ottengo che il punto non è classificabile devo ritornare la combinazione delle altre due regioni non considerando quella esclusa.

e.g. $bps_1^{1,\rho}(q)=1$, $bps_2^{1,\rho}(q)=-$, $bps_3^{1,\rho}(q)=-$

$$G(bps^{3,\rho}(q))=\{100, 101, 110, 111\}$$

Questo perché, intersecando sia la zona 1, sia la 0 che quella esclusa, significa che devo andare a controllare i bucket delle due regioni prima di passare al livello successivo.

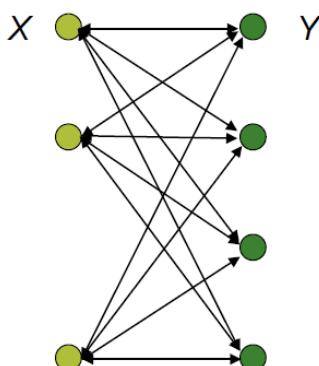
L'algoritmo generalizzato accede **nel peggiore dei casi** a $h+1$ bucket, l'algoritmo precedente accedeva sempre a $h+1$, in questo modo si possono quindi diminuire gli accessi e abbassare i costi di esecuzione.

Di seguito è riportato un riassunto delle caratteristiche che abbiamo discusso in precedenza.

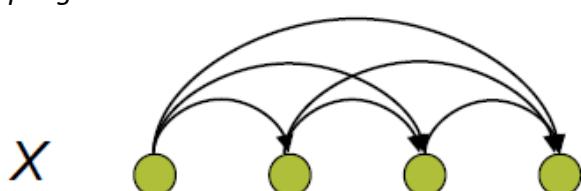
- supports disk storage
- insertion needs one bucket access
 - distance computations vary from m_1 up to $\sum_{i=1..h} m_i$
- $h+1$ bucket accesses at maximum
 - for all queries such that qualifying objects are within ρ
- exact match ($R(q, 0)$)
 - successful – one bucket access
 - unsuccessful – typically no bucket is accessed

Similarity Join Query

Questo tipo di query servono per ricavare coppie di oggetti di database diversi la cui distanza non supera una certa soglia impostata dall'utente. Queste coppie possono essere trovate facilmente calcolando un numero di distanze pari a $|N| \times |M|$, chiaramente questo approccio è altamente inefficiente.



Questo tipo di query può anche essere effettuata sullo stesso dataset, si troveranno quindi le coppie di oggetti più vicine fra di loro, si dovranno calcolare $|N| \times |N|$ distanze. Dato che però la distanza è simmetrica si può ottimizzare calcolando solo $\frac{N(N-1)}{2}$ distanze. Questo si chiama *nested loop algorithm*.



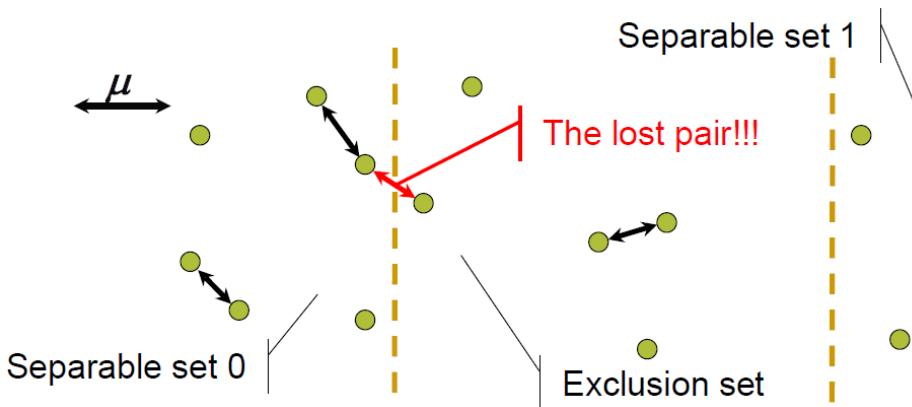
In un D-Index una *self join query* può essere svolta tramite il **range join algorithm**. In pratica per ogni oggetto del dataset si va a valutare una range query con raggio μ che rappresenta la distanza massima che ci può essere fra due oggetti.

Range Join Algorithm (RJ):

```
for each  $o$  in dataset  $X$  do
    range_query( $o, \mu$ )
end do
```

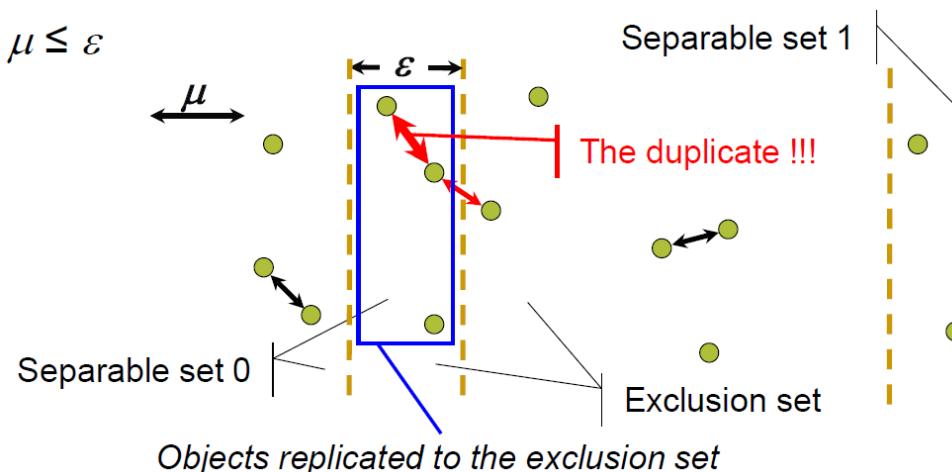
Extended D-index (eD-index)

Invece di calcolare le distanze fra un oggetto e tutti gli altri del dataset, si calcolano indipendentemente dentro a ogni bucket. La soluzione sarà l'unione di tutte le soluzioni trovate in ogni bucket. In figura è mostrato come si vada a perdere una coppia dal momento che non si controllano le distanze fra oggetti appartenenti a bucket differenti.



Se stiamo facendo una query con risultato approssimativo, possiamo anche ignorare la coppia che perdiamo. In questo momento stiamo cercando un risultato esatto. Dobbiamo quindi usare degli stratagemmi per non perdere nessuna coppia.

Possiamo utilizzare delle repliche delle zone nell'intorno dei bordi con la regione esclusa. La zona viene quindi salvata nel bucket del livello corrente e in qualche bucket del livello successivo, la larghezza della zona da replicare è larga ε . In questo modo si trovano le coppie perse, però troviamo anche delle coppie duplicate come nell'esempio successivo.



Per fare questo è necessario modificare l'algoritmo di inserimento, dobbiamo considerare il caso in cui un oggetto si trovi all'interno dell'area di duplicazione, in quel caso l'oggetto deve essere inserito anche al livello successivo, altrimenti deve essere inserito solo nel livello corrente.

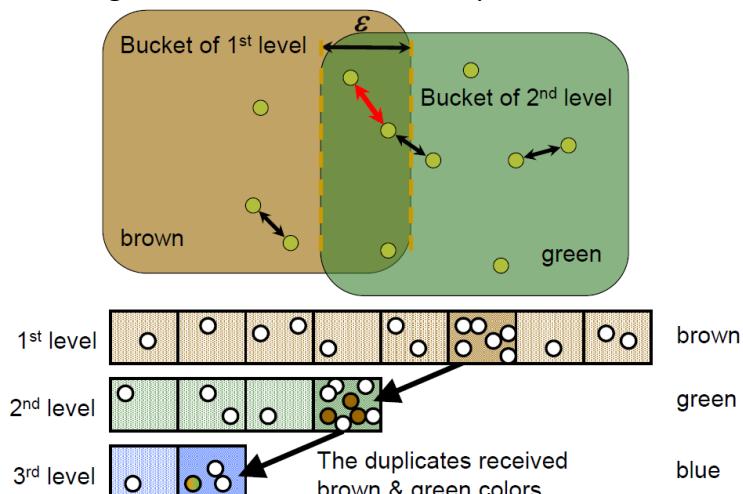
- $eDindex^{\rho, \varepsilon}(X, m_1, m_2, \dots, m_h)$
- Algorithm – insert the object o_N :


```

        for i=1 to h do
          if  $bps^{m_i, \rho}(o_N) \neq '-'$  then
             $o_N \rightarrow$  bucket with the index  $bps^{m_i, \rho}(o_N)$ .
            if  $bps^{m_i, \rho+\varepsilon}(o_N) \neq '-'$  then    (not in the overloading area)
              exit
            end if
          end if
        end do
      
```

$o_N \rightarrow$ global exclusion bucket.

Vediamo ora come si possono evitare i duplicati fra le coppie. Ad ogni bucket si assegna un colore e agli oggetti presenti nella zona duplicata assegniamo tutti e due i colori. Nell'esempio successivo si estraggono prima le coppie dal bucket del primo livello e si contrassegnano come marroni. Passando al livello successivo ad un certo punto analizzeremo la coppia con la freccia rossa, questa coppia non sarà contata una seconda volta dal momento che entrambi i componenti sono stati contrassegnati con il colore del livello precedente.



In altre parole, quando un oggetto viene dal livello 1, nel secondo livello viene salvato con il colore relativo al primo livello. Grazie a questo, se al livello 2 si analizzano le coppie, se entrambi sono contrassegnati con il marrone allora **devono** essere già state conteggiate nell'analisi del livello precedente. Quando scendiamo al terzo livello un oggetto che proviene dal secondo e già era salvato col marrone allora avrà i due colori.

Abbiamo visto che la *similarity self-join query* $SJ(\mu)$ è eseguita indipendentemente in ogni bucket per ogni livello. Possiamo ottimizzare questo processo in quanto le distanze da calcolare in ogni bucket potrebbero essere tante. Applichiamo un algoritmo con una finestra a scorrimento per ogni bucket, in pratica si sceglie un pivot per ogni bucket e si ordinano tutti gli oggetti di un bucket in base alla distanza col pivot. Si fa scorrere una finestra larga μ e si controllano le distanze solo fra gli oggetti contenuti nella finestra, si può fare questo perché si sfrutta la disuguaglianza triangolare. Due oggetti fuori dalla finestra non rispettano sicuramente la disuguaglianza triangolare e di conseguenza la loro distanza sarà maggiore di μ .

$$d(x,y) \geq d(x,p) - d(y,p) > \mu$$

Chiaramente questo algoritmo sfrutta il pivot filtering e lo schema dei colori appena visto per evitare i duplicati.

