“Efficient Algorithms for Mining Outliers from Large Data Sets”

Articolo di S. Ramaswamy, R. Rastogi, K. Shim

Presentazione a cura di
Marcolini Serena, Marino Renato

Indice dell'intervento

- Definizioni di Outliers
- Algoritmi
- Risultati e confronti

Da large patterns a small patterns

- Gli algoritmi trattati fino ad ora (tra cui Birch, Cure, DBScan) trattano gli outliers, ma tentano di ridurne l'interferenza con il processo di ricerca di large patterns.

- In molte applicazioni reali gli outliers (small patterns) contengono le vere informazioni rilevanti (es. frodi sulle carte di credito, ricerca farmaceutica, applicazioni finanziarie).

Un confronto tra definizioni (1)

- “ Un punto *p* in un data set è un outlier rispetto ai parametri *k* e *d* se non più di *k* punti sono a distanza minore o uguale a *d* da *p*.” (Knorr, Ng, 1998)

- Vantaggi:
  - non richiede a priori la conoscenza della distribuzione dei dati.

- Svantaggi:
  - Il parametro *d* viene definito come input.
  - Non classifica gli outliers.
  - Problemi computazionali degli algoritmi che lo implementano.

Un confronto tra definizioni (2)

- "Dati un data set di *N* punti e i parametri *n* e *k*, un punto *p* è un outlier D_n^k se non ci sono più di *n-1* punti *p'* t.c. $D^k(p') > D^k(p)$"
  - *n* numero di outliers ricercati
  - *k* numero di vicini considerati
  - $D^k(p)$ distanza di p dal kNN

Vantaggi:

- Non richiede di specificare la distanza *d* in input.
- Gli outliers vengono classificati in fuzione della loro $D^k(p)$.
- Funziona con distanze metriche (L_p) e non metriche.

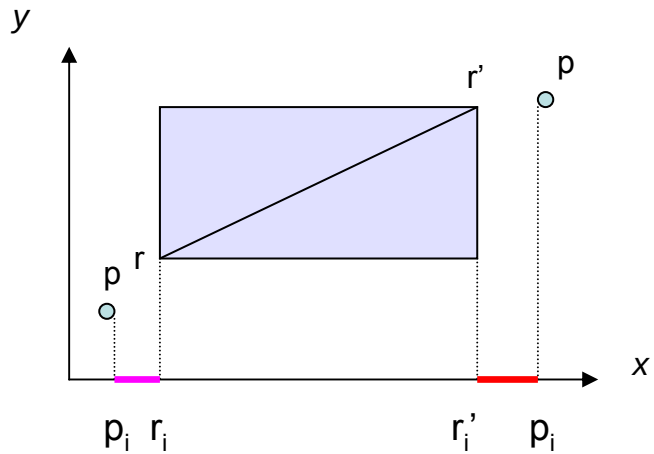
Notazioni (1)

- Distanza tra due punti:
  euclidea al quadrato

- Distanza tra p e R:

  p,R ϵ spazio δ-dimensionale

$$MINDIST(p,R) = \sum_{i=1}^{\partial} x_i^2$$



Dove

$$x_i = \begin{cases} r_i - p_i, p_i < r_i \\ p_i - r_i', r_i' < p_i \\ 0, altrimenti \end{cases}$$

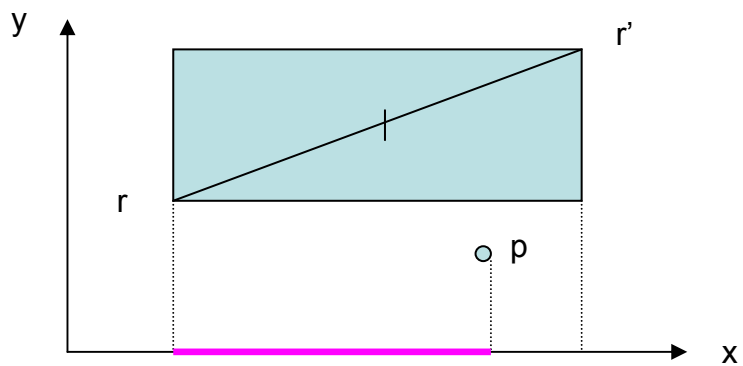
Notazioni (2)

