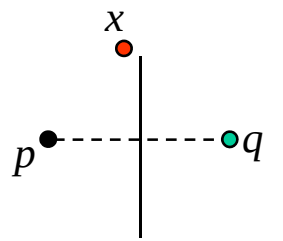
2.2.4 Indexbasierte Reverse k-Nächste-Nachbarn Suche

[Tao, Papadias, Lian. Int. Conf. Very Large Databases (VLDB), 2004]

– Idee:

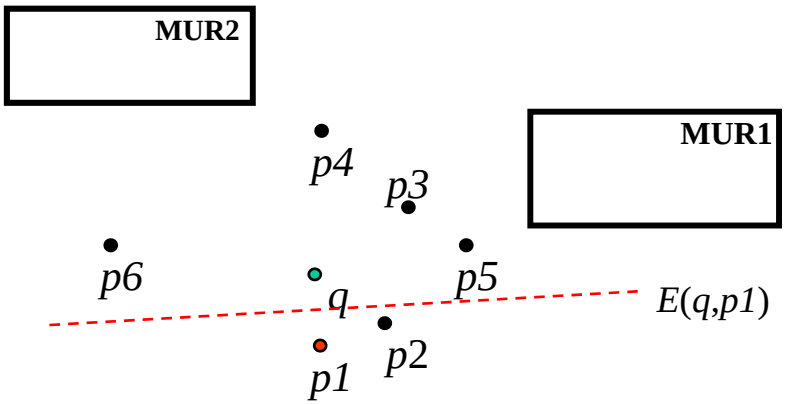
- Geometrisches Pruning:
  - Gegeben: Voronoi-Ebene zwischen *q* und beliebigen Punkt *p.*
  - Liegt ein Punkt *x* auf der Seite von dieser Voronoi-Ebene, kann *q* nicht NN von *x* sein und damit $x \notin$ RNN(*q*).



Voronoi-Ebene $E(p,q)$ zwischen p und q

- Iterative Berechnung der RkNN-Ergebnisse (Filter-Verfeinerung)
  - Berechne ein NN-Ranking der DB
  - Solange noch Objekte im Ranking sind:
    - » Rufe getNext() auf
    - » Wenn aktueller Punkt *p* nicht „hinter“ einer Voronoi-Ebene liegt, dann konstruiere neue Voronoi-Ebene *E*(*p*,*q*); *p* wird zur Kandidatenmenge hinzugefügt
    - » sonst (p liegt hinter einer Voronoi-Ebene): p kein Kandidat (muß aber für Verfeinerungsschritt (s.u.) berücksichtigt werden).
  - Punkte, die die Ebenen bestimmen, müssen verfeinert werden, d.h. für diese Punkte muss jeweils eine kNN-Anfrage berechnet werden

- Beispiel:



Bisherige Kandidaten:

{*p1*}

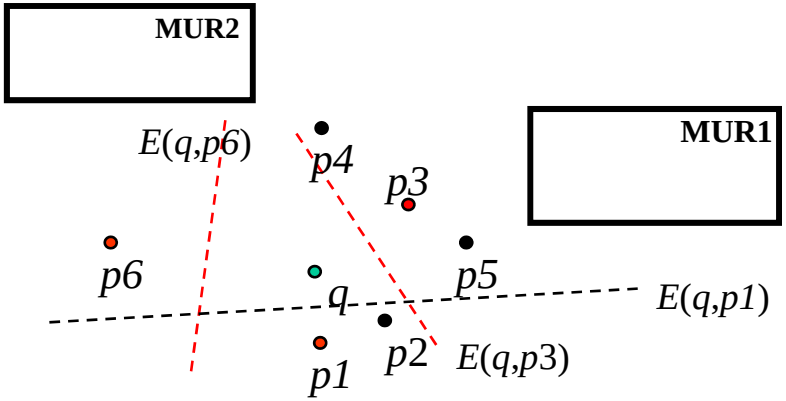
Inhalt des Rankings

p2: → ausschließen (pruning)
p3: → evaluieren
p4: → evaluieren
p5: → evaluieren
p6: → evaluieren
MUR1: verfeinern
MUR2: verfeinern

