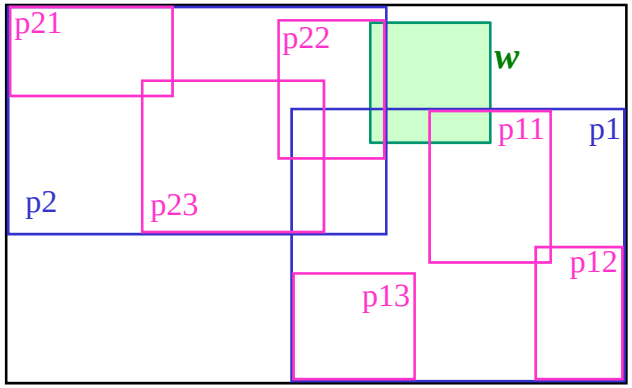
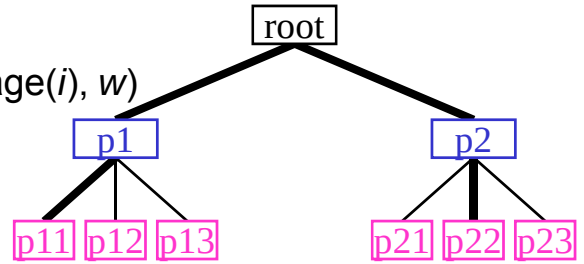
2.2 Indexbasiere Nachbarschaftssuche

2.2.1 Indexbasierte Fensteranfrage

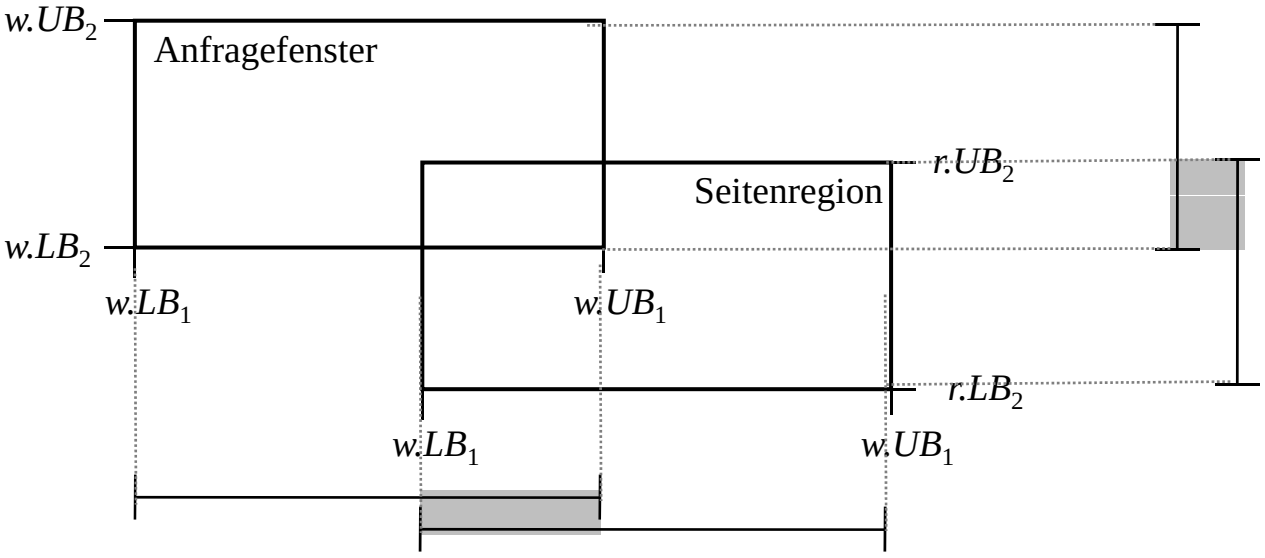
- Algorithmus

```
// w := Anfragefenster
WQ-Index(pa,w)          // pa initialisiert mit Diskadress der Wurzel des R*-Baums
    result = ∅;
    p := pa.loadPage();
    IF p.isDataPage() THEN
        FOR i=0 TO p.size() DO
            o = p.getObject(i);
            IF o ∈ w THEN
                result := result ∪ o;

    ELSE                    // p ist Directoryseite
        FOR i=0 TO p.size() DO
            IF intersect(w, p.getRegion(i)) THEN
                result := result ∪ WQ-Index(p.childPage(i), w)
    RETURN result;
```