- Distanza tra p e R:
  p,R ϵ spazio δ-dimensionale

$$MAXDIST(p,R) = \sum_{i=1}^{\partial} x_i^2$$

Dove

$$x_i = \begin{cases} r_i' - p_i, p_i < \dfrac{r_i + r_i'}{2} \\ p_i - r_i, altrimenti \end{cases}$$

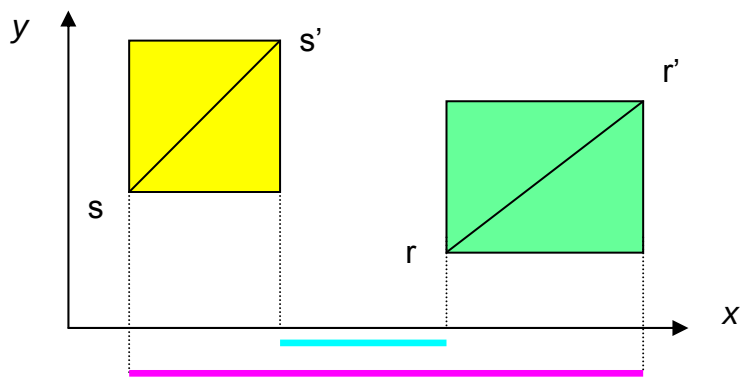


Notazioni (3)

- Distanza tra S e R: S,R ϵ spazio δ-dimensionale

$$MINDIST(S,R) = \sum_{i=1}^{\partial} x_i^2 \qquad x_i = \begin{cases} r_i - s_i', s_i' < r_i \\ s_i - r_i', r_i' < s_i \\ 0, altrimenti \end{cases}$$

$$MAXDIST(S,R) = \sum_{i=1}^{\partial} x_i^2 \qquad x_i = MAX \left\{ \left| s_i' - r_i \right|, \left| r_i' - s_i \right| \right\}$$



Algoritmi

- Nested Loop Algorithm
- Index Based Algorithm
- Partition Based Algorithm

Nested Loop Algorithm

Idea di base: per ogni punto p del DS calcola $D^k(p)$ e prende i primi n punti con il massimo valore di $D^k(p)$.

Calcolo di D^K:

- Mantenimento dei k-NN di p
- Scansione del data base per ogni punto p
- Check: per ogni q del data set, se dist(p,q) è minore della D^k corrente, q viene inserito nella lista dei k-NN
- Se la lista contiene più di k elementi si elimina il punto più lontano da p.

Index-Base Algorithm (1)

Idea di base vedi algoritmo precedente

Problema: complessità computazionale $O(N^2)$

Utilizzo di R^*-tree: diminuire il numero di computazioni nel calcolo delle distanze.

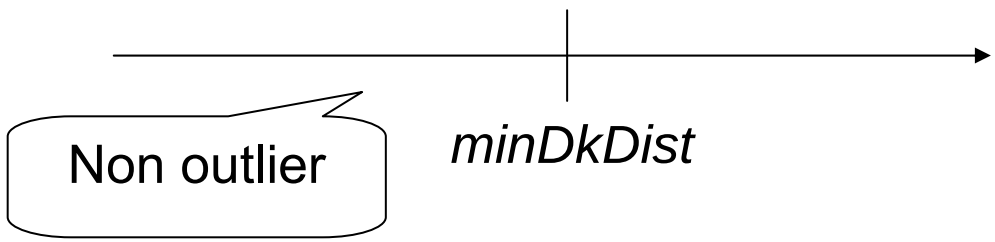
- Pruning optimization 1:

  se la distanza minima tra p e MBR di un nodo (MINDIST(p,Node)) è maggiore del $D^k(p)$ corrente

  ⇨ Pruning dell' intero sottoalbero

Index-Base Algorithm (1)

- Pruning optimization 2:
  1. Manteniamo in memoria una lista degli *n* outlier migliori trovati fino ad ora.
  2. Ogni outlier ha una sua D^k. Chiamiamo *minDkDist* la distanza minima tra queste.
  3. Ogni punto p, il cui valore corrente di $D^k(p)$ è minore o uguale a *minDkDist* , non sarà mai un oulier.



