Kapitel 2
Prinzipien der Anfragebearbeitung in STMM-DBS

Skript zur Vorlesung: Spatial, Temporal, and Multimedia Databases
Sommersemester 2014, LMU München

Übersicht

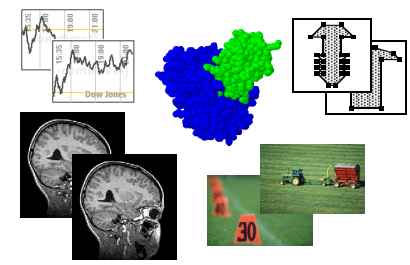
2.1 Feature-Räume

2.1.1 Das Prinzip der feature-basierten Ähnlichkeit

– Grundidee der Feature-Transformation

- **Erwünscht**: effiziente Ähnlichkeitssuche in Datenbanken
- Meist ist Effizienz ohne Einsatz von Indexstrukturen nicht zu verwirklichen
- Entwickle nicht für jedes der einzelnen Anwendungsgebiete/ Ähnlichkeitsmaße spezielle Indexstrukturen
- Versuche mit wenigen Arten von Indexstrukturen möglichst viele Anwendungsgebiete abzudecken
- Indexstrukturen für:
  - Multidimensionale Vektoren
  - Allgemein metrische Daten (beliebige Objekte, auf denen eine metrische Distanzfunktion definiert ist)

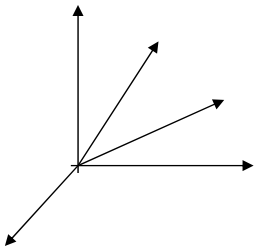
- Extrahiere charakteristische (nummerische) Eigenschaften („Features“) aus den Objekten



Anwendungsgebiete (Objektraum)

Feature-Transformation

Histogramm-Bildung
Moment-Invarianten
Rechteck-Abdeckung
Sektoren-Ermittlung
Fourier-Transformation Kurvatur
usw.



Featurevektor-Raum

- Wichtigste Eigenschaft der Feature-Transformation:
  - Ähnlichkeit der Objekte entspricht geringem Abstand der Feature-Vektoren

  => Ähnlichkeitsanfragen im Objektraum entsprechen Nachbarschaftsanfragen im Feature-Raum

  => Unterstützung durch geeignete multidimensionale Indexstrukturen

– Erweiterungen
  - Oft reichen die Mächtigkeit von Vektoren für die Modellierung nicht aus (vergleiche z.B. Relationales und OO-Modell)
  - Transformation in andere Räume, die die Definition einer Distanzfunktion mit Metrik-Eigenschaften erlauben
    – Graphen
    – Punktmengen
    – …

– Fazit:
  - Das Prinzip der Feature-Transformation ist ein mächtiges Werkzeug zur Modellierung der Ähnlichkeit von komplexen STMM-Objekten
  - Herausforderung: Finden geeigneter Feature-Transformationen

2.1.2 Feature-Räume und Distanzen

– Allgemeiner Feature-Raum

Ein Feature-Raum ist ein Tupel Φ = (Dom, dist) mit
  - Dom ist ein Wertebereich (Domain)
  - dist ist eine Distanzfunktion, d.h. es gilt
    – Reflexivität $\forall x,y \in Dom: dist(x,y) = 0 \Leftrightarrow x=y$
    – Positiv-Definitheit $\forall x,y \in Dom, x \neq y: dist(x,y) > 0$
    – Symmetrie $\forall x,y \in Dom: dist(x,y) = dist(y,x)$

– Metrischer Raum

Ein metrischer Raum ist ein Tupel Φ_M = (Dom, dist) mit
  - (Dom,dist) ist ein allgemeiner Feature-Raum
  - dist erfüllt zusätzlich die
    – Dreiecksungleichung $\forall x,y,z \in Dom: dist(x,z) \leq dist(x,y) + dist(y,z)$

– Euklidischer Vektorraum

Ein (euklidischer) Vektorraum der Dimension *d* (*d*-dimensionaler Vektorraum) ist ein Tupel Φ_E = (Dom, dist) mit

- (Dom,dist) ist ein metrischer Raum
- Dom = $\mathbb{R}^d$

– Feature-Transformation

Eine Feature-Transformation ist eine Abbildung

T: OBJ $\rightarrow$ (Dom,dist)

die jedem Objekt o$\in$OBJ aus dem Objektraum ein Objekt aus dem Wertebereich Dom zuordnet.