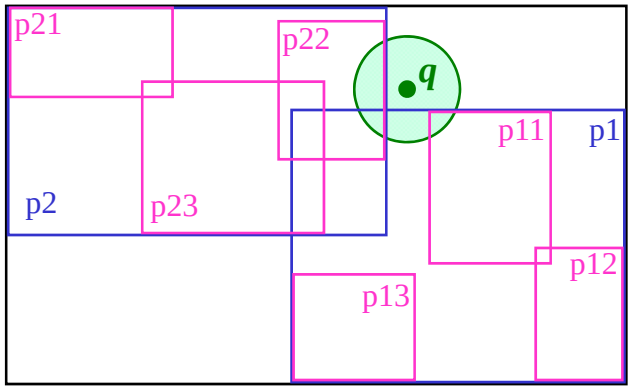
- Auswertung des Intersect-Prädikates *intersect*():
  - Test ob Anfragefenster *w* ein anderes achsenparalleles Rechteck *r* (z.B. R*-Baum-Seitenregion) schneidet
  - Gegeben:
    - Anfragefenster $w = [(w.LB_1, w.UB_1), \ldots, (w.LB_d, w.UB_d)]$
    - Seitenregion $r = [(r.LB_1, r.UB_1), \ldots, (r.LB_d, r.UB_d)]$
  - Anfragefenster w scheidet Seitenregion r genau dann, wenn sich alle entsprechenden Projektionen auf die d eindimensionalen Räume schneiden.

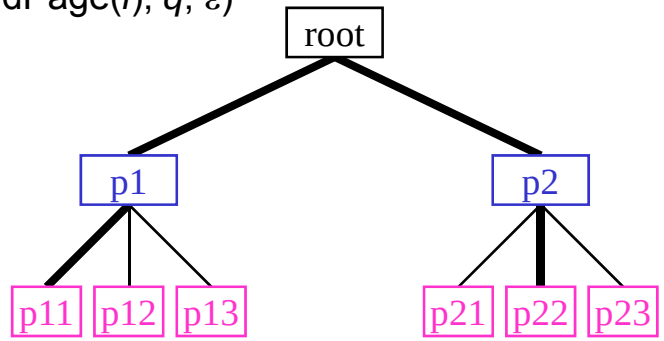


2.2.2 Indexbasierte Bereichsanfrage

- Algorithmus:

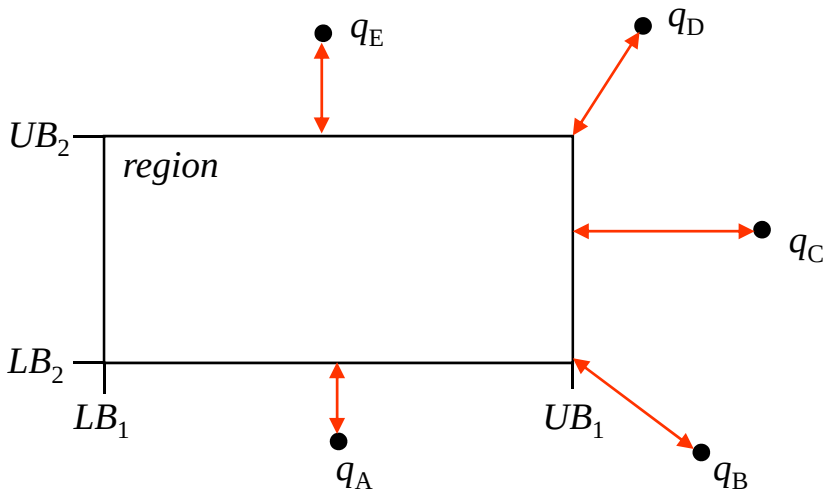
```
RQ-Index(pa,q)          // pa initialisiert mit Diskadress der Wurzel des R*-Baums
    result = ∅;
    p := pa.loadPage();
    IF p.isDataPage() THEN
        FOR i=0 TO p.size() DO
            o = p.getObject(i);
            IF dist(q,o) ≤ ε THEN
                result := result ∪ o;

    ELSE                    // p ist Directoryseite
        FOR i=0 TO p.size() DO
            IF MINDIST(q, p.getRegion(i)) ≤ ε THEN
                result := result ∪ RQ-Index(p.childPage(i), q, ε)
    RETURN result;
```





- MINDIST
  - Test ob Anfragebereich (q,ε) sich mit Seitenregion schneidet
  - Minimale Distanz zwischen Anfragepunkt und allen Punkten der Seitenregion (=> Lower Bound!!!)
  - Beispiel: Berechnung der MINDIST für L_2-Norm

$$\text{MINDIST}(region, q) = \sqrt{\sum_{0<i\le d} \begin{cases} (region.LB_i - q_i)^2 & \text{if } q_i \le region.LB_i \\ 0 & \text{if } region.LB_i \le q_i \le region.UB_i \\ (q_i - region.UB_i)^2 & \text{if } region.UB_i \le q_i \end{cases}}$$