– Evaluierung von *p3*

» Bilde Ebene *E*(*q*,*p3*) → *p4, p5* und MUR1 nicht mehr evaluieren/verfeinern (diese liegen nun hinter *E*(*q*,*p3*))

» MUR2 muss weiterhin verfeinert werden



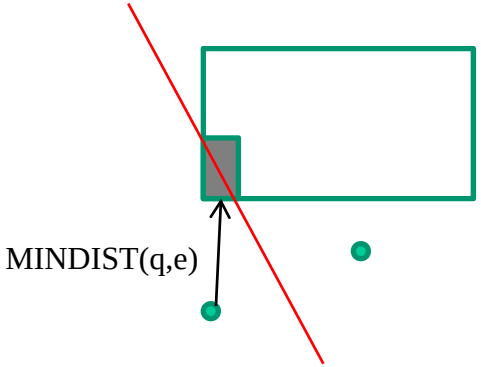
Bisherige Kandidaten:

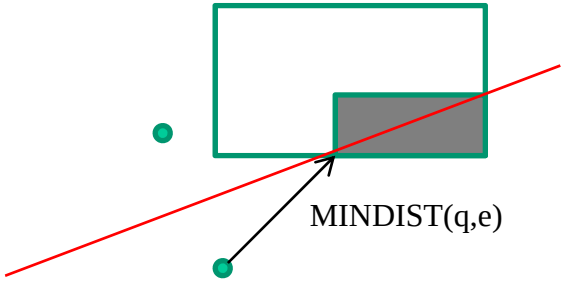
{*p1*, *p3*, *p6*}

Inhalt des Rankings (ungeordnet):

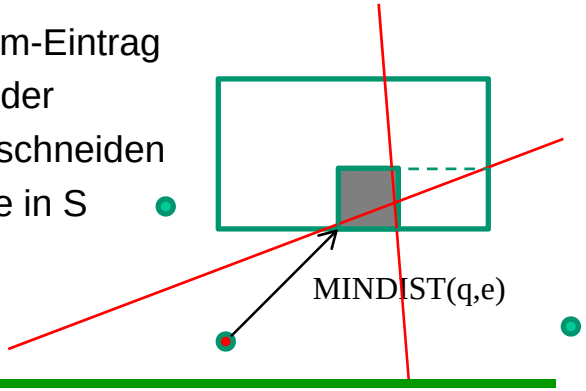
MUR2: verfeinern

- Trimmen
  - Partielles abschneiden (trimmen) von Seitenregionen (Rechtecken) bzgl. einer Pruningebene





  - Anpassung der MINDIST(q,e) nach dem Trimmen
    - ➔ führt eventuell zur Erhöhung der MINDIST(q,e)
    - ➔ erhöht die Chance, dass Seitenregion *e* früher ausgefiltert (geprunt) werden kann.
  - Funktion ***trim****(q,S,e)*:
    - » q:=Anfrageobjekt, S:=Objektmenge, e:=R-Baum-Eintrag
    - » Berechnet **minimale Distanz** zwischen q und der Region N^{resM}, die durch das kollektive partielle Abschneiden (trimmen) der Seitenregion von e durch die Punkte in S gebildet wird.
    - » Wenn $N^{resM} = \varnothing$, dann *trim(q,S,e)* = ∞



– Algorithmus (Filter):