Parliamo brevemente delle limitazioni dell' **eD-index**. La distanza fra due oggetti devi essere più piccola della larghezza della zona di duplicazione, altrimenti non si risolve il problema delle coppie perse, quindi $\mu \leq \epsilon$. In più, la larghezza della zona esclusa e la larghezza della zona di duplicazione sono correlate. $\epsilon \leq 2\rho$ perché altrimenti la zona di duplicazione sarebbe talmente larga da sforare nel bucket dell'altra zona.

Approximate Similarity Search

In molte applicazioni reali user-oriented la ricerca approssimata può andar bene perché non necessitiamo di risultati esatti, quelli approssimativi vanno più che bene. La ricerca per similarità approssimativa supera i problemi della ricerca per similarità esatta utilizzando i metodi di accesso tradizionali.

La ricerca per similarità approssimativa processa le query più velocemente a costo di un po' di imprecisione nel result set. I miglioramenti in termini di performance sono notevoli, fino a due ordini di grandezza! (100 volte meglio).

Strategie di approssimazione:

- *Relax pruning conditions*: le regioni di dati che si sovrappongono con la regione della query possono essere scartate a seconda di strategie specifiche
- *Terminazione anticipata degli algoritmi di ricerca*: gli algoritmi di ricerca potrebbero fermarsi prima che tutte le regioni vengano accedute.

Vediamo adesso le seguenti tecniche per rendere una ricerca approssimativa:

- relative error approximation (pruning condition): Range and k-NN search queries
- good fraction approximation (stop condition): K-NN search queries
- small chance improvement approximation (stop c.): K-NN search queries
- proximity-based approximation (pruning condition): Range and k-NN search queries
- PAC nearest neighbor searching (pruning + stop c.): 1-NN search queries

Relative error approximation (pruning condition)

Dato o^A come il NN alla query q se vale

$$\frac{d(o^A, q)}{d(o^N, q)} \leq 1 + \epsilon$$

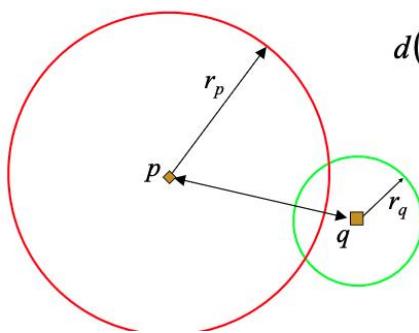
(distanza tra l'approssimazione e la query diviso la distanza tra il NN e la query)

Allora o^A è il $(1+\epsilon)$ approximate NN di q .

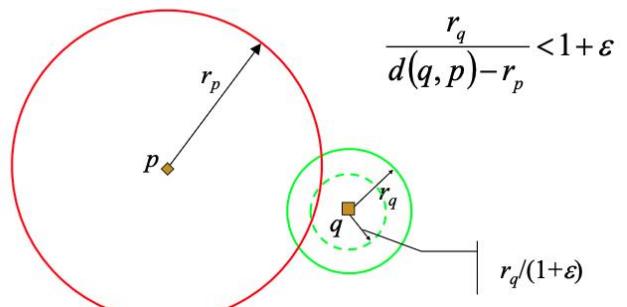
Possiamo anche generalizzare il k-esimo NN con

$$\frac{d(o_k^A, q)}{d(o_k^N, q)} \leq 1 + \epsilon$$

Exact pruning strategy:



Approximate pruning strategy:



Usando l'approssimazione è come se si usasse un raggio più piccolo, sicuramente perderemo degli oggetti ma siamo certi che l'errore sia inferiore a ε .

Good Fraction Approximation (stop condition)

Il k-NN determina il risultato finale riducendo le distanze del result set corrente. Quando questo appartiene ad una specifica frazione degli oggetti vicino alla query, l'algoritmo di approssimazione si ferma (ad es. si ferma quando il result set corrente appartiene al 10% degli oggetti più vicini alla query).

Andiamo a fruttare distance distribution, già vista nei capitoli precedenti.

The *distance distribution* with respect to the object p (*viewpoint*) is

$$F_{D_p}(x) = \Pr\{D_p \leq x\} = \Pr\{d(p, o) \leq x\}$$

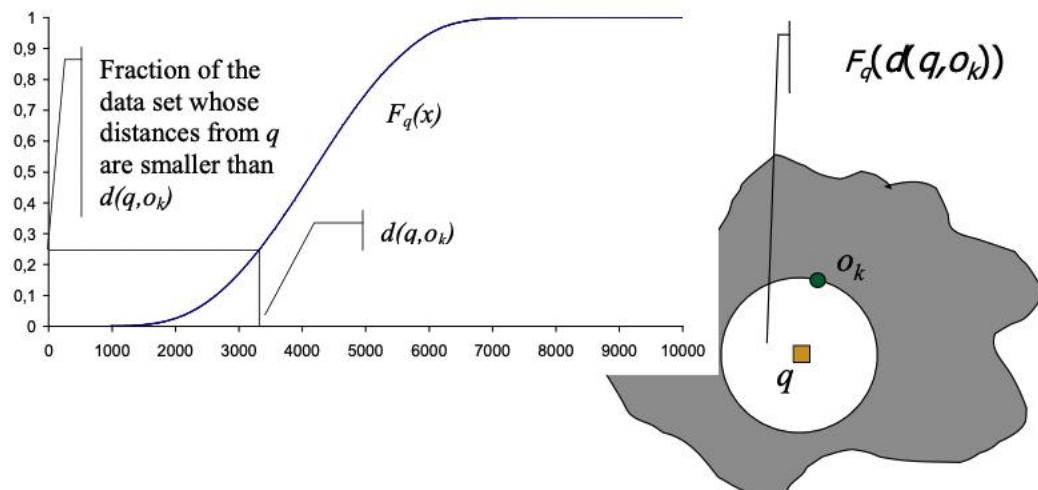
where D_p is a random variable corresponding to the distance $d(p, o)$ and o is a random object of the metric space.

The *distance density* from the object p can be obtained as the derivative of the distribution.

The *overall distance distribution* (informally) is the probability of distances among objects

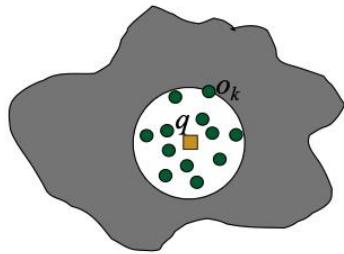
$$F(x) = \Pr\{d(o_1, o_2) \leq x\}$$

where o_1 and o_2 are random objects of the metric space.



Se seleziono una distanza sull'asse x, sull'asse y avrò la frazione di dati (p) che distano meno di $d(q, o_k)$ dalla query.

Quando $F_q(d(o_k, q)) < \rho$ tutti gli oggetti del result set corrente appartengono alla frazione ρ del dataset.

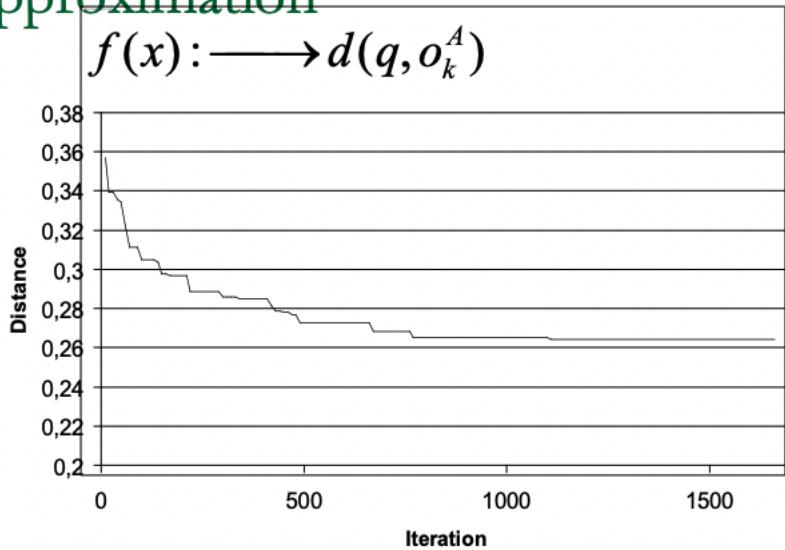


$F_q(x)$ è difficile da gestire perché dobbiamo calcolarlo per tutte le query possibili. E' stato provato che in pratica possiamo usare la overall distance distribution $F(x)$ dato che statisticamente hanno lo stesso comportamento. $F(x)$ può essere facilmente stimata come una funzione discreta e può essere facilmente mantenuta in memoria.

Small Chance Improvement Approximation (stopping condition)

L'idea alla base è se andando avanti con l'algoritmo mi porta ad avere solo un piccolo miglioramento allora mi fermo prima.

Approximation



Sull'asse y è riportata la distanza $d(q, o_k)$.

Quello da capire è quando non c'è più un miglioramento consistente e quindi possiamo fermarci. Andiamo a tal proposito a costruire una curva di regressione $\phi(x)$, calcolata usando i minimi quadrati, che approssima $f(x)$. Tramite la derivata della curva possiamo capire quando fermarsi.