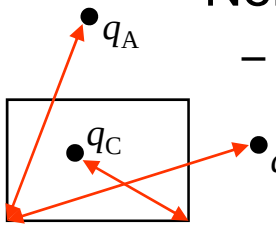


2.2.3 Indexbasierte *k*-Nächste-Nachbarn Anfrage

– Tiefensuche (Depth-first search nach [RKV 95])

[Roussopoulos, Kelley, Vincent. Proc. ACM Int. Conf. Management of Data (SIGMOD), 1995]

- Vermeidet langsame Einschränkung des Suchraums durch
  - Verwendung der Seitenregionen zur Abschätzung der k-NN-Distanz
  - Priorisierung der Tiefensuche nach Distanz der Seitenregion zur Query
- Neben MINDIST weitere Abschätzungen der NN-Distanz durch:
  - MAXDIST
    - » Maximale Distanz zwischen Query und allen Punkten der Seitenregion
    - » NN-Distanz kann nicht schlechter als MAXDIST werden



$$\text{MAXDIST}(region, q) = \sqrt{\sum_{0<i\le d} \max\{(q_i - region.UB_i)^2, (q_i - region.LB_i)^2\}}$$

  - MAXDIST aller bisher bekannten Seitenregionen wird zum frühzeitigen Abschneiden von Teilbäumen (Verbesserung der Pruning-Distanz) verwendet.
  - Vor dem rekursiven Abstieg: sortieren der Kindseiten nach MINDIST (experimentell als bestes Prioritätsmaß ermittelt)

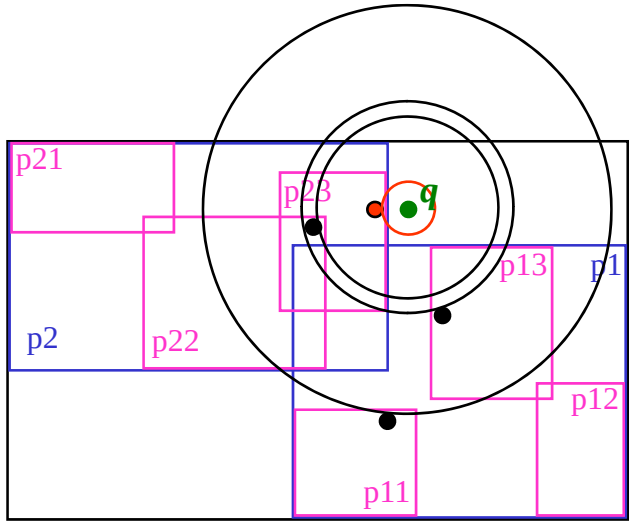
- Algorithmus:

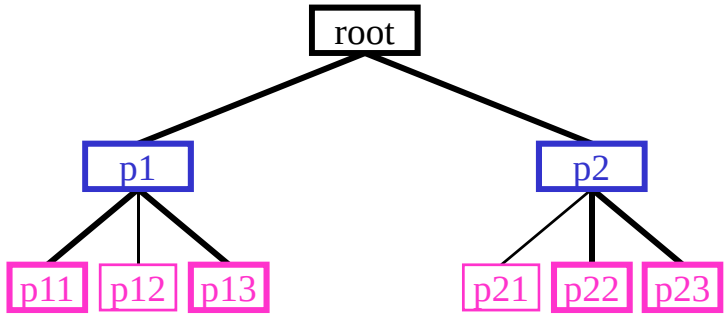
Globale Variablen: pruningdist = +∞;

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

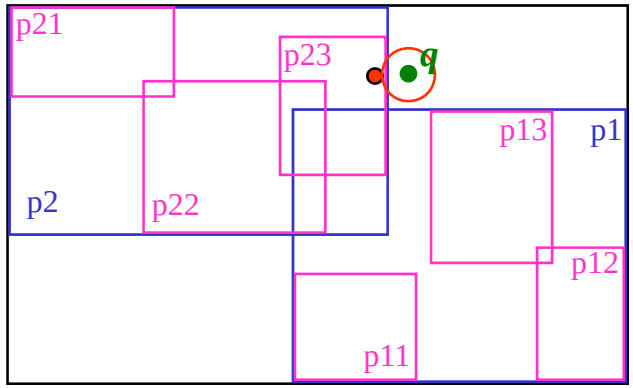
```
k-NN-Index-RKV (pa, q, k)          // pa = Diskadress z.B. der Wurzel des Indexes
  p := pa.loadPage();
  IF p.isDataPage() THEN
    FOR i=0 TO p.size() DO
      IF dist(q, p.getObject(i)) ≤ pruningdist THEN
        result .insert((getObject(i));
        pruningdist = dist(q, result.first);
  ELSE                              // p ist Directoryseite
    FOR i=0 TO p.size() DO    // Vorausgesetzt: k << # Objekte in
                                      // den Teilbäumen unter p
      IF MAXDIST(q, p.getRegion(i)) < pruningdist THEN
        pruningdist = MAXDIST(q, p.getRegion(i));
    quicksort(p.getObjectArray(), MINDIST);
    FOR i=0 TO p.size() DO
      IF MINDIST(q, p.getRegion(i)) ≤ pruningdist THEN
        k-NN-Index-RKV(p.childPage(i), q, k);
  END IF;
```