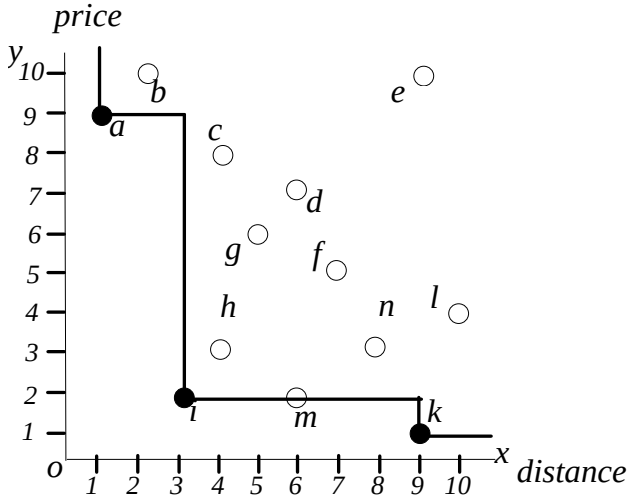
TPL-filter(*q*) // Objekt q ist Anfrageobjekt
apl = **LIST OF** (dist:Real, e:entry) **ORDERED BY** dist **ASCENDING**
apl = [(0.0, *root*)]
$S_{cnd} = \emptyset$; $S_{rfn} = \emptyset$;
WHILE NOT apl.isEmpty() **DO**
(d,e) := apl.pop_first_element;
IF *trim*(*q*,S_{cnd},*e*.getRegion)=∞ **THEN**
$S_{rfn}=S_{rfn} \cup \{e\}$;
ELSE // *e* ist (oder enthält) evtl. ein Kandidat
IF *e* is a point object **THEN**
$S_{cnd}=S_{cnd} \cup \{e\}$;
ELSE // *e* ist ein Directory-Eintrag
FOR i=0 **TO** e.size() **DO**
IF *trim*(*q*,S_{cnd},*e*.getRegion(i) = ∞ **DO**
$S_{rfn} = S_{rfn} \cup \{e\}$;
ELSE
apl.insert(trim(q,Scnd,e.getRegion(i)),e.getChild(i));
END IF;
END IF;
END IF;

Quelle: [Tao, Papadias, Lian. Int. Conf. Very Large Databases (VLDB), 2004]

2.2.5 Indexbasierte Skyline-Anfrage

– Anforderungen:

- Gegeben:
  - Menge von d-dimensionalen Punkte (Objekte)
  - Indexierung mittels R-Baum



- Gesucht:
  - Alle Objekte, die von keinem anderen Objekt dominiert werden.
- Ziele:
  - Wenig Seitenzugriffe
  - Wenig Dominanzüberprüfungen (Objektvergleiche)
  - Möglichst früh erste Ergebnisse ausgeben

- Grundsätzlich viele unterschiedliche Ansätze
  - Hauptspeicher-basiert ←→Sekundärspeicher-basiert
  - Iterative Berechnung ←→ Nicht-Iterative Berechnung

  Skyline-Anfrage Varianten:
  - Mit explizitem Anfrageobjekt(en) (dynamische Skyline)
  - zusätzliche Bedingungen
  - andere Skylinevarianten: z.B: Top-k-Dominanz, etc.

- Bekannteste Ansätze die auf Sekundärspeicher beruhen:

  (Zusammenfassung aus [Papadias et al., ToDS 2005])
  - Divide-and-Conquer, Block-Nested Loop [Borzsonyi et al., 2001]
  - Sort First Skyline [Chomicki et al., 2003]
  - Bitmap, Index [Tan et al., 2001]
  - Nearest-Neighbor [Kossmann et al., 2002][Papadias et al., ToDS 2005]

    Eigenschaften:
    - » Sekundärspeicherbasiert
    - » Erfüllen alle drei Ziele:

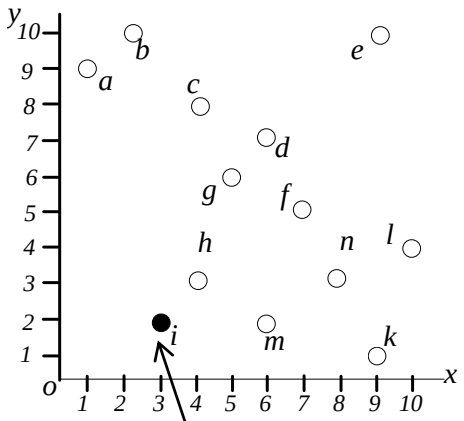
      - wenig Seitenzugriffe und Dominanzüberprüfung mittels Index (R-Baum).
      - Erste Ergebnisse werden frühzeitig ausgegeben durch iterative Verarbeitung.