Die Distanz im Objektraum wird durch die Distanz im Feature-Raum repräsentiert, d.h.

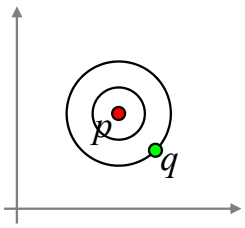
$\forall x,y \in OBJ: dist_{OBJ}(x,y) \equiv dist_{Dom}(T(x),T(y))$

– Distanzmaße in Vektorräumen

- Euklidische Norm (L_2):

  $dist = ((p_1-q_1)^2+(p_2-q_2)^2+...)^{1/2}$

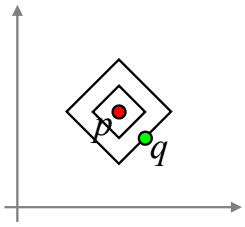
  Natürliches Distanzmaß



- Manhattan-Norm (L_1):

  $dist = |p_1-q_1|+|p_2-q_2|+...$

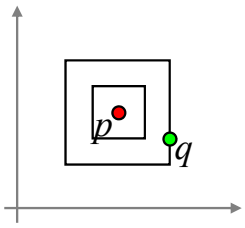
  Die Unähnlichkeiten der einzelnen Merkmale werden direkt addiert



- Maximums-Norm (L_∞):

  $dist = \max\{|p_1-q_1|, |p_2-q_2|,...\}$

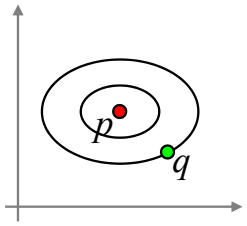
  Die Unähnlichkeit des am wenigsten ähnlichen Merkmals zählt



- Verallgemeinerung L_p-Abstand: $dist_p = (|p_1-q_1|^p + |p_2-q_2|^p + ...)^{1/p}$
- Gewichtete Euklidische Norm:

  $dist = (w_1(p_1-q_1)^2+w_2(p_2-q_2)^2+...)^{1/2}$

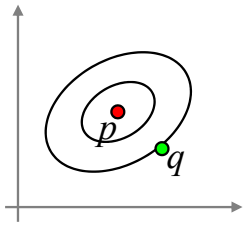
  Häufig sind die Wertebereiche der Merkmale deutliche unterschiedlich
  Beispiel: Merkmal $M_1 \in [0.01 \ldots 0.05]$
  Merkmal $M_2 \in [3.07 \ldots 22.2]$
  Damit M1 überhaupt berücksichtigt wird muss es höher gewichtet werden



- Quadratische Form:

  $dist = ((p - q)\ \mathbf{M}\ (p - q)^T)^{1/2}$

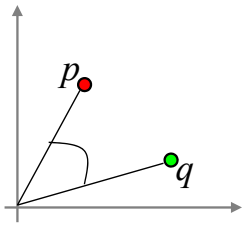
  Bisherige Abstandsmasse gewichten Merkmale nur getrennt
  Besonders bei Farbhistogrammen müssen verschiedene Merkmale gemeinsam gewichtet werden



- Cosinus-Distanz

  $dist$ = cos(winkel(p,q))

  Berechnet den cosinus des Winkels
  Meist für sehr hochdimensionalen Featurevektoren (z.B. Texten)



– Bemerkungen

- Jeder Vektorraum ist ein metrischer Raum, jeder metrische Raum ein allgemeiner Feature-Raum
- Sprechweise meist: „Feature-Raum“ statt (euklidischer) Vektorraum
- Transformation komplexer Objekte meist immer in metrische Räume wegen der Dreiecksungleichung (Performanz!!!)

2.2 Algorithmische Paradigmen zur Anfragebearbeitung

2.2.1 Übersicht

- Typen von Ähnlichkeitsanfragen
  - Bereichsanfragen
  - Nächste Nachbarn Anfragen
  - Reverse Nächste Nachbarn
  - Skyline Anfragen
- Algorithmische Paradigmen
  - Naive (sequentielle) Suche
    - Für alle *n* Objekte der Datenbank wird das Anfrage-Prädikat (d.h. meist eine Distanzberechnung zum Anfrageobjekt) ausgewertet
    - Kosten: O(*n* · Kosten für das Anfrage-Prädikat)
  - Indexbasierte Suche
  - Mehrstufige Anfragebearbeitung