Index-Base Algorithm (2)

Scopo:calcolare D_n^k outlier

1,2 Costruzione dell'indice

3 outHeap:lista degli n punti con massimo D^k in ordine crescente (outliers)

5,6 Per ogni punto p si calcola $D^k(p)$= p.DkDist usando la procedura

7 Test sul valore di $D^k(p)$

9,10 Se Dk(p) è inserito si aggiorna la lista affinchè contenga *n* elementi

11 Si pone minDkDist uguale al primo elemento della lista outHeap

```
Procedure computeOutliersIndex(k,n)
begin
1. for each point p in input data set do
2.   insertIntoIndex(Tree, p)
3. outHeap :=0
4. minDkDist := 0
5. for each point p in input data set do {
6.    getKthNeighborDist(Tree.Root, p, k, minDkDist)
7.    if (p.DkDist> minDkDist){
8.       outHeap.insert(p)
9.     if (outHeap.numPoints()> n)outHeap.deleteTop()
10.   if (outHeap.numPoints( )= n)
11.      minDkDist := outHeap.top().DkDist
12. }
13. }
14. return outHeap
end
```

Index-Base Algorithm (3)

Scopo: calcolare $D^k(p)$ di un punto p esaminando i nodi del R*-tree

1 nodeList: contiene i nodi dell' indice ordinati secondo MINDIST da p ascendente.

3 nearHeap:lista dei k vicini di p esaminati, in ordine decrescente della loro distanza da p

4 Durante ogni iterazione si esamina il primo nodo della lista

8,9 .Si cercano i k vicini del punto p

10-12 controllo del numero di elementi in nearHeap, eventuale eliminazione ed aggiornamento di pDkDist

13 Pruning 2

17,18 Se il nodo è interior node inserimento dei nodi figli in nodeList

21Pruning 1

```
Procedure getKthNeighborDist(Root, p, k, minDkDist)
begin
1. nodeList := { Root }
2. p.Dkdist :=∞
3. nearHeap := 0
4. while nodeList is not empty do {
5.     delete the first element, Node, from nodeList
6.     if (Node is a leaf) {
7.         for each point q in Node do
8.             if (dist(p, q) < p.DkDist) {
9.                 nearHeap.insert(q)
10.                if (nearHeap.numPoints()> k)nearHeap.deleteTop()
11.                if (nearHeap.numPoints() = k)
12.                    p.DkDist :=dist(p, nearHeap.top())
13.                if (p.Dkdist<=minDkDist) return
14.            }
15.    }
16.    else {
17.        append Node's children to nodeList
18.        sort nodeList by MINDIST
19.    }
20.    for each Node in nodeList do
21.      if (p.DkDist <= MINDIST(p,Node))
22.          delete Node from nodeList
23. }
end
```

Partition Based Algorithm

Idea di base: Partizionamento del DS e pruning sulle partizioni che non contengono outliers.

Implicazioni: > velocità nel calcolo degli outliers

< overhead nella fase di preprocessing.

Macofasi del processo:

1. Generazione delle partizioni.
2. Calcolo del limite sup. e inf. di $D^k(p)$ per i punti in ogni partizione.
3. Selezioni delle partizioni che possono contenere outliers.
4. Calcolo degli outliers dai punti delle sole partizioni considerate.

1-Generazione delle Partizioni

- Usiamo la fase di pre-clustering di Birch (scala lineramente in N)
- Birch genera un set di clusters di dimensione uniforme che stanno in memoria.
- Ogni cluster è una partizione e viene rappresentata come un MBR.