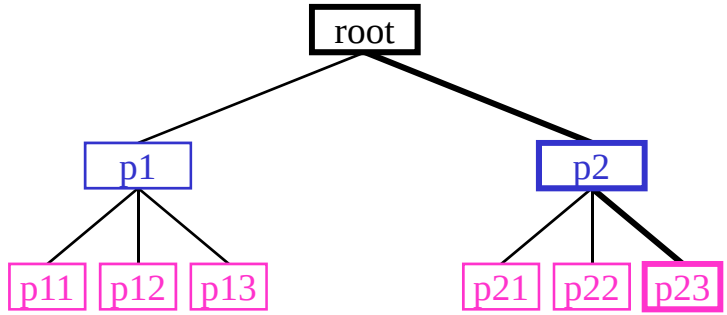
- Ablaufbeispiel
  - Einfache Tiefensuche



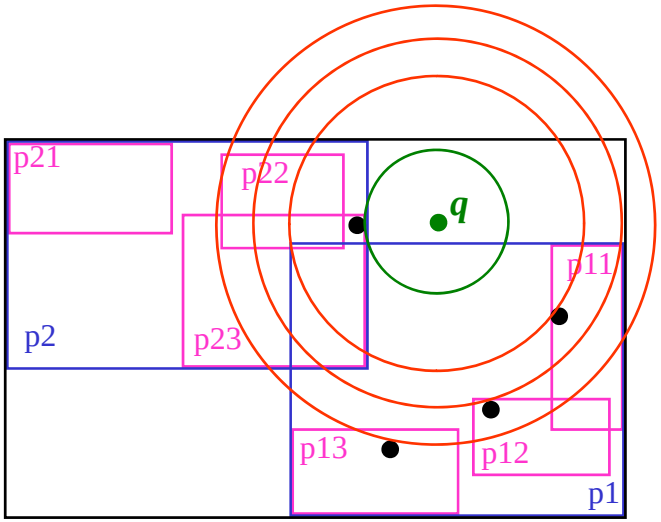


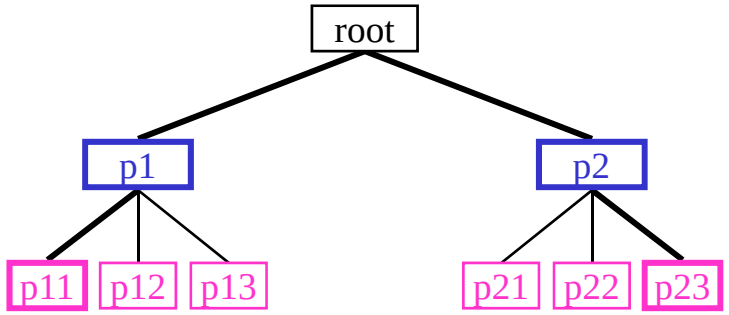
  - Tiefensuche nach RKV





- Fazit:
  - Priorisierung mit MINDIST bewirkt Reduktion der Seitenzugriffe von 7 auf 3
  - MAXDIST kann grundsätzlich die Pruning-Distanz verbessern, verhindert hier aber keine Seitenzugriffe
  - Trotz Priorisierung: Tiefendurchlauf kann prinzipiell stark fehlgeleitet werden wenn z.B. eine Seite auf dem ersten Level sehr nah am Queryobjekt liegt, ihre Kindseiten aber relativ weit weg





  - Bei Start mit p_2 hätte keine der Kindseiten von p_1 geladen werden müssen

– Priorität-basierte Suche (Best-first search nach [HS 95]

[Hjaltason, Samet. Proc. Int. Symp. on Large Spatial Databases (SSD), 1995]