The regression curve has the following form

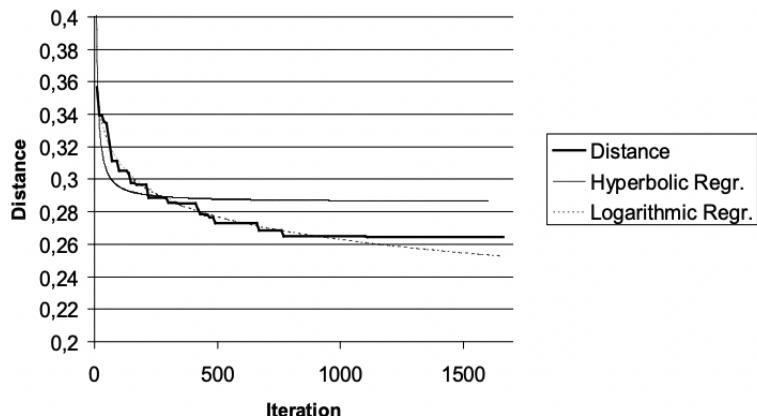
$$\phi(x) = c_1 \varphi_1(x) + c_2$$

where c_1 and c_2 are such that

$$\sum_{i=0}^j (c_1 \varphi_1(i) + c_2 - f(i))^2$$

is minimum

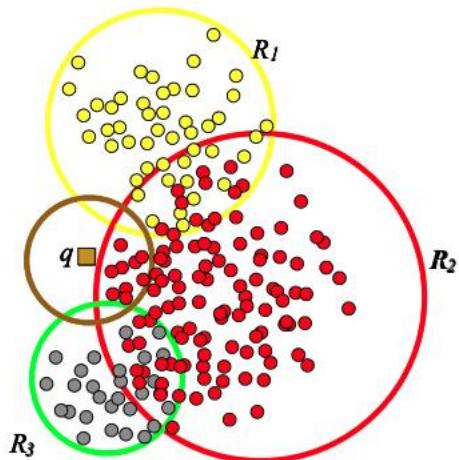
Tests were executed with $\varphi_1(x) = \ln(x)$ and $\varphi_1(x) = 1/x$



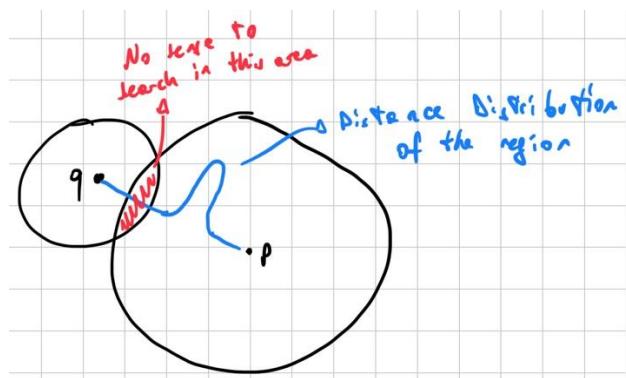
Proximity-based approximation (pruning condition)

L'idea è che le regioni che hanno una probabilità di contenere oggetti sotto una certa soglia possono non essere considerate (pruning) anche se intersecano la query.

Si ottengono miglioramenti di due ordini di grandezza sia per le range queries che per la NN query. Dall'esempio in figura si nota come non avrebbe senso andare a cercare nelle regioni gialle e verdi che intersecano la query, mentre è necessario cercare nell'intersezione con la regione rossa.



Non andiamo a visitare le regioni dove c'è poca probabilità di trovare oggetti rilevanti per la query (vedi Proximity of Ball Regions nei capitoli precedenti).



PAC Nearest Neighbor Searching (pruning + stop c)

Può essere usata solo per le 1-NN.

Andiamo ad usare il relaxed branching condition visto prima per trovare un $(1+\varepsilon)$ approximate-NN, alla quale aggiungiamo una stopping condition: si ferma prematuramente quando la probabilità che abbiamo di trovare $(1+\varepsilon)$ approximate-NN è sotto una threshold δ .

La PAC può essere usata solo per le **1-NN query**.

Supponiamo quindi che il NN trovato fino ad ora sia o^A , e definiamo ε_{act} come l'errore attuale sulla distanza di o^A :

$$\varepsilon_{act} = \frac{d(o^A, q)}{d(o^N, q)} - 1$$

$$\Pr\{\varepsilon_{act} \geq \varepsilon\} \leq \delta$$

Allora l'algoritmo di fermerà quando

La probabilità appena definita è ottenuta calcolando **la distribuzione della distanza del NN**, che definiamo rispetto a X (di cardinalità n) e rispetto alla query q come:

$$G_q(x) = \Pr\{\exists o \in X : d(q, o) \leq x\} = 1 - (1 - F_q(x))^n$$

Dove $(1 - F_q(x))^n$ è la probabilità che tutti gli oggetti abbiano una distanza più grande di x, dove la n sta a significare che vale per tutti gli oggetti di X.

$: 1 - (1 - F_q(x))^n$ significa la probabilità che tra n oggetti nel db ce n'è almeno uno che ha la distanza minore di x

L'idea è, come detto prima di fermarsi quando $\Pr\{\varepsilon_{act} \geq \varepsilon\} \leq \delta$ che può essere scritto come

$$\begin{aligned} \Pr\{\varepsilon_{act} \geq \varepsilon\} &= \Pr\{\exists o \in X : d(q, o^A) / d(q, o) - 1 \geq \varepsilon\} = \\ &= \Pr\{\exists o \in X : d(q, o) \leq d(q, o^A) / (1 + \varepsilon)\} = G_q(d(q, o^A) / (1 + \varepsilon)) \end{aligned}$$

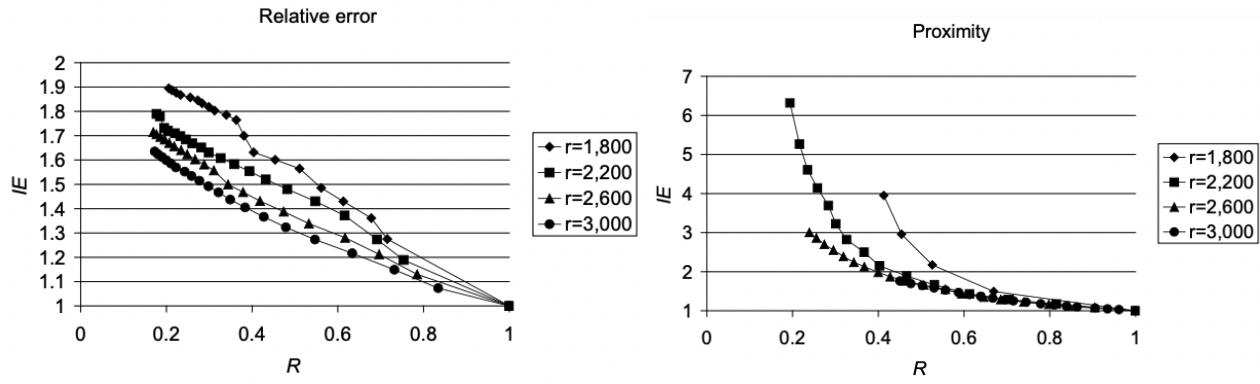
Quindi il nostro algoritmo si **fermerà quando**

$$G_q(d(q, o^A) / (1 + \varepsilon)) \leq \delta$$

Andiamo a confrontare i risultati usando delle **range query**, prima con l'*errore relativo* e poi con la *proximity*.

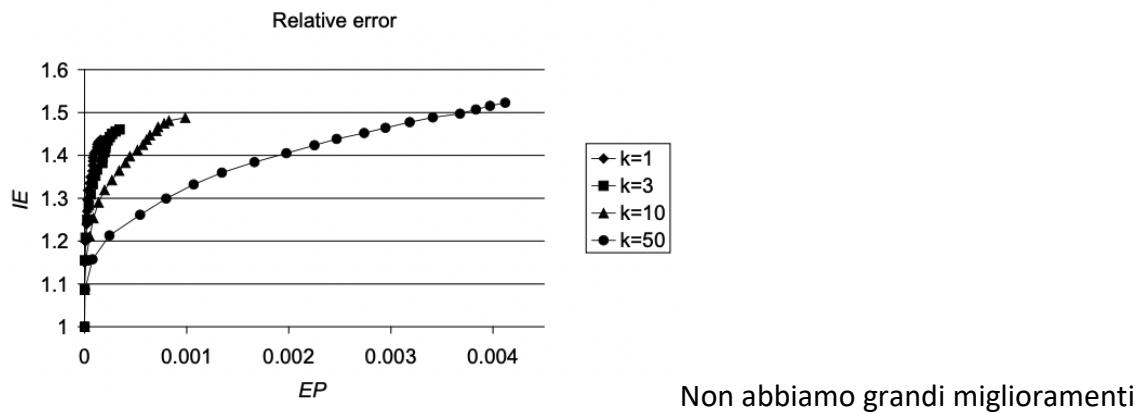
Sugli assi abbiamo:

- *Improvement of efficiency (IE)* che è il costo dell'algoritmo approssimato diviso il costo di quello reale.
- *Recall (R)*



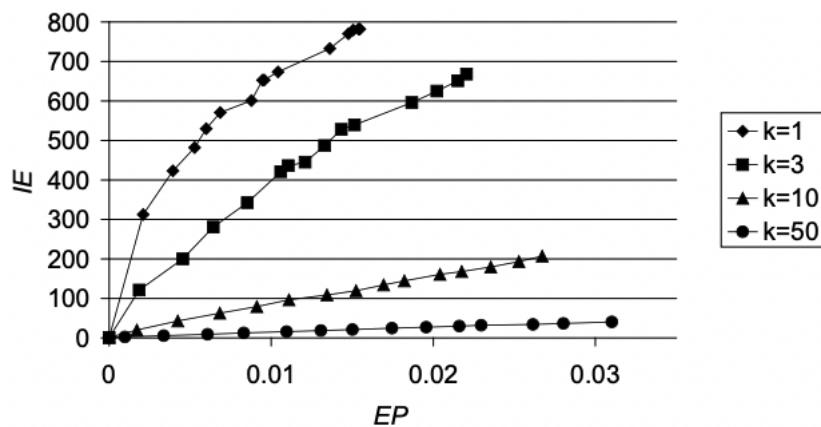
Nel primo grafico vediamo un miglioramento dell'IE fino a circa due volte mentre nel secondo fino a 6 ma abbiamo una recall bassa (0.2).