2.2.2 Indexstrukturen

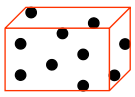
- Prinzip [Böhm, Berchtold, Keim. ACM Computing Surveys, 2001]
  - Organisiere Objekte der Datenbank so, dass während einer Ähnlichkeitsanfrage nur auf „relevante“ Objekte zugegriffen werden muss
  - Baumartige Organisation; jedem Knoten des Baumes ist zugeordnet
    - Seite des Hintergrundspeichers
    - Region des Datenraums
  - Typen von Knoten (= Seiten)
    - Blattknoten sind Datenseiten, speichern Objekte
    - Innere Knoten sind Directoryseiten, speichern Directory-Einträge
      - » Verweis zur Kindseite (Adresse auf dem Hintergrundspeicher)
      - » Beschreibung der Region der Kindseite

- Physische vs. logische Seiten
  - ursprünglich: eine physische Seite des Hintergrundspeichers wird verwendet
    - » kleinste Informationseinheit, die zwischen Plattenspeicher und RAM übertragen werden kann
    - » ABER: physische Seiten meist zu klein
  - daher: logische Seiten fassen aufeinanderfolgende physische Seiten zusammen
  - Meistens: einheitliche Seitengröße für alle Seiten eines Indexes (um komplizierte Freispeicherverwaltung zu vermeiden)
  - Kapazität der Directory-/Datenseiten (max. Anzahl der Einträge)

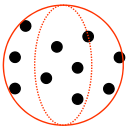
$$c_{\text{Data}} = \left\lfloor \frac{\text{Seitengröße} - \text{Verwaltungsoverhead}}{\text{Größe eines Datensatzes}} \right\rfloor \qquad c_{\text{Directory}} = \left\lfloor \frac{\text{Seitengröße} - \text{Verwaltungsoverhead}}{\text{Größe eines Directoryeintrags}} \right\rfloor$$

  - Meist keine 100% Füllung der Seiten (Platz für neue Datensätze) aber minimale Füllung (z.B. 40%) wegen Speicherauslastung
  - Speicherauslastung ist die durchschnittliche Anzahl besetzter Einträge
- Jedes Objekt wird in genau einer Datenseite gespeichert
- Regionen gewährleisten, dass ähnliche Objekte möglichst auf den selben Datenseiten (oder Teilbäumen) gespeichert werden

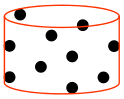
- Gestalt der Seitenregionen
  - Vektordaten:
    - » Achsenparallel Rechtecke, die minimal um die Punktmenge gespannt werden (MUR=minimal umgebendes Rechteck/ MBR=minimum bounding rectangle) (R-Tree, R*-Tree, X-Tree, …)



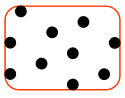
    - » Kugeln (SS-Tree)



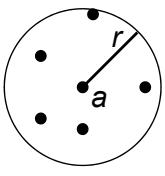
    - » Zylinder (TV-Tree)



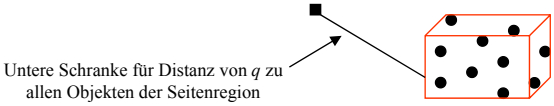
    - » Kombinationskörper (Kugel+MBR, SR-Tree)



  - Algemein metrische Daten
    - » Kein „Raum“ im Euklidischen Sinne, keine Geometrie
    - » Ankerobjekt *a* (anchor object) + Hüllradius *r* (covering radius) Für alle Objekte *o* der Seitenregion gilt: dist(*o*,*a*) ≤ *r*



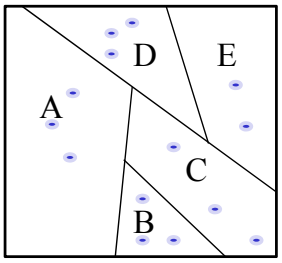
- Seitenregion umfasst immer alle Objekte der Datenseite bzw. des Teilbaums der Directoryseite
  - Konservative Approximation, lower-bounding property: die Distanz von einem Objekt zu einer Seitenregion kann immer mit einer unteren Schranke abgeschätzt werden



  - Damit kann man Vollständigkeit der Anfrageergebnisse gewährleisten
- Bäume sind meist balanciert (alle Blätter auf selbem Level)
- Indexstrukturen sind dynamisch, d.h. Insert/Delete sind effizient
  - Tiefensuche nach entsprechender Datenseite
  - Einfügen/Löschen des Objektes
  - Überlauf/Unterlauf-Behandlung durch Split/Merge
    - » Verschiedene Kriterien (minimale Überlappung, toter Raum, …)
    - » Verschiedene Algorithmen (linear, quadratisch, …)
    - » Ensprechendes Einfügen/Löschen im Elternknoten (rekursiv, notfalls bis Wurzel)