– Nächste-Nachbarn-Skyline (NNS) Algorithmus:

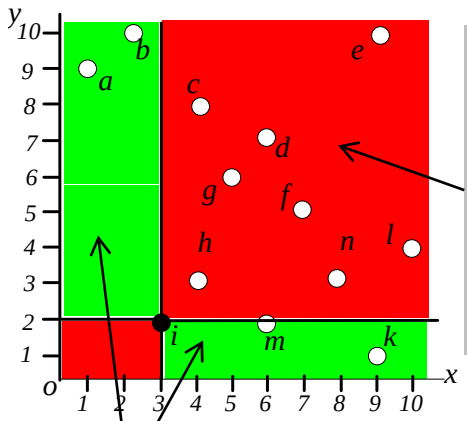
[Kossman et al., VLDB 2002]

Prinzip:

- Benutzt Nächste-Nachbarn-Suche zur (rekursiven) Partitionierung des Suchraums



Nächster Nachbar
(des Koordinaten-Ursprungs)
→ erstes Skyline-Ergebnis

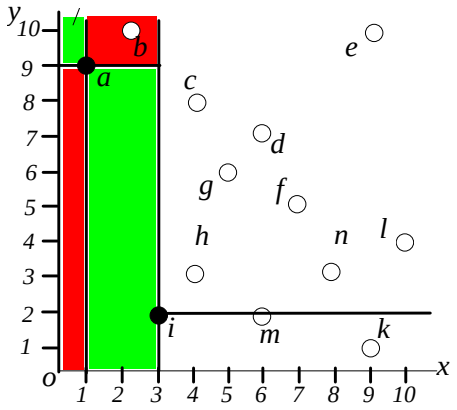


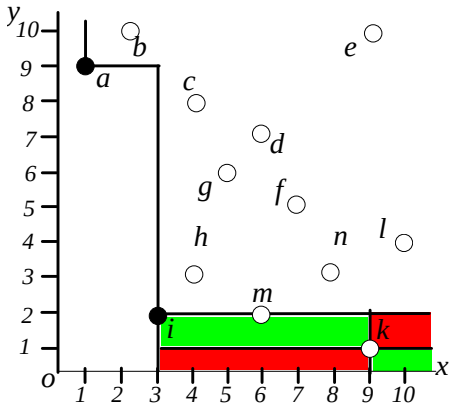
Raumpartitionen mit
weiteren Skyline-Kandidaten

Raumpartition mit Objekten die von Objekt i dominiert werden => Objekte gehören nicht zum Ergebnis (*true drops*).

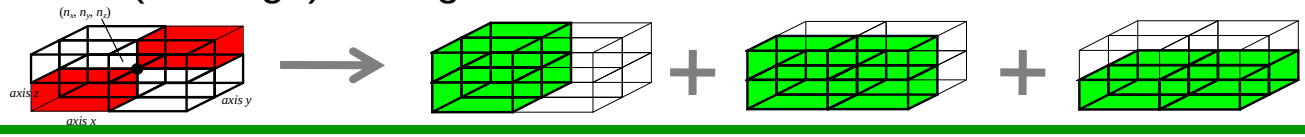
- Nächste-Nachbar-Suche kann durch R-Baum Index beschleunigt werden (z.B. Alg.: k-NN-Index-HS, siehe Folie 63).

- Nächste-Nachbar-Suche wird zur weiteren Partitionierung **in jeder** Kandidaten-Suchregion rekursiv fortgesetzt.





- Vorteile:
  - Verwendung von effizienten Methoden zur NN-Suche.
  - Erste (relevante) Resultate können schnell ausgegeben werden.
- Nachteile:
  - Im d-dimensionalen Raum führt jedes gefundene Skyline-Objekt (Punkt) zu d weiteren Fensteranfragen.
    → viele redundante Anfragen → Duplikateliminierung
    → viele (unnötige) Anfragen auf leeren Raum



– Branch-and-Bound Skyline (BBS) Algorithm:

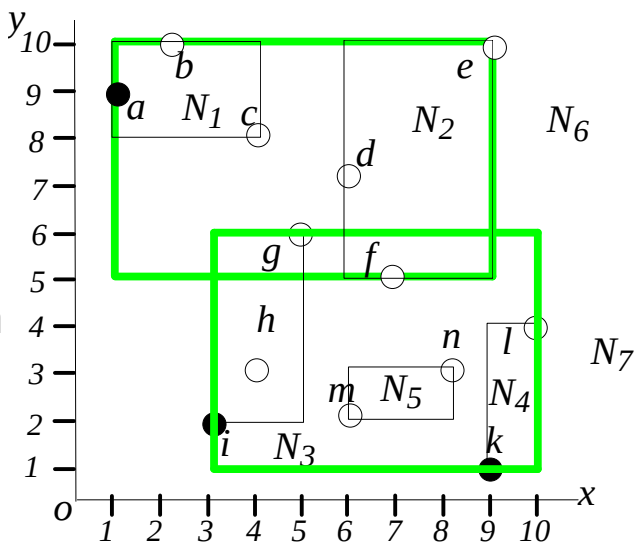
[Papadias et al., ToDS 2005]