- Statt rekursivem Durchlauf: Liste der aktiven Seiten (active page list APL)
  - Seite p ist aktiv genau dann wenn folgende Bedingungen erfüllt sind:
    - » p wurde noch nicht geladen
    - » Elternseite von p wurde bereits geladen
    - » MINDIST(q, p.getRegion()) ≤ pruningdist
  - APL wird mit Wurzel des Indexes initialisiert
  - Seiten in APL nach MINDIST zum Anfrageobjekt aufsteigend sortiert
  - Algorithmus entnimmt immer die erste Seite aus APL (mit kleinster MINDIST)
  - Entnommene Seite wird geladen und verarbeitet: („verfeinert“)
    - » Datenseiten werden wie bisher verarbeitet
    - » Directoryseiten: Kindseiten mit MINDIST ≤ pruningdist in APL einfügen
  - Ändert sich pruningdist werden Seiten mit MINDIST > pruningdist alternativ:
    - » aus APL entfernt
    - » als gelöscht markiert
    - » ohne explizite Markierung später ignoriert

- Algorithmus:

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).

k-NN-Index-HS(pa, *q*) // pa = Diskadress z.B. der Wurzel des Indexes

apl = **LIST OF** (dist:Real, da:DiskAdress) **ORDERED BY** dist **ASCENDING**

apl = [(0.0, pa)]

WHILE NOT apl.isEmpty() **AND** apl.first().dist ≤ pruningdist **DO**

p := apl.pop_first_element;

IF p.isDataPage() **THEN …**

* siehe k-NN-Index-RKV *

ELSE // p ist Directoryseite

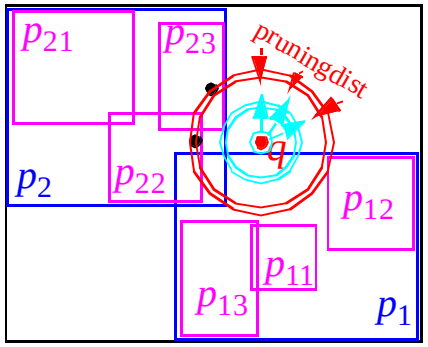
FOR *i*=0 **TO** p.size() **DO**

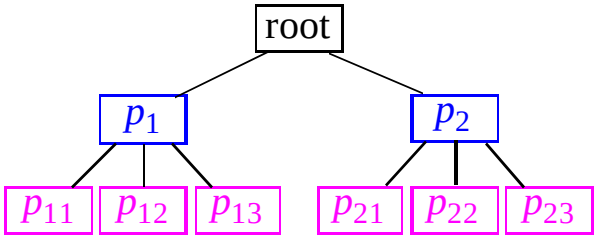
IF MINDIST(*q*, p.getRegion(*i*)) ≤ pruningdist **THEN**

apl.insert(MINDIST(*q*, p.getRegion(*i*)), p.childPage(*i*));

END IF;

- Beispiel





APL:

1. root
2. p_1 p_2
3. p_2 p_{12} p_{13} p_{11}
4. p_{23} p_{22} p_{12} p_{13} p_{11} p_{21}
5. p_{22} p_{12} p_{13}

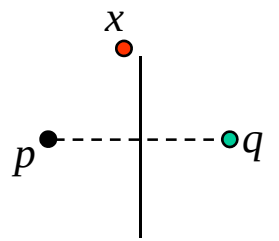
- Eigenschaften
  - Allgemein
    - » Seiten werden nach aufsteigendem Abstand geordnet zugegriffen (blaue Kreise)
    - » pruningdist wird kleiner, sobald nähergelegenes Objekt gefunden (rote Kreise)
    - » Anfragebearbeitung stoppt, wenn beide Kreise sich treffen. Optimalität!!!
  - Speicherbedarf
    - » Wie bei Breitensuche kann gesamter unterste Directorylevel in APL stehen
    - » Dieser Fall is allerdings unwahrscheinlicher als bei Breitensuche
    - » Speicherkomplexität O(n) (Tiefensuche O(log n)

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 $x \notin \text{RNN}(q)$.



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, konstruiere neue Voronoi-Ebene $E(p,q)$; p wird zur Kandidatenmenge hinzugefügt
    - » Punkte/Directoryseiten, die „hinter“ einer der Voronoi-Ebenen liegen (außer der eigenen), können aus dem Ranking/Kandidatenmenge gelöscht werden
  - Punkte, die die Ebenen bestimmen, müssen verfeinert werden, d.h. für diese Punkte muss jeweils eine kNN-Anfrage berechnet werden

– 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.