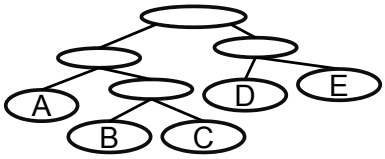
Binary Space Partitioning Tree (BSP-Tree):

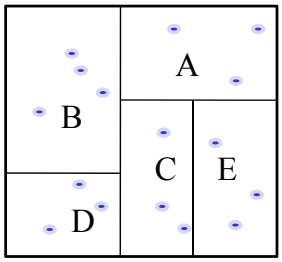
- Wurzel enthält gesamten Datenraum
- Jeder innere Knoten hat 2 Söhne
- Datenobjekte in den Blättern



Bekannteste Variante: *kD-Tree*

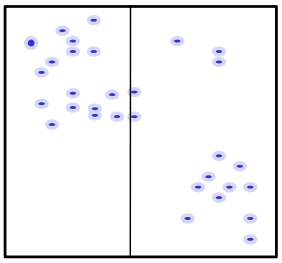
- max. Seitenkapazität sind *M* Einträge
- min. Seitenkapazität sind *M/2* Einträge
- bei Überlauf achsenparalleler Split
- nach Löschen Vereinigung von Geschwisterknoten
- Split-Achse wechselt nach jedem Split
- 50%-50% Aufteilung der Daten





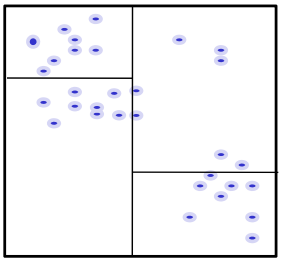
Problem

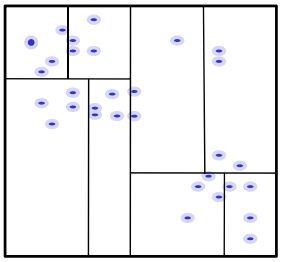
- keine Balancierung (Degeneration des Baums)
- Korrektur der Balancierung möglich aber aufwendig
- => Hohe Update-Komplexität



Bulk-Load

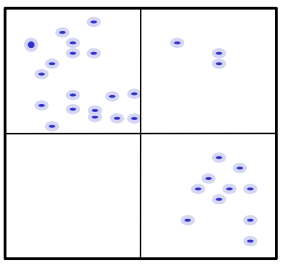
- Annahme: Kenntnis aller Datenobjekte
- Aufbau: durch rekursive 50-50 Aufteilung der Objekte bis jedes Blatt weniger als *M* Objekte enthält
- Bulk-Load erzeugt immer einen balancierten Baum
- Datenseite eine Baums mit *n* Objekten und Höhe *h* enthält mindestens $\left\lfloor \frac{n}{2^h} \right\rfloor$ Objekte und höchstens $\left\lfloor \frac{n}{2^h} \right\rfloor + 1$

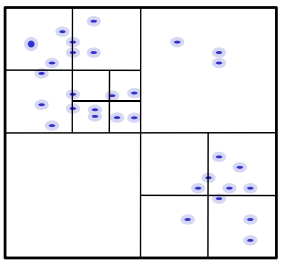


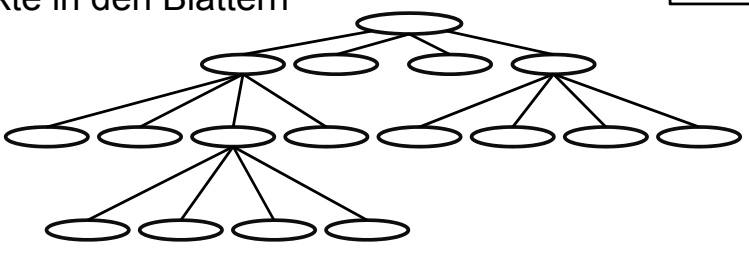


QuadTree für 2D Daten:

- Wurzel stellt den ganzen Datenraum dar
- Jeder innere Knoten hat 4 Nachfolger
- Geschwisterknoten teilen den Raum ihres Elternknotens in 4 gleich große Teile ein
- Quad-Trees sind i.d.R. nicht balanciert
- Seiten haben einen max. Füllungsgrad *M,* aber keine Mindestfüllung
- Datenobjekte in den Blättern