Prinzip:

- Idee: Datenorientierte Suche statt Datenraumorientierte Suche
  - Vermeidung von Suche in “leeren” Datenraumpartitionen.
  - Kandidaten werden direkt über einen Index (R-Baum) ermittelt.
- Prioritäts-basierte Suche des nächsten Skyline-Objektes
  - Priorität entsprechend Manhattan-Distanz zum Koordinaten-Ursprung
  - Iterative Verfeinerung des Index (R-Baum) mittels Prioritätsliste (vgl. k-NN-Index-HS, Folie 63)

→ Verwendung eines Heaps aufsteigend sortiert über MINDIST(e,(0,0)),
e := Seitenregion oder Objekt (Punkt)

→ Verwendung einer Liste mit bereits gefundenen Skylineobjekten zum Prunen von anderen Seitenregionen / Objekten

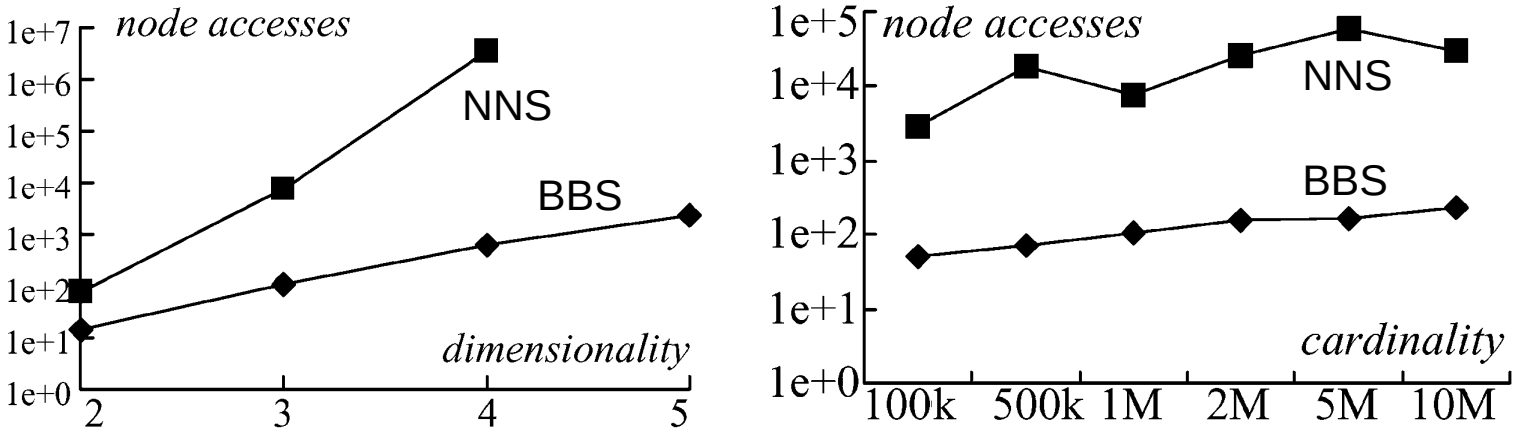


- Algorithmus:

Algorithm BBS (R-tree R)
S=Ø
Füge alle Einträge der Wurzel R in den Heap ein
Solange Heap nicht leer:
    entferne ersten Eintrag e
    wenn e von einem Punkt in S dominiert wird, verwerfe e
    sonst (e ist nicht dominiert)
        wenn e kein Datenpunkt ist
            für jedes Kind e_i von e
                falls e_i nicht von einem Punkt in S dominiert wird, füge e_i in den Heap ein
        sonst (e ist ein Datenpunkt)
                füge e in S ein