Valutiamo ora le **NN queries**, sull'asse y abbiamo sempre IE invece sull'asse x abbiamo *Error per Position (EP)*



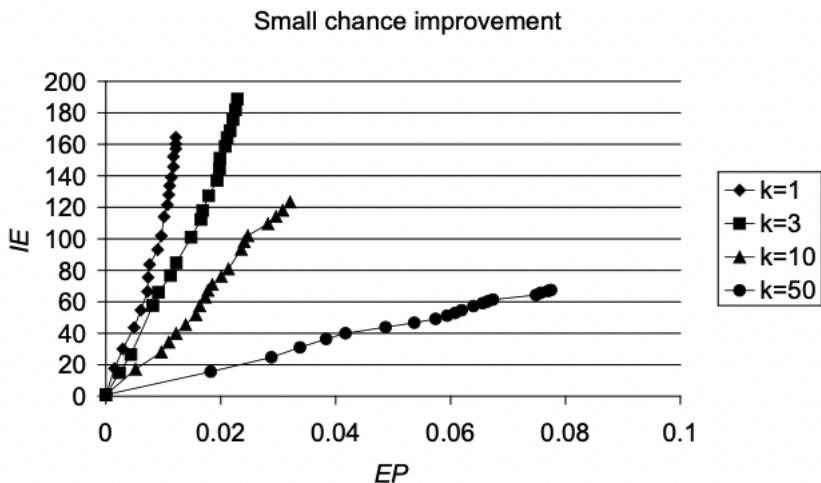
Non abbiamo grandi miglioramenti

Vediamo adesso la tecnica basata su **good fraction**, quella che sfrutta le distribution distances
Good fraction

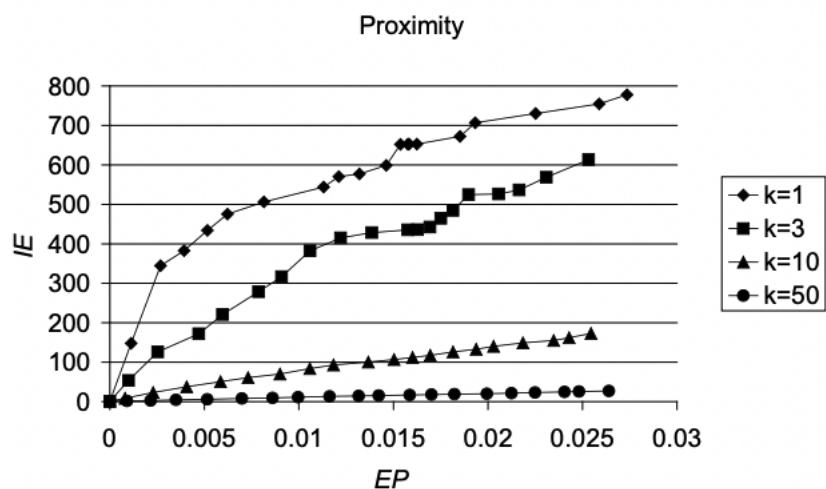


Abbiamo miglioramenti fino a ca 800 volte rispetto a $k=50$, con EP ca 0.015

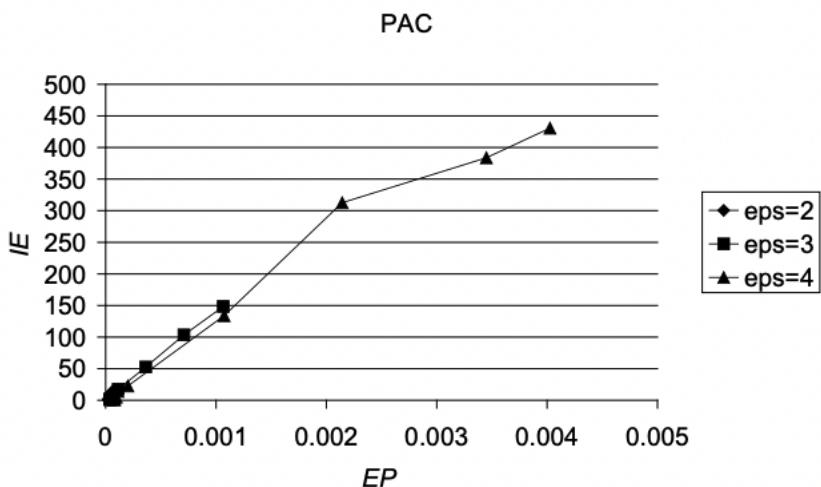
Usando **small chance improvement** abbiamo un miglioramento significativo ma con un EP=0.02 cioè un po' maggiore rispetto al caso precedente



Usando **proximity**abbiamo miglioramenti significativi con errori accettabili



Con **PAC**abbiamo solo una linea perché si può fare solo 1-NN, anche qua abbiamo un incremento notevole.



Considerazioni finali

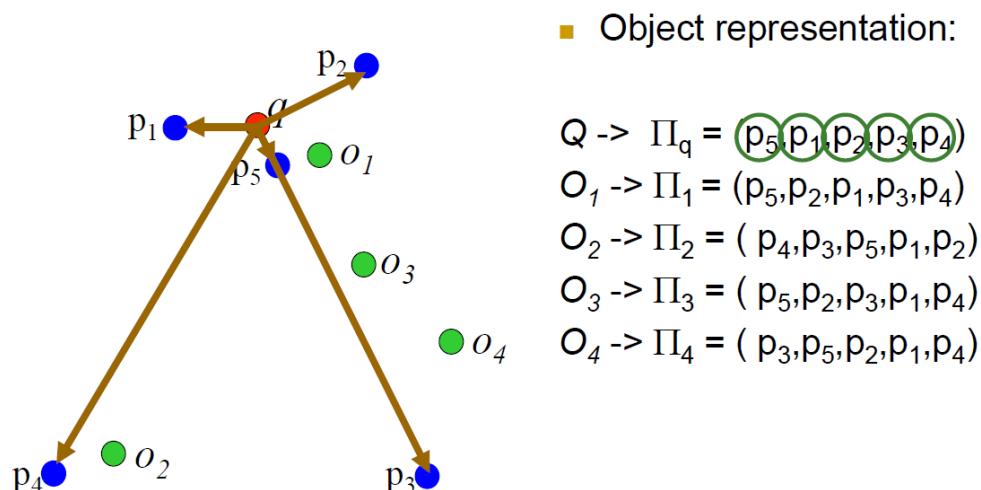
Queste tecniche per la ricerca approssimata possono essere applicate al generic metric space, ricordiamo che il vector space è un caso speciale del metric.

Un'alta accuratezza dei risultati approssimati è generalmente ottenuta con un alto incremento dell'improvement of efficiency, dai risultati ottenuti possiamo affermare che:

- Le migliori performance sono ottenute con **good fraction**
- Il **proximity** è un po' peggio del good fraction ma può essere usato per le range query e per le k-NN.

Permutation based indexes

Vediamo come possiamo rappresentare un oggetto dato un set di pivot P. Di conseguenza, se due oggetti o_1 e o_2 sono simili allora lo saranno anche le loro rappresentazioni basate sulle distanze dai pivot del set P. Dopo aver definito questo concetto possiamo rappresentare l'oggetto o come permutazione di pivot del set P in base alla distanza da essi e possiamo misurare la similarità fra due oggetti semplicemente misurando la similarità fra le due permutazioni. Nella seguente immagine possiamo notare come vengono rappresentati tutti gli oggetti del database.



Per calcolare la distanza fra due permutazioni si usa la **Spearman Footrule Distance**.

$$SFD(\Pi_1, \Pi_2) = \sum_{1 \leq i \leq k} |\Pi_1^{-1}(i) - \Pi_2^{-1}(i)|$$

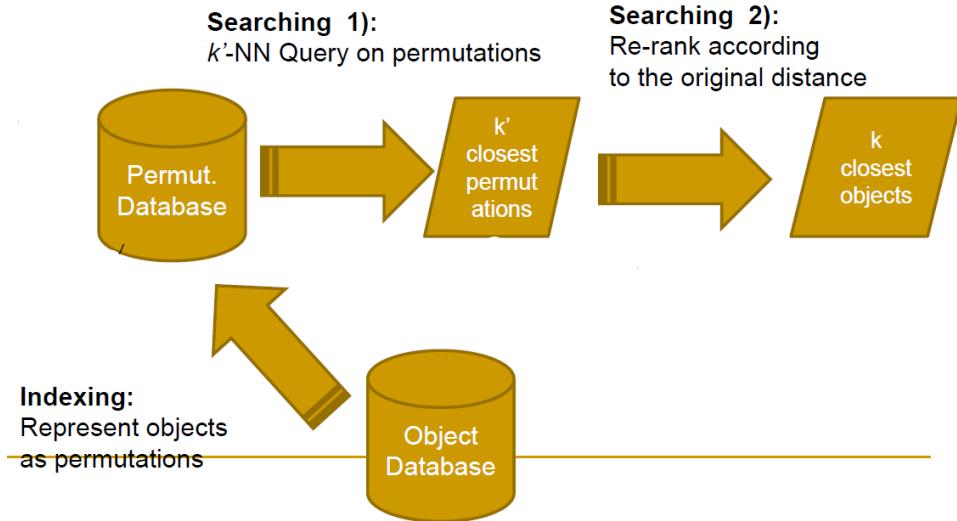
Dove π^{-1} è una funzione che restituisce la posizione dell'oggetto i all'interno della permutazione, sommando tutte le differenze otteniamo la distanza. Chiaramente più la distanza è bassa e più le due permutazioni sono simili.

Permutation Spearman Rho (PSR)

Si utilizza sempre l'idea di rappresentare gli oggetti tramite le permutazioni di pivot solo che la comparazione di due permutazioni viene fatta tramite la seguente formula.

$$S_\rho(\Pi_1, \Pi_2) = \sum_{1 \leq i \leq k} (\Pi_1^{-1}(i) - \Pi_2^{-1}(i))^2$$

Per eseguire una ricerca di tipo k-NN dobbiamo recuperare le k' permutazioni, dove $k' > k$, tramite la formula appena vista e successivamente recuperare i k oggetti più vicini alla query utilizzando però la metrica originale come ad esempio la distanza euclidea. Vediamo con un'immagine come funziona questo ragionamento.



Per prima cosa si trasforma il database originale tramite le permutazioni, successivamente si recuperano le k' permutazioni e alla fine si trovano i k oggetti.

Questo approccio non risulta scalabile dal momento che dobbiamo scansionare l'intero dataset per poter generare le permutazioni, si genera quindi una permutazione per ogni oggetto e successivamente dobbiamo scannerizzare nuovamente il database compatto per trovare le k' permutazioni più vicine alla query. Inoltre, più oggetti ho nel database e più pivot devo prendere per poter discriminare tutte le permutazioni possibili, allungando quindi le rappresentazioni per ogni oggetto. Tuttavia, con pochi oggetti e relativamente poche features possiamo ottenere un database più compatto e più facile da analizzare.