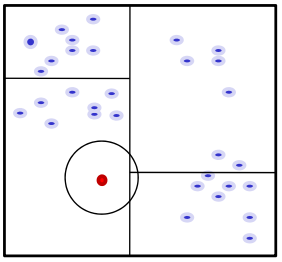






Raumpartitionierende Verfahren:

- Aufteilung des gesamten Datenraums durch Splits in den Dimensionen
- Seitenregionen enthalten toten Raum
  => evtl. schlechtere Suchperformanz bei räumlichen Anfragen



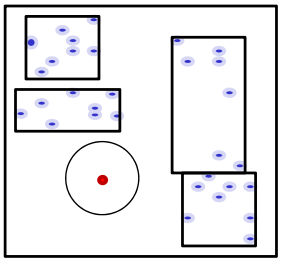
Range-Query in BSP-Tree

Datenpartitionierende Verfahren:

- Beschreibung der Seiten-Region durch minimal umgebende Regionen (z.B. Rechtecke)

  => Bessere Pruning Leistung
- Seitenregionen können überlappen

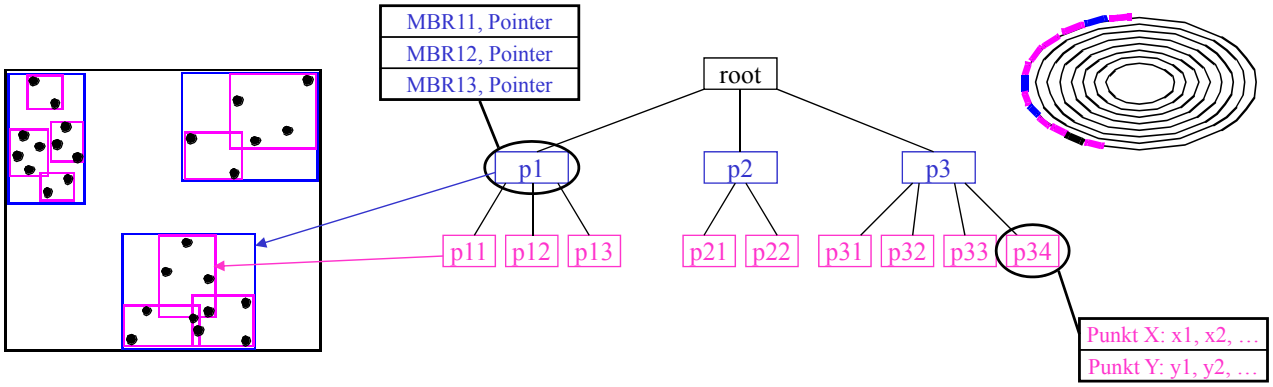
  => Degeneration bzgl. Überlappung
- Split- und Einfüge-Algorithmen minimieren:
  - Überlappung der Seitenregionen
  - Toten Raum in den Seiten
  - Balancierung bzgl. des Füllungsgrades



Range-Query in R-Tree

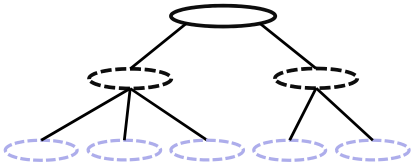
R-tree und R*-tree

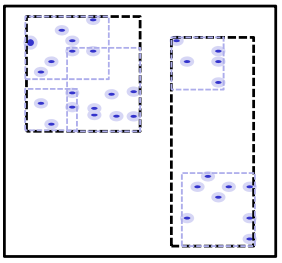
- Entworfen für 2D Rechtecksdaten (Punkte sind spezielle Rechtecke), oft verwendet für hochdimensionale Vektordaten
- Design
  - Balancierter Index mit einheitlich großen Daten-/Directoryseiten
  - Seitenregionen: MBRs
  - Insert-Strategie: Overlap, Volumen
  - Überlaufbehandlung: Forced Re-Insert-Konzept
  - Splitkriterien: Umfang/Oberfläche, Überlappungsvolumen, (toter Raum)



Struktur eines R-Baums:

- Wurzel umgibt den gesamten Datenraum und hat maximal *M* Einträge
- Seitenregionen werden durch minimal umgebende Rechtecke (MUR) modelliert
- innere Knoten im R-Baum haben zwischen *m* und *M* Nachfolger (wobei m ≤ M/2)
- Das MUR eines Nachfolgers ist vollständig im MUR des Vorgängers enthalten
- Alle Blätter sind auf dem gleichen Level
- Datenobjekte werden in den Blättern gespeichert
- Mögliche Datenobjekte:
  - Punkte
  - Rechtecke