N.B. non utilizziamo Birch per scovare outliers, ma solo per generare partizioni *!*

2-Calcolo del limite sup. e inf. di $D^k(p)$

- Per ogni partizione P, calcoliamo il limite superiore ed il limite inferiore della $D^k(p)$, validi per tutti i punti interni alla partizione.
  per ogni p ϵ P, P.lower <= $D^k(p)$ <= P.upper

- "P.lower e P.upper possono essere definiti determinando le L partizioni più vicine a P per MINDIST e MAXDIST tali che il numero totale dei punti di $P_1,\ldots,P_L$ è almeno k." (S. Ramaswamy, R. Rastogi, K. Shim).

Quindi…

Considero le *L* partizioni più vicine a *P* tali che il numero totale dei punti di $P_1,\ldots,P_L$ è almeno k.

P.lower è la *minima MINDIST (P,P_i)* per i = 1,…,L

P.upper è la *massima MAXDIST (P,P_i)* per i = 1,…,L

3-Determinazione delle partizioni candidate

- Si identificano le partizioni che potenzialmente contengono outliers (Candset) e si potano quelle rimanenti.

- Si definisce minDkDist dai bound dello step precedente:

  Siano $P_1,\ldots,P_L$ le partizioni con il massimo valore di *P.lower* tali che la somma dei punti sia almeno *n:*

  $minDkDist = min \{ P_i.lower \}$ per $1 <= i <= L$

- Data una partizione *P*, questa è una partizione candidata se *P.upper* >= *minDkDist*.

4-Ricerca degli outliers

- Per ogni partizione P chiamiamo P.neighbors l'insieme delle partizioni più vicine:

  $\{ P_i \} : MINDIST(P, P_i) <= P.upper$

- Procediamo con la ricerca degli outliers attraverso il calcolo della $D^k(p)$ per ogni punto p (Index Based Algorithm).

➡ se il punto p ϵ P, allora i soli altri punti da considerare per il calcolo di $D^k(p)$ sono solo quelli appartenenti a P.neighbor.

$$|P.neighbor| << |DataSet|$$

Partition Based Algorithm (1)

Scopo: selezionare le partizioni che contengono outliers

3 partHeap:lista delle partizioni con il valore P.lower maggiore contenenti almeno *n* punti, memorizzate in ordine crescente di P.Lower

8-13 se per una partizione P, Plower è più grande del valore corrente di minDkDist,la partizione viene inserita in partHeap e viene aggiornato il valore di minDkDist

13 minDkDist è posto uguale al primo valore della lista partHeap

17-22 per ogni partizione candidata P si va a costruire Pneighbors composte dalle partizioni Q che potenzialmente possono contenere il Kth NN per un punto i P.

23 la procedura restitusce l'insieme delle partizioni candidate da esaminare nell'ultimo step dell'algoritmo

```
Procedure computeCandidatePartitions(PSet, k, n)
begin
1.    for each partition P in PSet do
2.        insertIntoIndex(Tree, P)
3.   partHeap := 0
4.   minDkDist := 0
5.   for each partition P in PSet do {
6.    computeLowerUpper(Tree.Root, P, k, minDkDist)
7.    if (P.lower > minDkDist) {
8.      partHeap.insert(P)
9.      while partHeap.numPoints() _
10.           partHeap.top().numPoints()>=n do
11.           partHeap.deleteTop()
12.     if (partHeap.numPoints() >=n)
13.           minDkDist := partHeap.top().lower
14.  }
15.  }
16.  candSet := 0
17.   for each partition P in PSet do
18.   if (P.upper>= minDkDist) {
19.       candSet := candSet U {P}
20.       P.neighbors :=
21.           {Q: Q Є PSet and MINDIST(P,Q)<= P.upper }
22    }
23. return candSet
end
```

Partition Based Algorithm (2)

Scopo: calcolare P.lower e P.Upper della partizione P

3 lowerHeap: lista delle partizioni vicine ordinate per MINDIST da P decrescente

3 upperHeap: lista delle partizione vicine ordinate per MAXDIST da p decrescente

6-15 controllo MINDIST(Q,P), eventale inserimento in lowerHeap ed aggiornamento della lista

16-23 controllo MAXDIST(P,Q), eventale inserimento in upperHead ed aggiornamento della lista.