- Vorteile:
  - Vorzeitige Ausgabe von ersten Resultaten
  - Keine unnötige Partitionierung des Datenraums → geeignet auch für Suchraumdimensionen > 3 (im Gegensatz zu NNS)
  - BBS ist optimal bzgl. der Seitenzugriffe im R-Baum (I/O-optimal)

- Experimenteller Vergleich: NNS ←→ BBS
  - Datensatz: 1 Mio. Objekte gleichmäßig verteilt

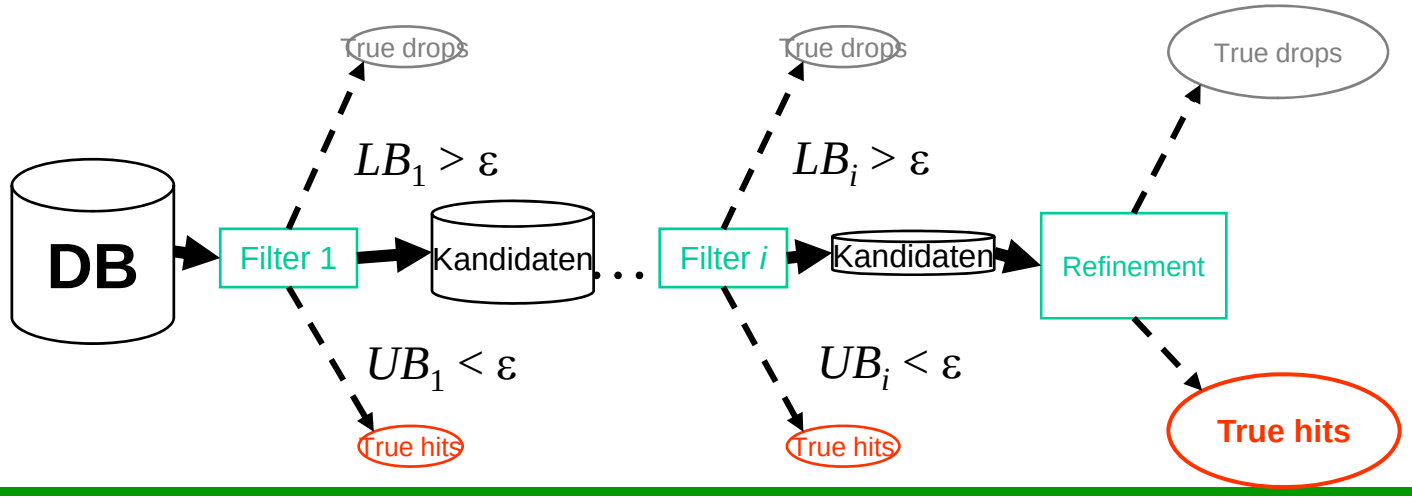


  - BBS schlägt NNS bzgl. I/O-Kosten über mehrere Größenordnungen

2.3 Mehrstufige Nachbarschafts-Anfragebearbeitung

2.3.1 Mehrstufige Bereichsanfrage

- Anfrage: Ergebnis enthält alle Objekte, die höchstens eine Distanz von ε zu Anfrageobjekt *q* haben.
- Filter:
  - filter-dist$_{LB}$(o,q) > ε => dist(o,q) > ε => Objekt o kann ausgeschlossen werden (true drop)
  - filter-dist$_{UB}$(o,q) < ε => dist(o,q) < ε => Objekt o gehört zum Resultat



- Mehrstufige-Bereichsanfrage Algorithmus: (Filter-/Refinement)
  - Lower Bounding Filterdistanz $dist_{LB}$
  - Upper Bounding Filterdistanz $dist_{UB}$