Das Objekt x ist in einen R-Baum einzufügen

Durch Überlappung können 3 Fälle auftreten:

Fall 1: x fällt in genau ein Directory-Rechteck D
⇒ Einfügen in Teilbaum von D

Fall 2: x fällt in mehrere Directory-Rechtecke $D_1, \ldots, D_n$
⇒ Einfügen in Teilbaum von D_i, das die geringste Fläche aufweist

Fall 3: x fällt in kein Directory-Rechteck
⇒ Einfügen in Teilbaum von *D*, das den geringsten Flächenzuwachs erfährt
(in Zweifelsfällen, das die geringste Fläche hat)
⇒ *D* muss entsprechend vergrößert werden

(im Folgenden wird von inneren Knoten ausgegangen: Objekte sind MURs)

Der Knoten K läuft mit |K| = M+1 über:

⇒ Aufteilung auf zwei Knoten K_1 und K_2, so dass $|K_1| \geq m$ und $|K_2| \geq m$

Quadratischer Algorithmus

1. Wähle das Paar von Rechtecken R_1 und R_2 mit dem größten Wert für den „toten Raum" im MUR, falls R_1 und R_2 in denselben Knoten K_i kämen.

   d (R_1, R_2) := Fläche(MUR($R_1 \cup R_2$)) - Fläche(R_1) - Fläche(R_2)

2. Setze K_1 := {R_1} und K_2 := {R_2}
3. Wiederhole den folgenden Schritt bis zum STOP:
   - wenn alle R_i zugeteilt sind: STOP
   - wenn alle restlichen R_i benötigt werden, um den kleineren Knoten minimal zufüllen: teile sie alle zu und STOP
   - sonst: wähle das nächste R_i und teile es dem Knoten zu, dessen MUR den kleineren Flächenzuwachs erfährt. Im Zweifelsfall bevorzuge den K_i mit kleinerer Fläche des MUR bzw. mit weniger Einträgen

Linearer Algorithmus

- Der lineare Algorithmus ist identisch mit dem quadratischen Algorithmus bis auf die Auswahl des initialen Paars (R_1, R_2).
- Wähle das Paar von Rechtecken R_1 und R_2 mit dem „größten Abstand", genauer**:**
  - Suche für jede Dimension das Rechteck mit dem kleinsten Maximalwert und das Rechteck mit dem größten Minimalwert (*maximaler Abstand*).
  - Normalisiere den *maximalen Abstand* jeder Dimension, indem er durch die Summe der Ausdehnungen der R_i in der Dimension dividiert wird (*setze den maximalen Abstand der Rechtecke ins Verhältnis zur ihrer Ausdehnung*).
  - Wähle das Paar von Rechtecken mit dem größten normalisierten Abstand bzgl. aller Dimensionen. Setze K_1 := {R_1} und K_2 := {R_2}.

Dieser Algorithmus ist linear in der Zahl der Rechtecke (*2m+1)*
und in der Zahl der Dimensionen *d*.

Idee der R*-Baum Splitstrategie

sortiere die Rechtecke in jeder Dimension nach beiden Eckpunkten und betrachte nur Teilmengen nach dieser Ordnung benachbarter Rechtecke

Laufzeitkomplexität ist O($d \cdot M \cdot log\ M$) für d Dimensionen und M Rechtecke

Bestimmung der Splitdimension

- Sortiere für jede Dimension die Rechtecke gemäß beider Extremwerte
- Für jede Dimension:
  - Für jede der beiden Sortierungen werden $M-2m+2$ Aufteilungen der $M+1$ Rechtecke bestimmt, so dass die 1. Gruppe der j-ten Aufteilung die ersten $m-1+j$ Rechtecke und die 2. Gruppe die übrigen Rechtecke enthält
  - U_G sei die Summe aus dem Umfang der beiden MURs R_1 und R_2 um die Rechtecke der beiden Gruppen
  - U_S sei die Summe der U_G aller berechneten Aufteilungen

  ⇒ Es wird die Dimension mit dem geringsten U_S als Splitdimension gewählt.