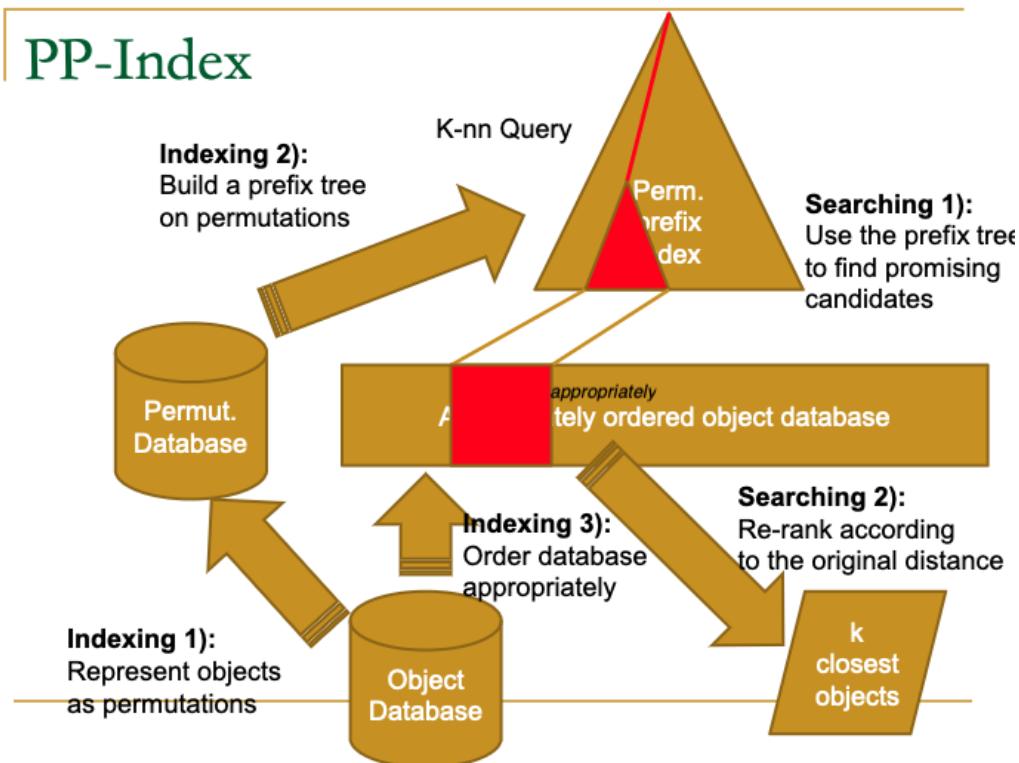
PP-Index

Il PP-Index è una permutazione basata su una data structure che rappresenta ogni oggetto con una permutazione più corta di lunghezza l , dove $l < |P|$.

La **data structure** è un prefix tree mantenuto in memoria principale e viene fatto un index sui prefissi delle permutazioni. Gli oggetti vengono salvati su disco in modo ordinato, salviamo le informazioni necessarie per calcolare le distanze reali tra gli oggetti.

Una **K-NN query** usando il PP-index viene fatta come segue:

- Il prefix tree viene usato per identificare velocemente il set di oggetti candidati
- La funzione di distanza esatta viene usata per determinare i k-NN del risultato finale dal set di candidati
- I candidati sono presi dal data storage con pochi accessi sequenziali al disco



Building the PP-index

La **creazione** dell'index è composta da tre fasi:

1. Calcolare i prefissi delle permutazioni di lunghezza l .

Supponiamo di avere r pivot, l deve essere più piccolo di r . Quello che si fa è costruire le permutazioni e poi troncarle per avere solo i primi l pivot più vicini.

For instance

- Suppose $r=6$ and $l=3$
- $O \rightarrow \Pi_o = (1,6,4,2,5,3)$
- Prefix of length l will be $= (1,6,4)$
- Permutations having the same prefix collide

Dei possibili numeri reali per r e l possono essere ad esempio $r=1000$ e $l=10$.

2. Organizzare le permutazioni nel prefix tree
 $r=6$ and $l=3$

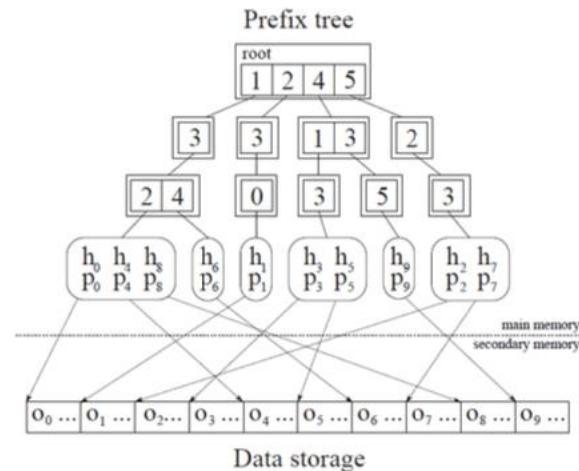
$$\Pi_0 = (1, 3, 2) \quad \Pi_1 = (2, 3, 0)$$

$$\Pi_2 = (5, 2, 3) \quad \Pi_3 = (4, 1, 3)$$

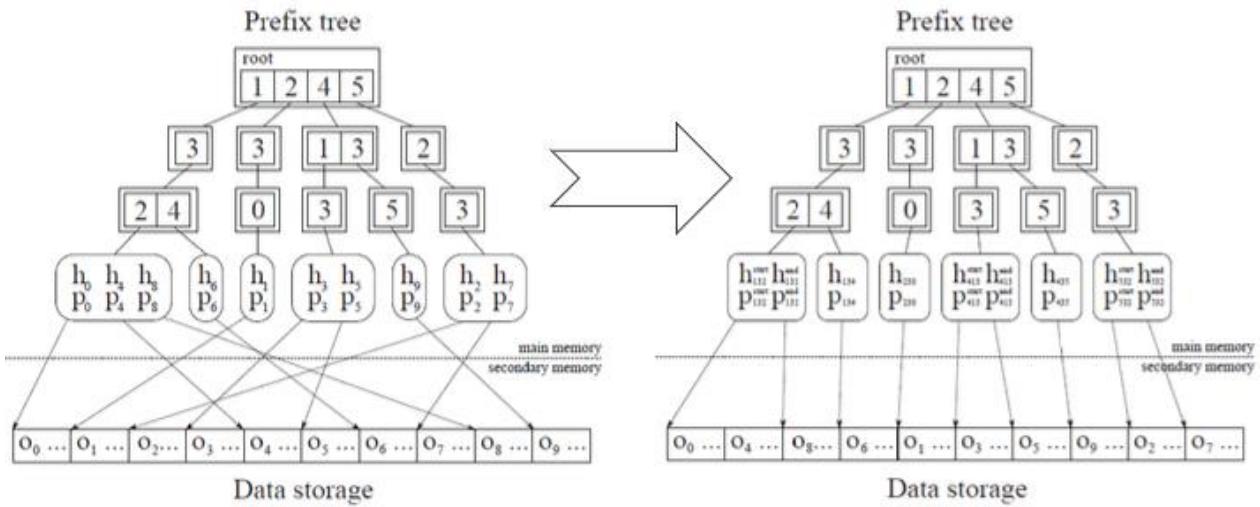
$$\Pi_4 = (1, 3, 2) \quad \Pi_5 = (4, 1, 3)$$

$$\Pi_6 = (1, 3, 4) \quad \Pi_7 = (5, 2, 3)$$

$$\Pi_8 = (1, 3, 2) \quad \Pi_9 = (4, 3, 5)$$



3. Ottimizzare lo storage degli oggetti



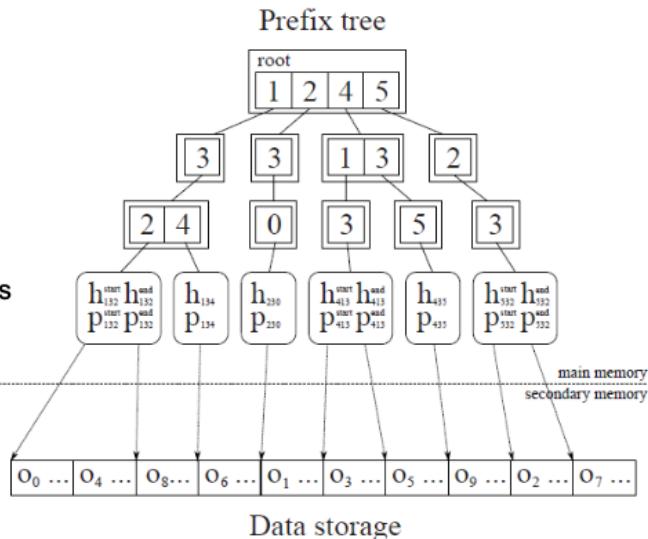
Quando facciamo una query andiamo a scorrere l'albero per trovare il set di candidati, è bene salvare sequenzialmente i set di candidati in modo da fare un accesso sequenziale al disco.

Searching PP index

Data un oggetto query q vogliamo trovare i k -NN di q , abbiamo due fasi:

1. Trovare i candidati: trovare un set di almeno z candidati, con $z > k$

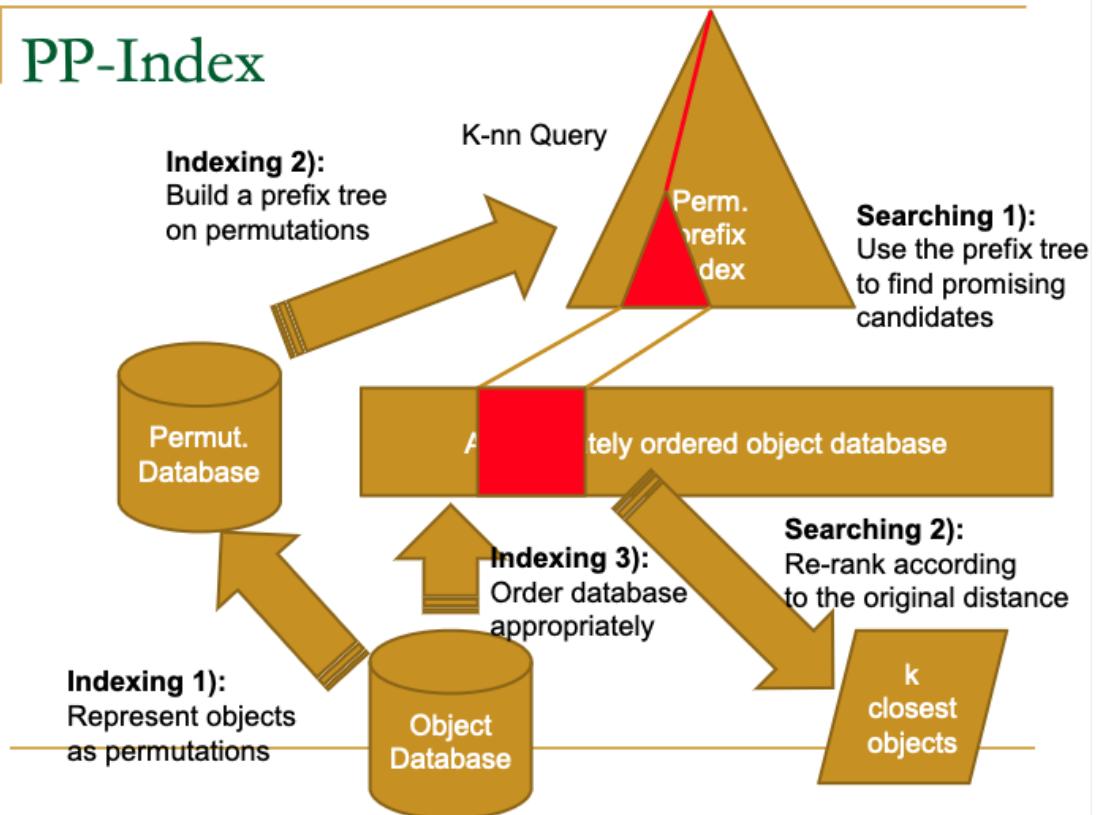
- Given the prefix representing the query
 - search the smallest subtree pointing to an interval containing at least z candidates
- Suppose $\Pi_q = (1, 3, 3)$, and $z=4$.
 - The smallest subtree has root $(1, 3)$