23 controllo con MinDkDis ed eventuale pruning

28,29 Se il nodo è interior node inserimento dei nodi figli in nodeList

31-34 controllo distanza del nodo dalla partizione considerata

```
Procedure computeLowerUpper(Root, P, k, minDkDist)
Begin
1. nodeList := { Root }
2. P.lower := P.upper := ∞
3. lowerHeap := upperHeap := 0
4. while nodeList is not empty do {
5.     delete the first element, Node, from nodeList
6.     if (Node is a leaf) {
7.       for each partition Q in Node {
8.         if (MINDIST(P,Q)<P.lower) {
9.           lowerHeap.insert(Q)
10.          while lowerHeap.numPoints() -
11.                lowerHeap.top().numPoints()>=k do
12.            lowerHeap.deleteTop()
13.        if (lowerHeap.numPoints() >= k)
14.           P.lower := MINDIST(P, lowerHeap.top())
15.      }
16.      if (MAXDIST(P,Q)< P.upper) {
17.         upperHeap.insert(Q)
18.         while upperHeap.numPoints() -
19.           upperHeap.top().numPoints() >= k do
20.           upperHeap.deleteTop()
21.         if (upperHeap.numPoints() >= k)
22.           P.upper := MAXDIST(P, upperHeap.top())
23.         if (P.upper <= minDkDist) return
24.       }
25.     }
26.   }
27.   else {
28.     append Node's children to nodeList
29.     sort nodeList by MINDIST
30.   }
31.   for each Node in nodeList do
32.     if (P.upper <= MAXDIST(P,Node) and
33.         P.lower <= MINDIST(P,Node))
34.         delete Node from nodeList
35. }
end
```

Caso applicativo

- Data base NBA, dati stagione 1998
  - 335 giocatori considerati
  - Normalizzazione dei valori delle colonne

- Specifiche di ricerca:
    - k=10 neighbors
    - n=5 outliers

Applicazione PBA

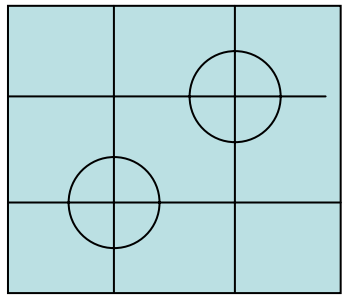
```
->findOuts.pl -n 5 -k 10 reb assists pts
          NAME   DIST  avgReb  (norm) avgAssts  (norm)   avgPts  (norm)
  Dennis Rodman  7.26  15.000  (4.376)   2.900  (0.670)   4.700 (-0.459)
 Rod Strickland  3.95   5.300  (0.750)  10.500  (4.922)  17.800  (1.740)
Shaquille Oneal  3.61  11.400  (3.030)   2.400  (0.391)  28.300  (3.503)
Jayson Williams  3.33  13.600  (3.852)   1.000 (-0.393)  12.900  (0.918)
    Karl Malone  2.96  10.300  (2.619)   3.900  (1.230)  27.000  (3.285)

->findOuts.pl -n 5 -k 10 steal blocks
           NAME  DIST    avgSteals (norm)    avgBlocks  (norm)
   Marcus Camby  8.44     1.100     (0.838)     3.700    (6.139)
Dikembe Mutombo  5.35     0.400    (-0.550)     3.400    (5.580)
  Shawn Bradley  4.36     0.800     (0.243)     3.300    (5.394)
   Theo Ratliff  3.51     0.600    (-0.153)     3.200    (5.208)
Hakeem Olajuwon  3.47     1.800     (2.225)     2.000    (2.972)
```