– Algorithmus (Filter):

Algorithm TPL-filter(q) /* q is the query point */
1. initialize a min-heap H accepting entries of the form (e, key)
2. initialize sets $S_{cnd}=\emptyset$, $S_{rfn}=\emptyset$
3. insert (R-tree root, 0) to H
4. while H is not empty
5. (e, key)=de-heap H
6. if (**trim**(q, S_{cnd}, e)=∞) then $S_{rfn}=S_{rfn}\cup\{e\}$
7. else // entry may be or contain a candidate
8. if e is data point p
9. $S_{cnd}=S_{cnd}\cup\{p\}$
10. else if e points to a leaf node N
11. for each point p in N (sorted on $dist(p,q)$)
12. if (**trim**(q,S_{cand},p)$\neq\infty$) then insert (p,$dist(p,q)$) in H
13. else $S_{rfn}=S_{rfn}\cup\{p\}$
14. else // e points to an intermediate node N
15. for each entry N_i in N
16. $mindist(N_i^{resM}, q)$=**trim**(q, S_{cnd}, N_i)
17. if ($mindist(N_i^{resM}, q)=\infty$) then $S_{rfn}=S_{rfn}\cup\{N_i\}$
18. else insert (N_i, $mindist(N_i^{resM}, q)$) in H

End TPL-filter

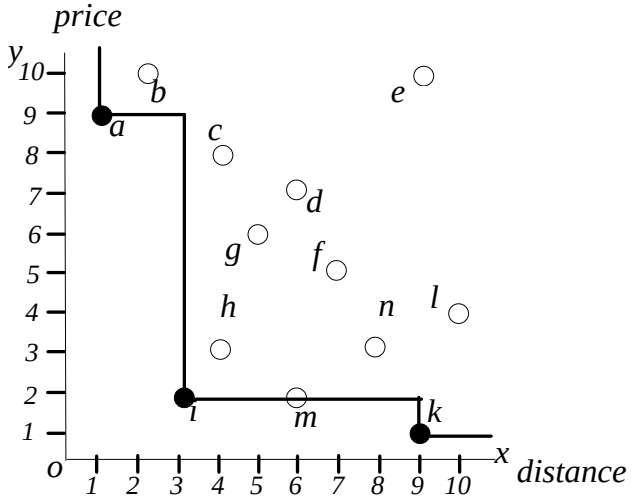
falls e hinter einer bestehenden Pruningebene

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:

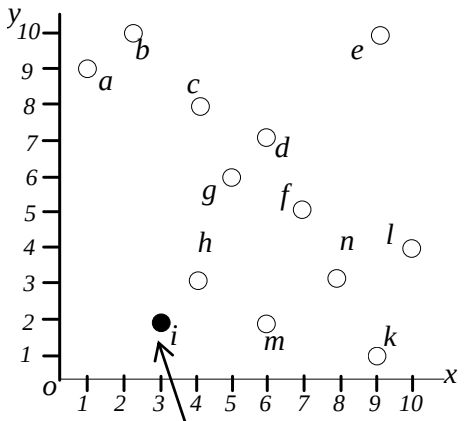
      - wenig Seitenzugriffe und Dominanzüberprüfung mittels Index (R-Baum).
      - Erste Ergebnisse werden frühzeitig ausgegeben durch iterative Verarbeitung.

– Nächste-Nachbarn-Skyline (NNS) Algorithmus:

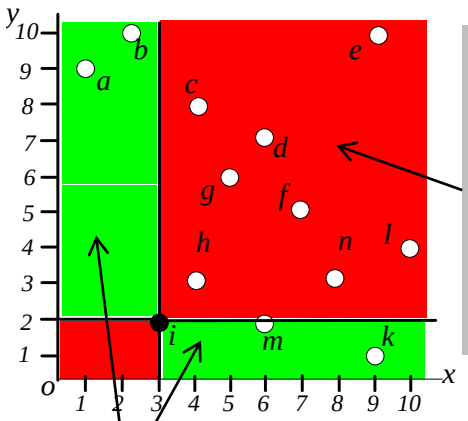
[Kossman et al., VLDB 2002]

Prinzip:

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



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

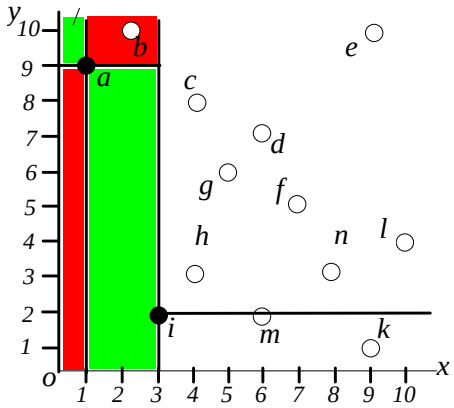


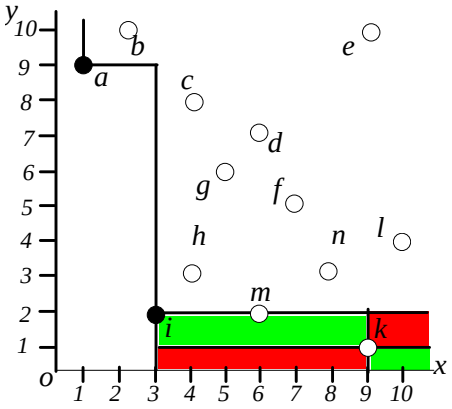
Raumpartitionen mit
weiteren Skyline-Kandidaten