2. Produciamo un risultato approssimativo: scegliamo i k -NN di tra i candidati.

- Scorriamo il set di candidati
- Ritorniamo i k oggetti più vicini alla query dall'intervallo dei candidati
- Dato che lo storage è ottimizzato l'intervallo dei candidati del sottoalbero corrisponde a un intervallo consecutivo su disco, facciamo quindi uno scan sequenziale che è molto efficiente

PP-Index



Improving search effectiveness

L'efficacia può essere migliorata usando:

- **Indici multipli:** possiamo usare n differenti indici usando n differenti set di pivot, ovviamente lo storage deve contenere n diversi ordini di oggetti. Quando facciamo una query i risultati del k-NN dei vari indici sono mescolati insieme in un unico risultato.
- **Query multiple:** data la permutazione di una query Π_q andiamo a fare dei piccoli cambiamenti per esplorare il suo vicino, quello che si fa è invertire coppie di elementi, partendo dai pivot più vicini alla query. Ad esempio se $\Pi_q=(1234)$ facciamo le query (1243), (1324), ecc. Tutte le query perturbate vengono usate per cercare nello stesso indice e i vari risultati vengono mescolati.

MI-File

Il nome sta per **Metric inverted file**, come suggerito da nome con questo approccio si rappresentano le permutazioni attraverso un inverted file. Ogni entry dell'inverted file rappresenta un pivot e all'interno della posting list di ogni pivot sono presenti delle coppie formate da l'id dell'oggetto e dalla posizione del pivot pi all'interno della permutazione che rappresenta quell'oggetto. Per effettuare una ricerca di tipo k-NN si calcola la *Footrule Distance* rilassate incrementalmente.

Vediamo come costruire un index. Dati un dataset di oggetti con le relative permutazioni, possiamo rappresentare l'inverted file tramite una tabella. L'ultima riga indica il pivot mentre la prima colonna indica l'oggetto. All'interno di ogni cella è indicata la posizione del pivot all'interno della permutazione dell'oggetto. Per esempio, il pivot numero uno è presente in posizione 3 all'interno della permutazione dell'oggetto o₁, troveremo quindi 3 nella cella corrispondente.

O1 =	(5,2,1,3,4)	O1	3	2	4	5	1
O2 =	(4,3,5,1,2)	O2	4	5	2	1	3
O3 =	(5,2,3,1,4)	O3	4	2	3	5	1
O4 =	(3,5,2,1,4)	O4	4	3	1	5	2
			1	2	3	4	5

Abbiamo però detto che ogni entry dell'inverted index è composta da un pivot e in ogni posting list sono presenti delle coppie di valori, manipoliamo quindi la tabella precedente in modo da ottenere quello che vogliamo.

1	3,O1	4,O2	4,O3	4,O4
2	2,O1	2,O3	3,O4	5,O2
3	1,O4	2,O2	3,O3	4,O1
4	1,O2	5,O1	5,O3	5,O4
5	1,O1	1,O3	2,O4	3,O2

La prima è una rappresentazione diretta della permutazione, la seconda è una rappresentazione invertita in quanto la entry non è più l'oggetto ma il pivot.

Vediamo adesso come poter eseguire delle query tramite questo inverted idex. Tramite l'inverted index, ad esempio, è molto facile recuperare tutti i documenti che contengono una certa parola in quanto basta scorrere la posting list di quella parola per leggere tutti i documenti in cui è contenuta. Se volessimo trovare tutti i documenti che contengono due specifiche parole si potrebbe recuperare le due posting list e poi farne l'intersezione. Vediamo come possiamo usare questo concetto per comparare una query con tutti i documenti dell'inverted index.

q

1	2	1	3,O1	4,O2	4,O3	4,O4
2	3	2	2,O1	2,O3	3,O4	5,O2
3	4	3	1,O4	2,O2	3,O3	4,O1
4	5	4	1,O2	5,O1	5,O3	5,O4
5	1	5	1,O1	1,O3	2,O4	3,O2

Abbiamo detto che due elementi si confrontano usando la spearman footrule distance che consiste nel sommare tutte le differenze fra le posizioni fra le due permutazioni.

Si comincia con il primo pivot e si scannerizza la relativa posting list, per ogni oggetto si fa la differenza fra la posizione riportata nella posting list e quella della query, chiaramente in valore assoluto. Dopo aver scannerizzato tutte le posting list abbiamo trovato tutti i contributi e basta sommare per ogni oggetto per ottenere il valore della Spaerman Footrule Distance.

q

1	2	1	3,O1	4,O2	4,O3	4,O4
2	3	2	2,O1	2,O3	3,O4	5,O2
3	4	3	1,O4	2,O2	3,O3	4,O1
4	5	4	1,O2	5,O1	5,O3	5,O4
5	1	5	1,O1	1,O3	2,O4	3,O2

O1	O2	O3	O4
1+	2+	2+	2+
1+	2+	1+	0+
0+	2+	1+	3+
0+	4+	0+	0+
0=	2=	0=	1=

SFD: 2 12 4 6

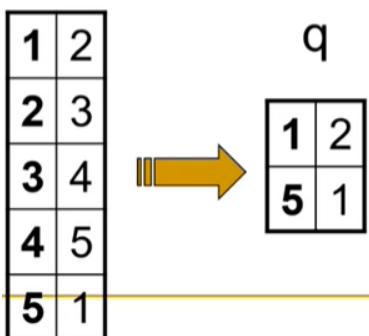
In questo semplice modo si è ottenuto l'oggetto più simile alla query, quindi quello più vicino, e anche quello più lontano.

In questo esempio abbiamo comparato la query con **ogni** oggetto del database, questo risulta essere troppo pesante a livello computazionale portando la complessità pari al numero di oggetti nel dataset moltiplicato per il numero di pivot che abbiamo usato. Apparentemente questo metodo risulta essere più complesso di un sequential scan. Abbiamo visto come questo approccio possa essere conveniente quando vogliamo trovare i documenti che contengono una parola perché semplicemente le posting list delle parole non contengono tutti i documenti del dataset, la matrice che verrà prodotta sarà quindi **sparsa**, questo ci porta a un vantaggio in termini di complessità. Nel nostro caso ogni posting list contiene tutti gli oggetti del database dal momento che ogni oggetto è rappresentato con una permutazione di tutti i pivot e deve essere scorsa **per intero**, inoltre quando eseguiamo una query usiamo tutto il vocabolario (tutti i pivot) e quindi dobbiamo accedere a tutte le entry dell'inverted index.

Vediamo adesso come possiamo risolvere questi problemi.

Prima di tutto di può decidere di rappresentare una query tramite i k_s pivot più vicini, dove k_s è un numero molto minore rispetto al numero totale di pivot.

q



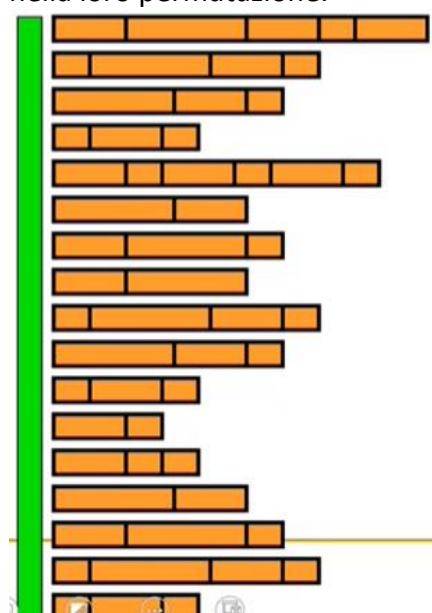
In questo esempio usiamo il pivot nella prima posizione e quello che si trova in seconda posizione. Questo trucco di porta ad accedere **solo** alle posting list dei k_s pivot più vicini, non dobbiamo più accedere tutte le posting list presenti nell'inverted index.

Applichiamo lo stesso ragionamento anche alla rappresentazione degli oggetti, essi saranno rappresentati dai k_x pivot più vicini. Il risultato è che le posting list diventano più corte dal momento che ogni oggetto è presente solo in k_x posting list (prima un oggetto era presente in ogni lista perché la permutazione conteneva tutti i pivot, adesso che è rappresentato solo con un sottoinsieme di pivot l'oggetto sarà inserito solo nelle posting list di pivot presenti nel sottoinsieme).

Con queste due migliorie abbiamo permesso di accedere a meno posting list quando si esegue una query e abbiamo ridotto la lunghezza delle posting list da accedere. Possiamo fare un'ulteriore miglioria.