- Gli outliers sono quei giocatori che presentano un valore maggore di D^k
  - I giocatori che tendono a dominare in una o due colone e sono particolarmente scarsi in altre, risultano gli outliers più forti.
  - Non compaiono, invece, nella lista giocatori ben bilanciati in tutte le statistiche.
- Gli otuliers tendono ad essere più interessanti se vengono considerate poche dimensioni

Confronto fra gli algoritimi

- **Si considera un data set costruito artificialmente:**
  - Data set bidimensionale
  - 100 cluster (sferici di raggio 4) organizzati in una griglia 10 X 10



Parameter	Default Value	Range of Values
Number of Points (N)	101000	11000 to 1 million
Number of Clusters	100	
Number of Points per Cluster	1000	100 to 10000
Number of Outliers in Data Set	1000	
Number of Outliers to be Computed (n)	100	100 to 500
Number of Neighbors (k)	100	100 to 500
Number of Dimensions (δ)	2	2 to 10
Maximum number of Partitions	6000	5000 to 15000
Distance Metric	euclidean	

- **Parametri di confronto**
  1. Numero di punti nel Data Set
  2. Numero di outlier
  3. KthNN
  4. Numero di dimensioni

Confronto fra gli algoritimi (2)

Numero di punti nel Data Set

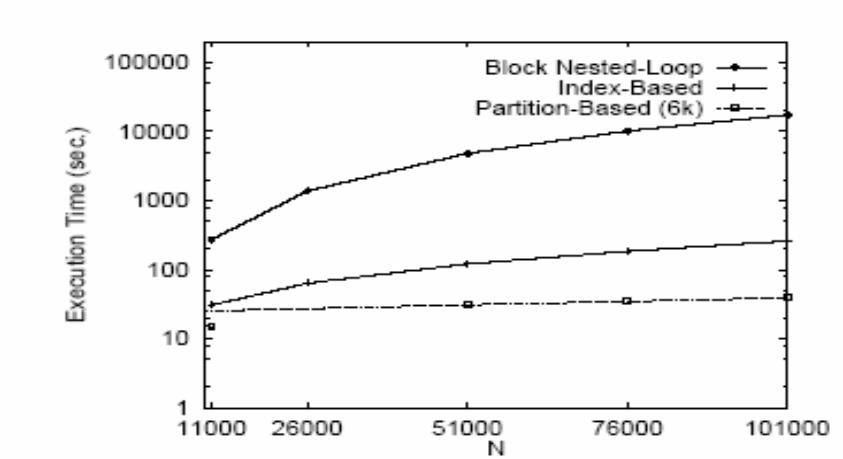
- Block nested-lopp: worst performer, $O(N^2)$

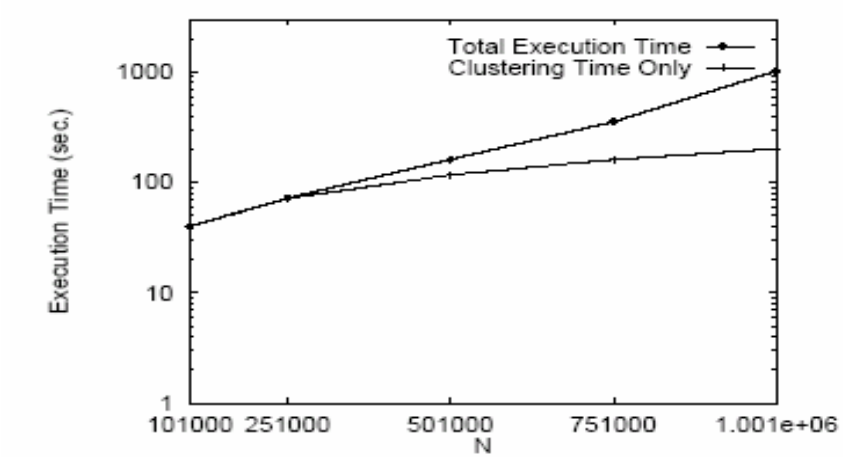
- Index- based: migliore di block nested-loop ma 2-6 volte più lento di PBA