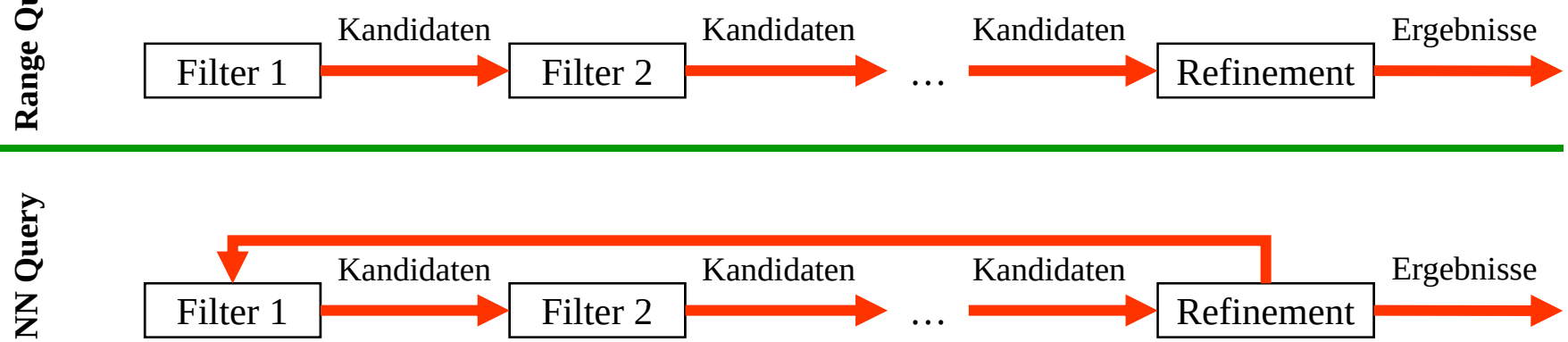
```
RQ-MultiStep(DB, q, ε)
    result = ∅;
    candidates = ∅;

    // Filter
    FOR i=1 TO n DO
        o = DB.getObject(i);
        IF dist_UB(q,o) ≤ ε THEN
            result := result ∪ o;
        ELSE IF dist_LB(q,o) ≤ ε THEN
            candidates := candidates ∪ o;

    // Refinement
    FOR i=1 TO candiates.size() DO
        c = candidates.getObject(i);
        IF dist(q,c) ≤ ε THEN
            result := result ∪ c;
    RETURN result;
```

2.3.2 Mehrstufige k-Nächste-Nachbarn Anfrage

- Allgemeines
  - Algorithmen verwenden meist nur LB-Filter
  - Bei mehreren Filterschritten: $dist_{Filter1} \leq dist_{Filter2} \leq \ldots$
  - Unterschied zu Bereichsanfragen:
    - » RQs können durch einfache Hintereinanderschaltung der Filterschritte und der Verfeinerung ausgewertet werden
    - » Bei NN-Queries nicht so leicht möglich, da der NN-Kandidat des (ersten) Filters nicht notwendigerweise der exakte NN sein muss
    - » Bei geeigneter Filterdistanz ist es aber wahrscheinlich, dass exakter NN unter den ersten NN-Kandidaten des Filterschritts ist
    - » Rückmeldung der im Refinement ermittelten Distanzen an den Filterschritt um aufgrund des Filters Objekte auszuschließen



- Es gibt verschiedene Auswertungsstrategien basierend auf LB-Filter
  - Auswertung mit Bereichsanfrage [Korn et al.,VLDB 1996]
  - Auswertung mit unmittelbarer Verfeinerung
  - Auswertung nach Priorität
- Auswertung mit Bereichsanfrage

  [Korn, Sidiropoulos, Faloutsos, Siegel, Protopapas. Proc. Int. Conf. Very Large Databases (VLDB), 1996]
  [Korn, Sidiropoulos, Faloutsos, Siegel, Protopapas. TKDE 10(6), 1998]
  - Idee
    - » Verfeinerungsdistanz ε eines beliebigen Punktes ist obere Schranke für die NN-Distanz
    - » Folge: ist p der NN von q, so gilt $dist(p, q) \leq \varepsilon$ und $LB_{Filter}(p, q) \leq \varepsilon$
    - » Also: $p \in RQ(q, \varepsilon)$
    - » Gutes ε ist z.B. der NN von q bzgl. der Filterdistanz
  - Prinzip
    - » Auf Filterebene wird zunächst eine NN-Anfrage ausgeführt
    - » Das resultierende Objekt wird verfeinert
    - » Anschließend wird eine Bereichsanfrage (RQ) ausgeführt (mit Index oder ebenfalls mehrstufig)
    - » Auf dem (hoffentlich kleinen) Ergebnis der RQ wird der exakte Test (Refinement) durchgeführt