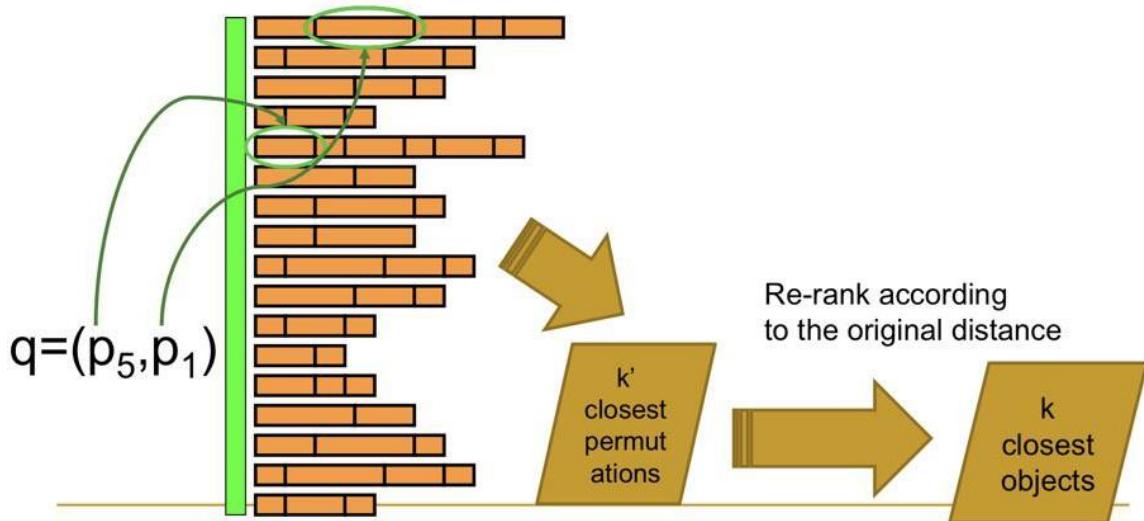
Ricordiamoci che noi stiamo cercando gli oggetti che più sono vicini alla query, quindi gli oggetti che hanno gli stessi pivot nella stessa posizione di quelli della query.

Quello che possiamo fare è ordinare le posting list secondo la posizione che occupa un determinato pivot, in questo modo possiamo formare dei blocchi. Ad esempio, nel primo blocco della prima posting list sono presenti gli oggetti che hanno il pivot numero 1 in prima posizione nella loro permutazione.



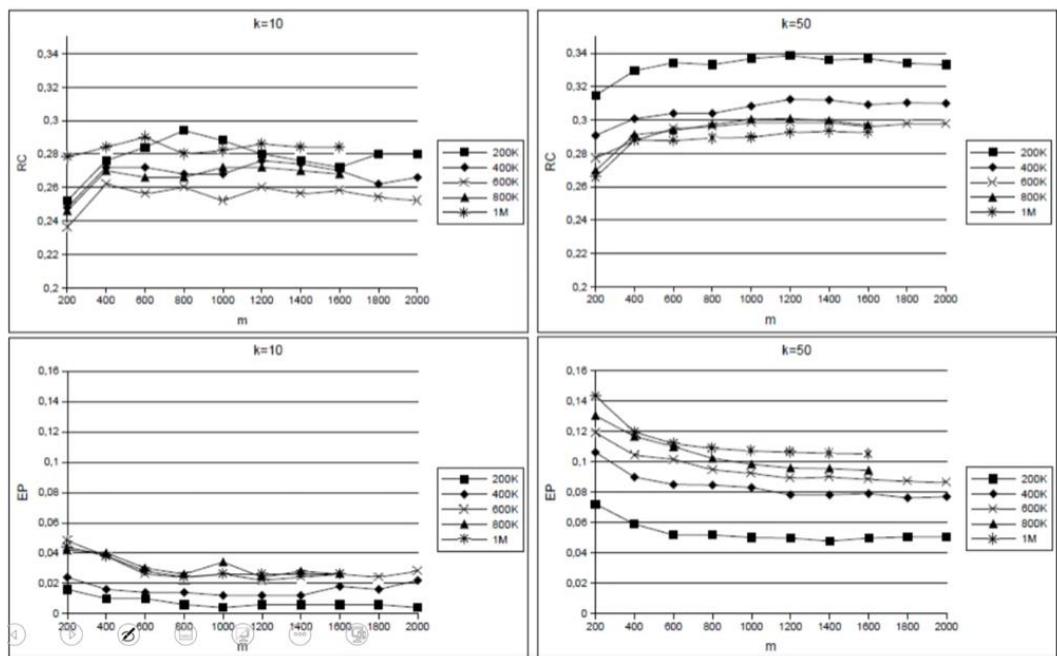
Facciamo un esempio per capire meglio, vogliamo applicare la query $q = (p_5, p_1)$ che presenta il pivot 5 in prima posizione e il pivot 1 in seconda posizione. Quello che facciamo è quello di accedere alla posting list relativa al pivot 5 ma, invece di scorrerla tutta, si **accede solo al primo blocco** perché li saranno gli oggetti che hanno quel pivot in quella posizione. Si applica lo stesso ragionamento al pivot 1. Se volessimo fare una ricerca esatta basterebbe fare l'intersezione fra quei due blocchi, noi però siamo cercando gli oggetti più simili alla query. Possiamo quindi allargare i limiti e cercare gli oggetti che si trovano nelle vicinanze del blocco scelto in modo da valutare più oggetti. Il vantaggio è quindi che non dobbiamo più scorrere tutta la posting list ma dobbiamo solo accedere ad una piccola parte.

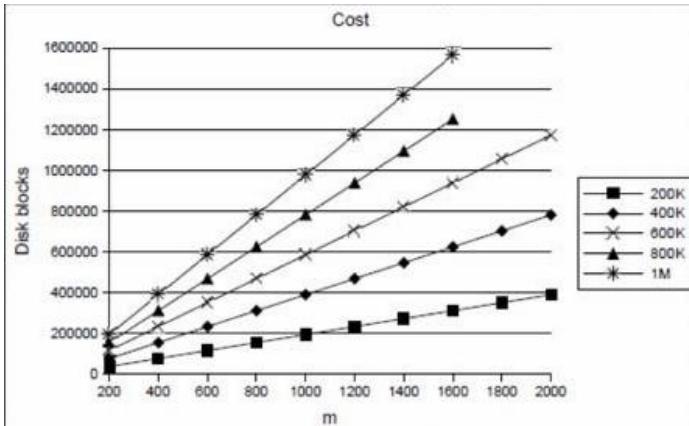
Tramite questo meccanismo otteniamo k' candidati, dove k' è un numero molto maggiore dei k che volevamo inizialmente. I k' candidati vengono quindi riordinati tramite la metrica originale usando le permutazioni e poi ne vengono scelti solo i primi k . Tipicamente k' è ottenuto moltiplicando k per un fattore di amplificazione.



Di seguito vediamo dei grafici che indicano le performance.

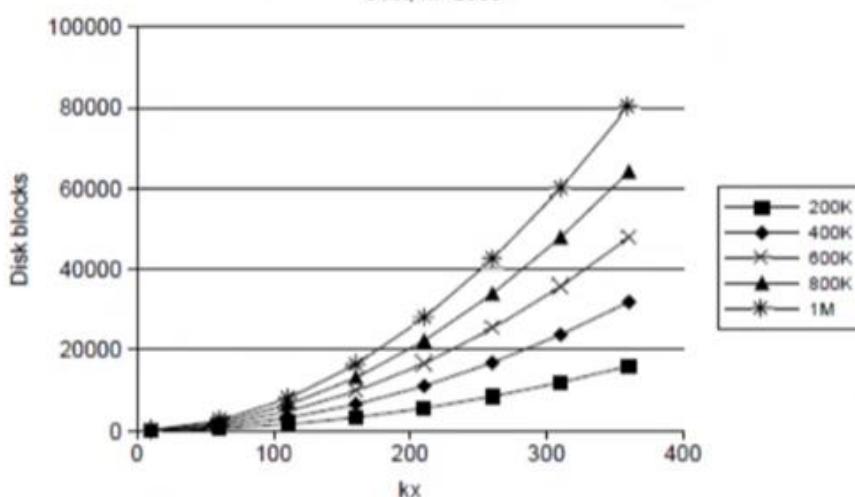
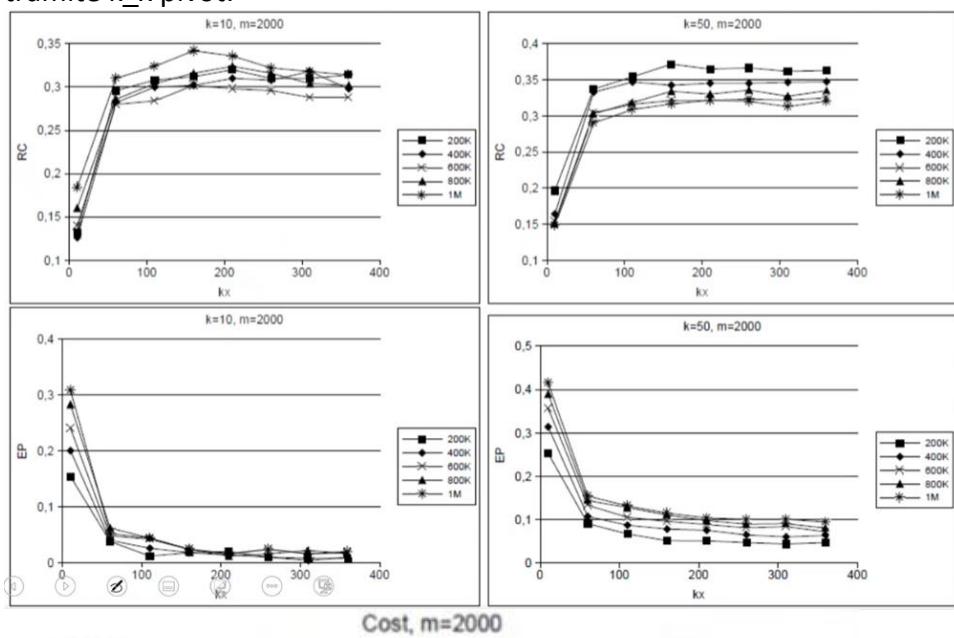
Performance figures: basic