Das Objekt x ist aus dem R-Baum zu löschen

Löschen :

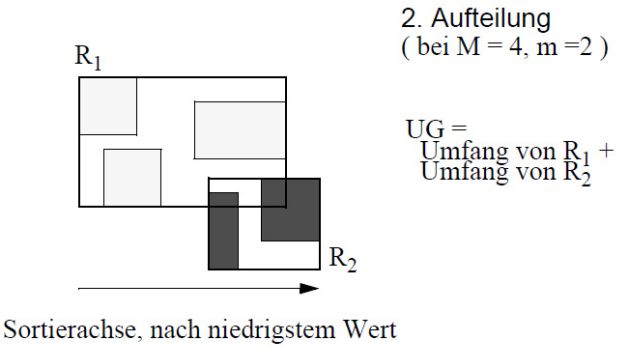
- Teste ob Seite S nach entfernen von x unterfüllt ist: |S| < m
- Falls nicht, entferne x und STOP
- Falls ja bestimme, welche Vorgängerknoten ebenfalls unterfüllt sind
- Für jeden unterfüllten Knoten:
  - Lösche die unterfüllte Seite aus dem Vorgängerknoten
  - Füge die restlichen Elemente der Seite in den R-Baum ein
  - Falls die Wurzel nur noch 1 Kind enthält wird Kind zur neuen Wurzel (Höhe verringert sich)

Bemerkungen:

Löschen ist mit diesem Algorithmus nicht auf einen Pfad beschränkt

Erfordert das Einfügen eines Teilbaums auf Ebene l in den R-Baum

Im Worst Case sehr teuer

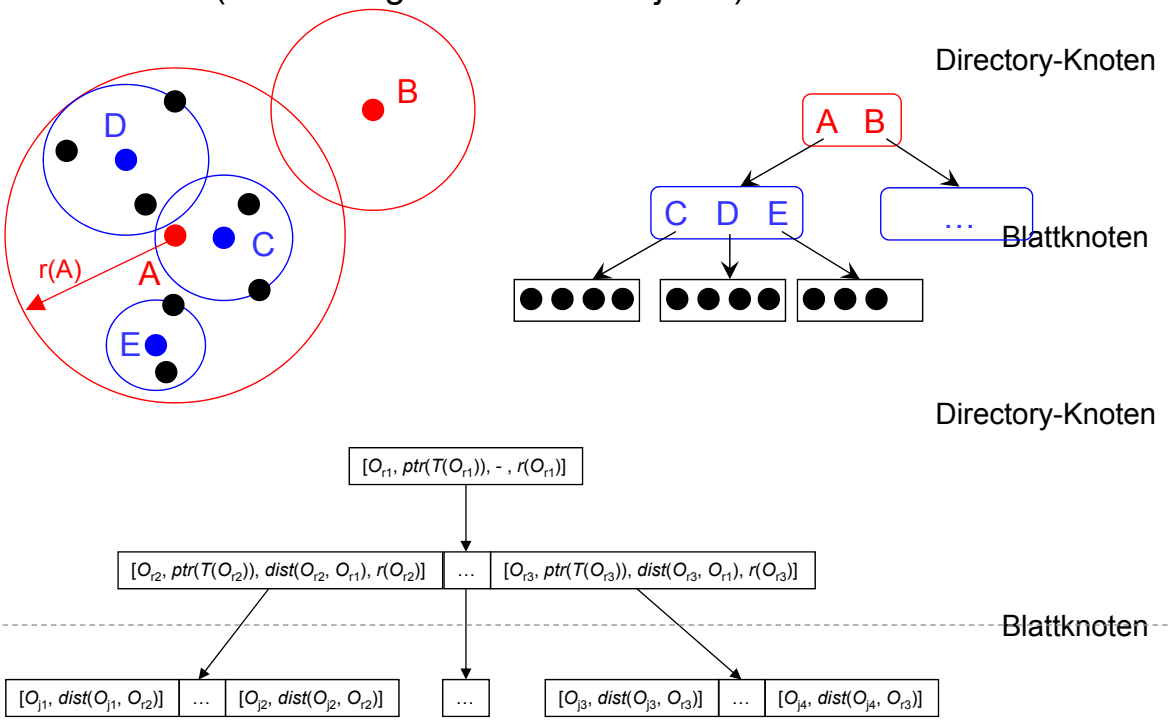


Bestimmung der Aufteilung

- Es wird die Aufteilung der gewählten Splitdimension genommen, bei der R_1 *und* R_2 die geringste Überlappung haben.
- In Zweifelsfällen wird die Aufteilung genommen, bei der R_1 und R_2 die geringste Überdeckung von toten Raum besitzen.