– Algorithmus

NN-MultiStep-RQ(DB, *q*)
r = NN-Query auf der Filterdistanz; // beliebig implementierbar
ε = dist(*q*, *r*);
candidates = **RQ-MultiStep**(DB, *q*, ε);
result = *r* ;
stopdist = ε;

// Refinement
FOR EACH p∈candidates **DO**
IF dist(p,q) ≤ stopdist **THEN**
stopdist = dist(*q*, *p*)
result = *p*;
RETURN result;

– Vorteil
» Einfacher Algorithmus
– Nachteil
» Leistung stark von Filterselektivität abhängig: schlechter Filter => großes ε => große Ergebnismenge der RQ => hohe Kosten für Verfeinerung

- Auswertung mit unmittelbarer Verfeinerung
  - Idee
    - » Jedes Objekt, das nicht aufgrund des Filters ausgeschlossen werden kann, wird sofort verfeinert
    - » Einbau in einen beliebigen NN-Algorithmus, z.B. in NN-Index-Simple-TS (S. 61)
  - Algorithmus

```
NN-MultiStep-Simple(pa, q)        // pa = Diskadress z.B. der Wurzel des Indexes
    result = ∅;
    p := pa.loadPage();
    IF p.isDataPage() THEN
        FOR i=0 TO p.size() DO
            IF dist_Filter(q, p.getObject(i)) ≤ stopdist THEN
                IF dist(q, p.getObject(i)) ≤ stopdist THEN
                    result := getObject(i);
                    stopdist = dist(q, p.getObject(i));
    ELSE                          // p ist Directoryseite
        FOR i=0 TO p.size() DO
            IF MINDIST(q, p.getRegion(i)) ≤ stopdist THEN
                result := NN-MultiStep-Simple(p.childPage(i), q)
    RETURN result;
```

- Vorteil
  - » Gute Speicherplatzkomplexität (je nach NN-Algorithmus!!!), da keine Kandidaten zwischen gespeichert werden müssen
  - » Einfache Erweiterung eines beliebigen NN-Algorithmus
- Nachteil
  - » Hohe Verfeinerungskosten (fast alle Punkte), wenn Filter wenig selektiv ist oder NN-Algorithmus langsam konvergiert

- **Auswertung nach Priorität**

  [Seidl, Kriegel. Proc. ACM Int. Conf. Management of Data (SIGMOD),1998]

  - Auf Filterebene läuft „Ranking Query“ ab

    (Erweiterung von k-NN-Index-HS auf Folie 63)
    - » Funktion getNext(): liefert beim ersten Aufruf den 1. Nachbarn, beim zweiten Aufruf den 2. Nachbarn, usw.
    - » Rufe solange getNext() auf, bis das erhaltene Objekt die aktuelle pruningdist überschreitet.
    - » Verfeinere das erhaltene Objekt und passe ggf. die pruningdist an.
  - Vorteil
    - » Beweisbar: Algorithmus optimal bzgl. der Anzahl der Verfeinerungen, d.h. eine minimale Anzahl von Kandidaten werden verfeinert.
  - Nachteil
    - » Komplexität des Ranking-Algorithmus (Speicher und/oder Zeit)

– Algorithmus

k-NN-MultiStep-Optimal(DB, *q*)

Globale Variablen: pruningdist = +∞;

result = ∅; // Heap mit (o,dist(q,o)) tupel, erster Eintrag hat höchsten dist()-wert, maximale Heapgröße = k (Bei Überlauf wird letztes Element gelöscht).

Ranking = initialisiere Ranking bzgl. *q* auf Filterdistanz

FOR i=1..k **DO**

p = Ranking.getNext();

result.insert(p,dist(q,p)); // Verfeinerung

pruningdist = result.first.dist;

REPEAT

p = Ranking.getNext();

IF $dist_{LB}(p, q) \leq$ pruningdist **THEN** // Filter

IF $dist(q, p) \leq$ pruningdist **THEN** // Verfeinerung

result.insert(*p*);

pruningdist = result.first.dist;

UNTIL $dist_{LB}(p, q) >$ pruningdist;

RETURN result;