- PBA: best performer
  - 75% partizioni completamente eliminate
  - Le operazioni di pruning sono meno efficaci se N cresce ma non cresce il numero di partizioni

Numero di outlier

- Crescita graduale del tempo di esecuzione per tutti gli algoritmi





Confronto fra gli algoritimi (3)

KthNN

- PBA non degrada al crescere di k

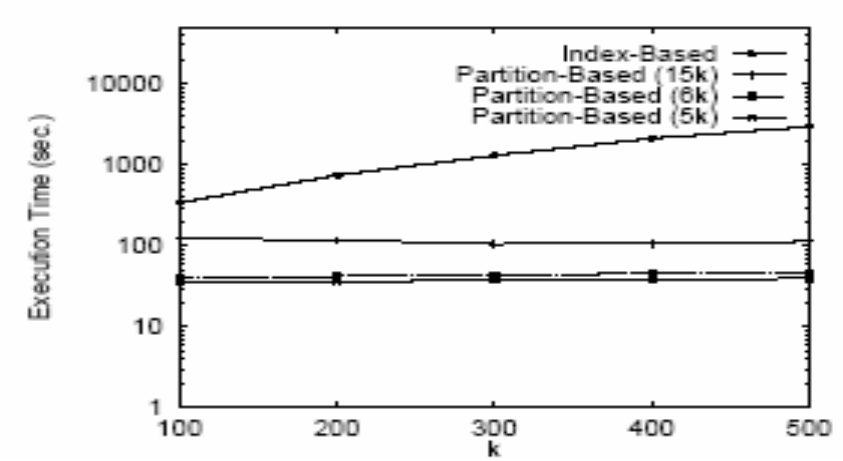
  Se k cresce il numero di partizioni candidate decresce significativamente ,in quanto k impone un valore di minDkDist maggiore e quindi più pruning, ma si tende ad avere molti vicini e quindi gli effetti si compensano.

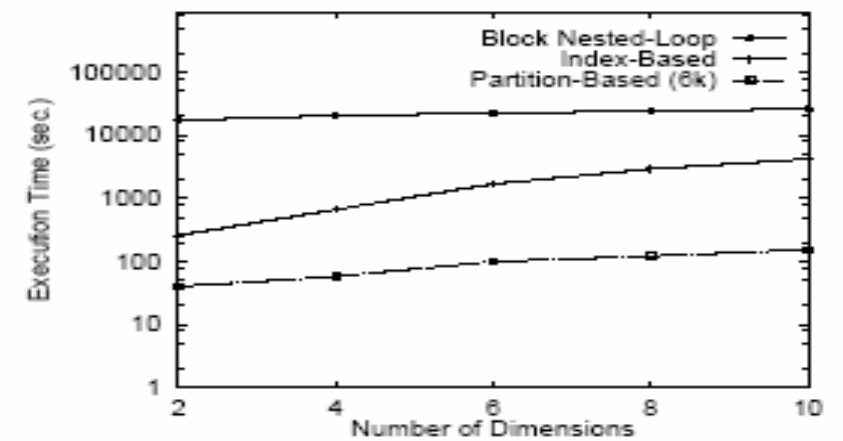
- Al crescere di k la performance dell'algoritmo peggiora, in quanto se ogni partizione contiene pochi punti il costo computazionale per il calcolo dei bounds si alza. In questo caso l'algoritmo tende a convergere con index-base

Numero di dimensioni

- PBA cresce sub-lineramente al crescere del numero di dimensioni

- Index based cresce molto rapidamente a causa dell'alto derivante dall'uso di R*-tree.





Conclusioni del confronto

- Partition based algoritm scala bene sia in relazione alla grandezza del Data Set (num. di punti), che in funzione della dimensionalità dello stesso.

- Partition based è più veloce di almeno un ordine di grandezza rispetto agli altri algoritmi considerati.