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

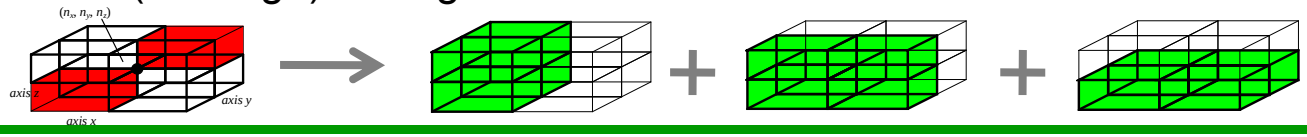
- Nächste-Nachbar-Suche kann durch R-Baum Index beschleunigt werden (z.B. Alg.: k-NN-Index-HS, siehe Folie 63).

- 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:

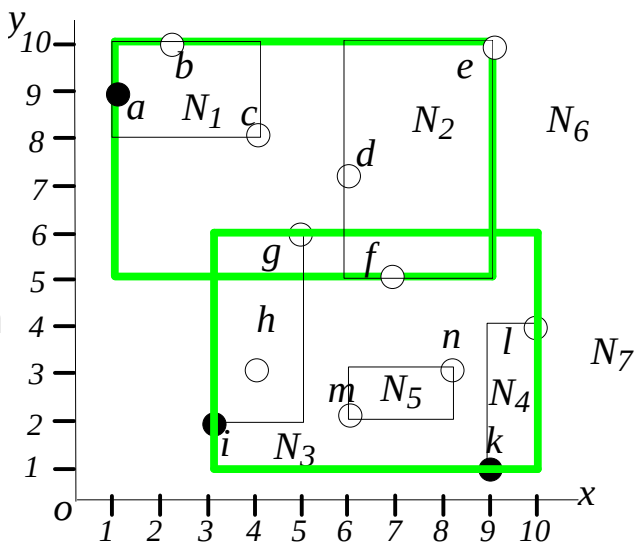
[Papadias et al., ToDS 2005]

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

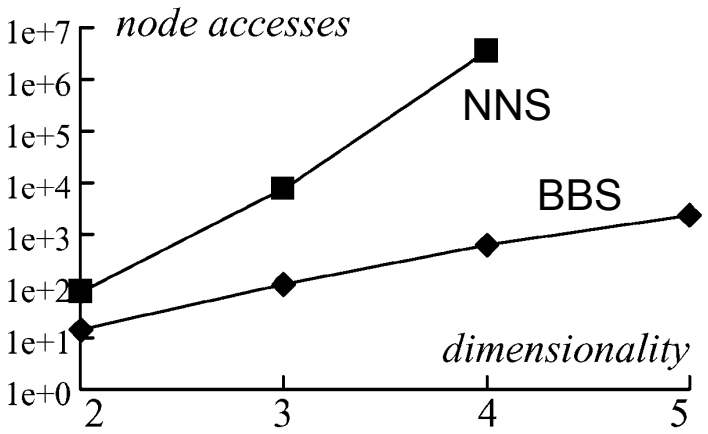


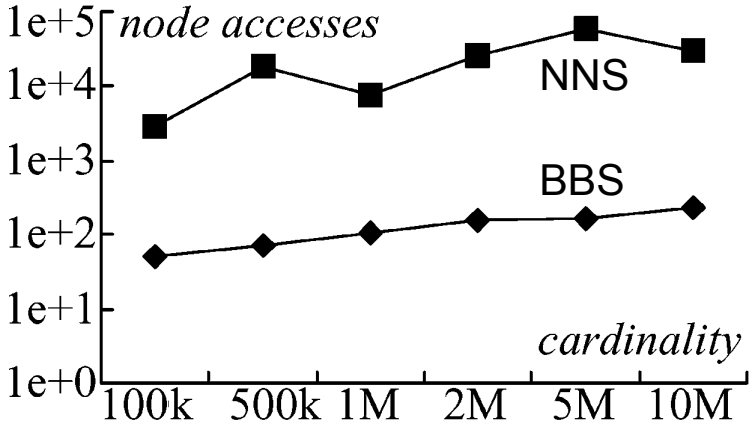
- Algorithmus:

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



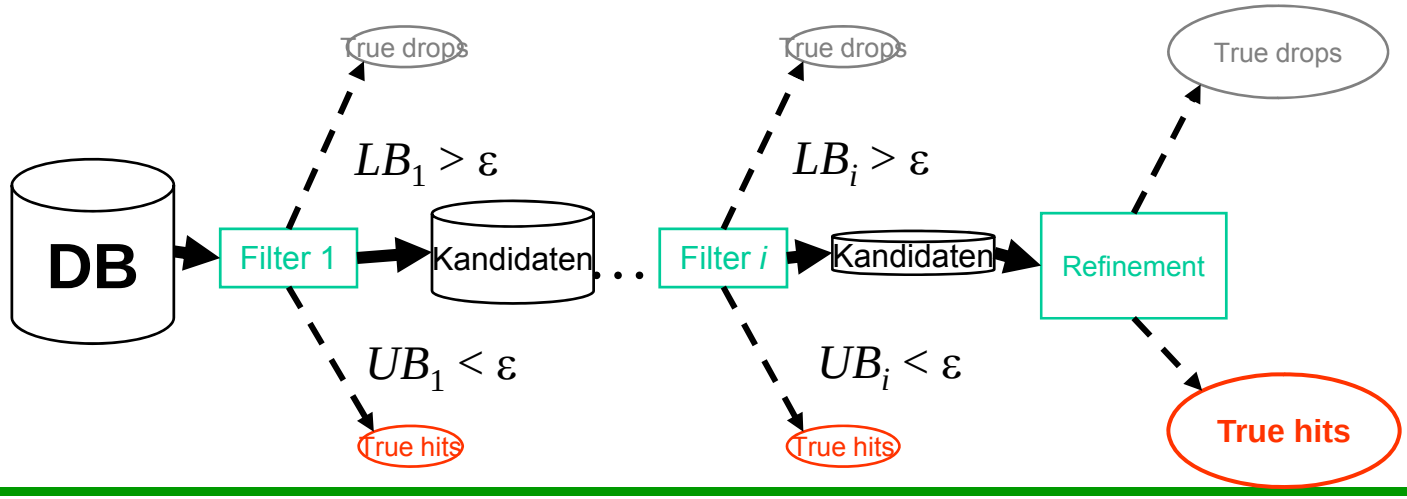


  - BBS schlägt NNS bzgl. I/O-Kosten über mehrere Größenordnungen

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

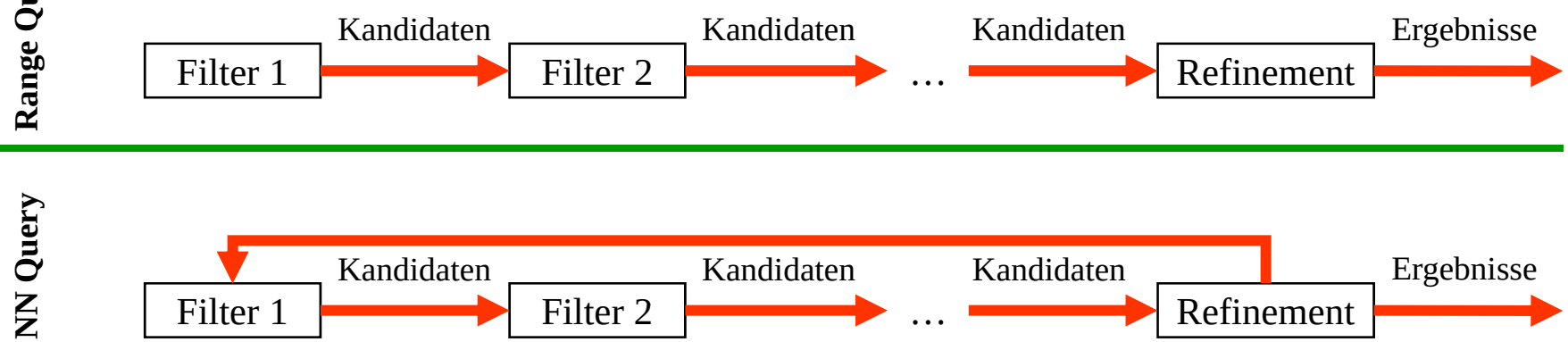


- Mehrstufige-Bereichsanfrage Algorithmus: (Filter-/Refinement)
  - Lower Bounding Filterdistanz $dist_{LB}$
  - Upper Bounding Filterdistanz $dist_{UB}$

```
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

  } siehe STMD I Skript

- 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;