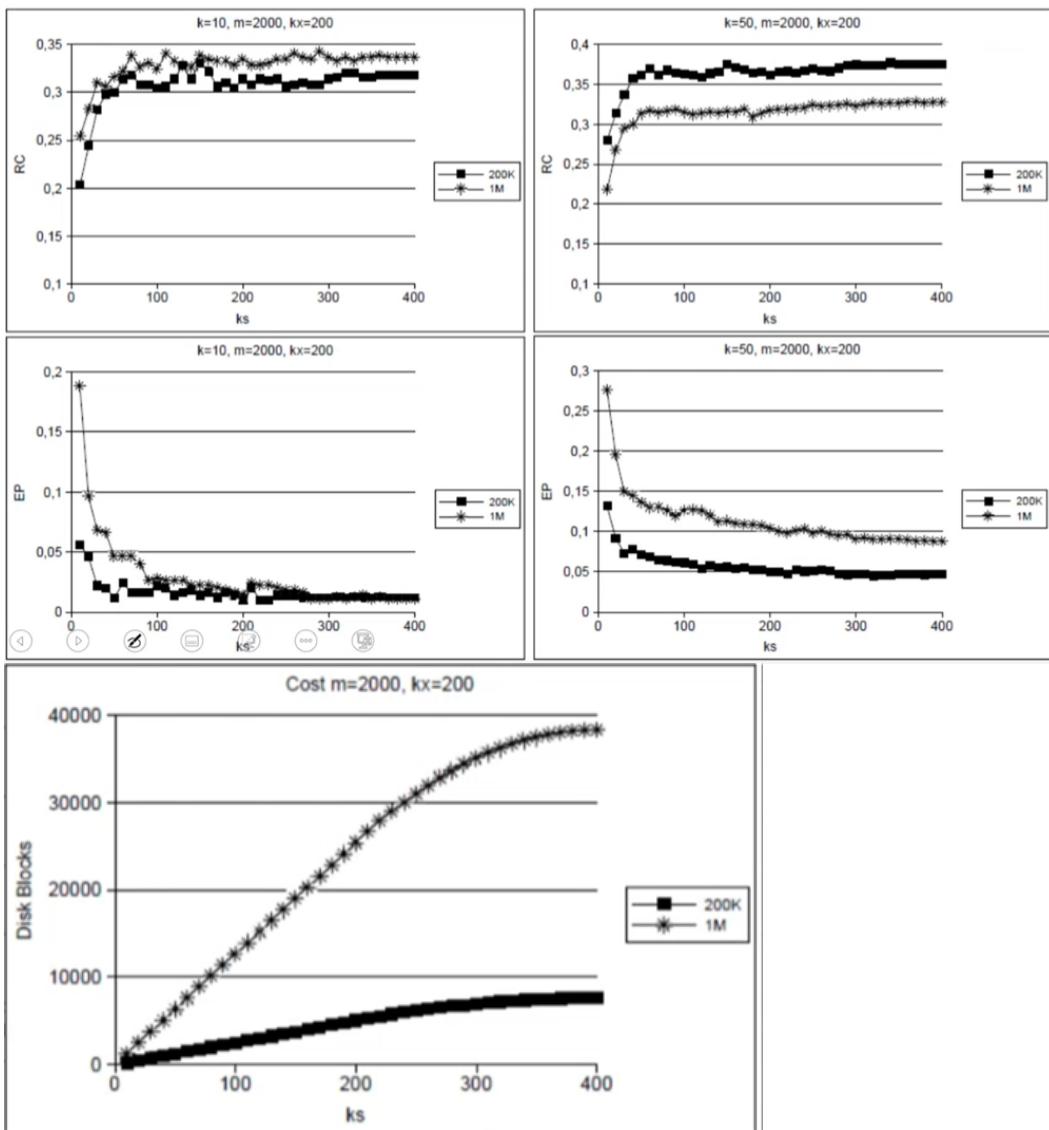
Questi primi grafici indicano le performance senza le tre migliori viste sopra.

Di seguito i grafici relativi alle performance quando applichiamo la rappresentazione degli oggetti tramite k_x pivot.



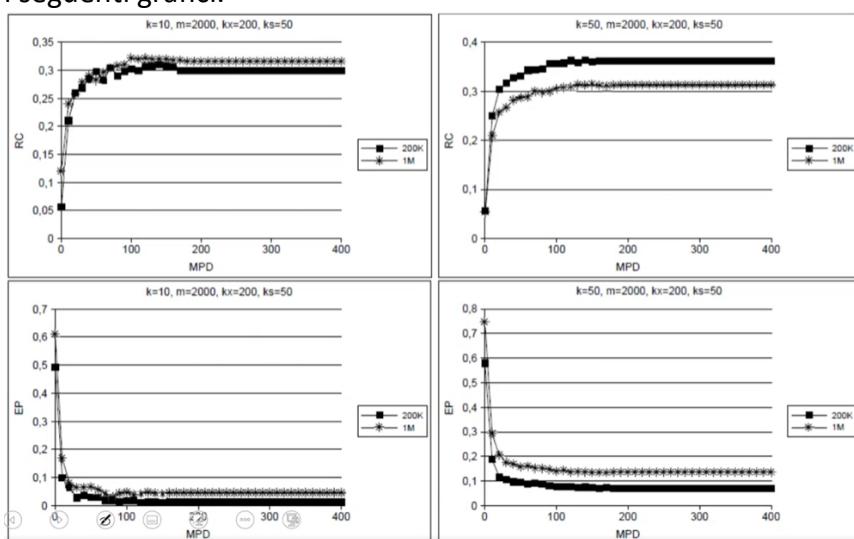
Si può vedere che il recall aumenta rispetto ai primi grafici, si può però vedere che dopo 80/100 pivot il recall non aumenta più. Possiamo quindi avere un incremento delle performance limitando di molto il costo.

Aggiungiamo ora la rappresentazione limitata della query con k_s .

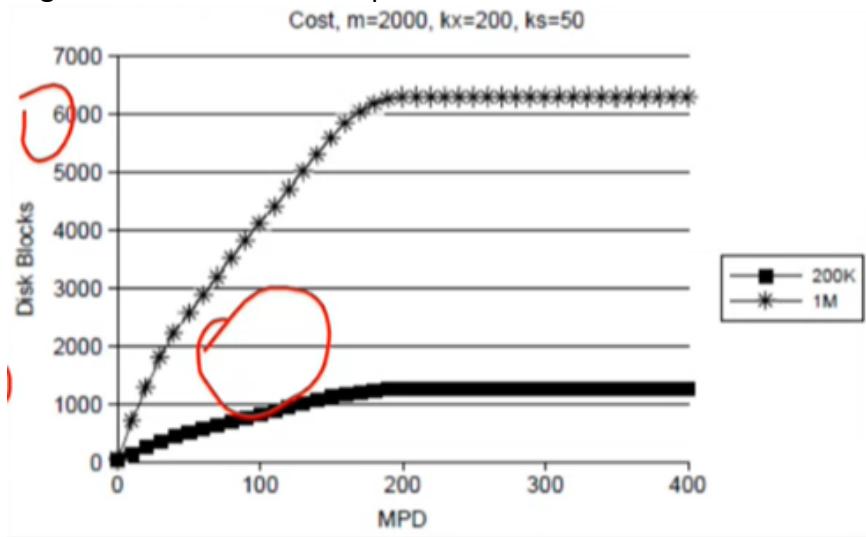


Si può vedere come dopo 100 pivot l'accuratezza rimane la stessa, non ha quindi senso aumentare ancora la rappresentazione. Possiamo quindi prendere ad esempio 50 pivot per rappresentare la query e il costo verrà ulteriormente abbattuto.

Come ultima ottimizzazione aggiungiamo l'accesso solo a una parte della posting list e otteniamo i seguenti grafici.

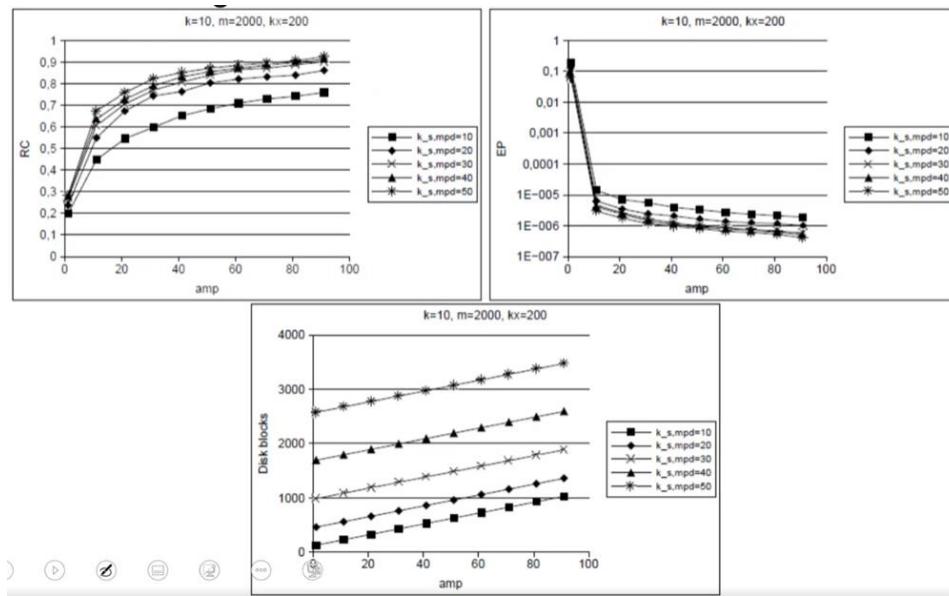


Sull'asse delle x è presente il fattore MPD che è la distanza massima che permettiamo di avere. Parte da 0 che significa accedere esattamente al blocco contenente gli oggetti che hanno il pivot proprio in quella posizione. Possiamo vedere che dopo una differenza pari a 50 non abbiamo un miglioramento in termini di performance.



Il grafico sopra riguarda i costi che sono ancora una volta abbattuti.

Abbiamo ridotto di molto il costo, ma l'accuratezza è sempre al 30%. Vediamo come aumentare l'accuratezza.



Applicando il riordinamento riusciamo a raggiungere il 90% di accuratezza con un fattore di amplificazione pari a 100. Questo aumenta un po' il costo totale in quanto andranno calcolate le distanze vere fra un gran numero di oggetti, tuttavia il costo rimane contenuto.

Come ultima cosa vediamo l'MI-File a confronto con le altre tecniche di approximate similarity search. Possiamo notare come questo approccio riesca a raggiungere il 90% di accuratezza con solo circa 3500 come costo totale. L'altro approccio migliore riesce a raggiungere solo il 70% ma con costi decisamente maggiori. Tutto questo è mostrato nella figura successiva.