  ⇒ Die besten Resultate hat bei Experimenten $m = 0{,}4 \cdot M$ ergeben.

M-tree (für beliebige Metrische Objekte)



M-tree

- Dynamische Indexstruktur für allgemeine metrische Räume
- Die Distanzfunktion zur Berechung der Ähnlichkeit zweier Objekte muss die Eigenschaften einer Metrik erfüllen
- Design
  - Balancierter Index mit einheitlich großen Daten-/Directoryseiten
  - Die indexierten Datenbank-Objekte werden in den Blattknoten abgespeichert
  - Die Directory-Knoten enthalten sog. *Routing Objekte*
  - Routing Objekte entsprechen Databank-Objekten, denen eine *Routing Rolle* zugewiesen wurde
  - Wenn ein Knoten überläuft und geplittet werden muß, vergibt der Splitalgoritmus eine Routing Rolle an ein Objekt
  - Zusätzlich zur Objektbeschreibung enthält ein Routing Objekt einen Zeiger auf seinen zugehörigen Unterbaum und den Radius, in dem sich alle Objekte des Unterbaums befinden
  - Wahl der Routing Objekte: die beiden am weitesten voneinander entfernt liegenden Objekte der übergelaufenen Seite

2.2.3 Mehrstufige Anfragebearbeitung

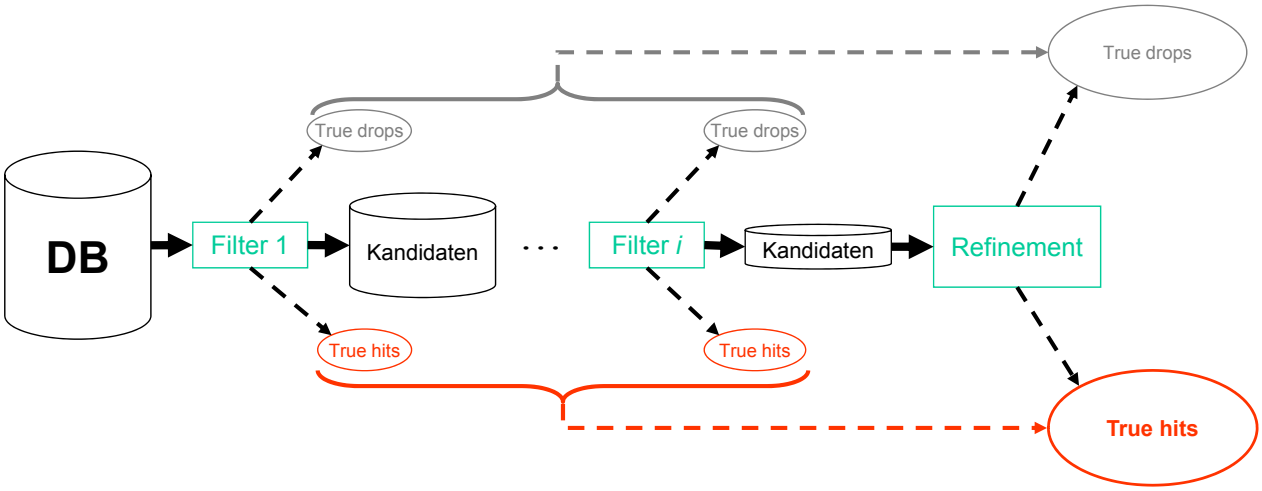
– Idee:

- Verwendetes Distanzmaß ist sehr teuer (z.B. Edit-Distanz) oder nicht feature-basiert (z.B. Überlappungsfläche von Polygonen)
- Vektorraum sehr hochdimensional (Curse of Dimensionality)
- Benutze ein feature-basiertes (meist niedrig-dimensionaler Vektorraum) Distanzmaß als Filterschritt (Filterdistanz)
  - Filterdistanz sollte billiger sein als exakte Distanz (entsprechend niedrig-dimensional => Dimensionsreduktion?)
  - Werte Anfrage-Prädikat (Distanzberechnung) mit Filterdistanz aus
  - Ergebnisse sind noch keine exakten Treffer sondern Kandidaten
  - Kandidatenmenge sollte möglichst klein sein (Filterselektivität)
  - Filterselektivität

$$\sigma_F = \frac{\#\text{Kandidaten}}{n}$$

  - Verfeinerung: für die Kandidaten wird das eigentliche Distanzmaß berechnet, was i.A. teurer dafür selektiver als der Filterschritt ist

– Mehrstufige Anfragebearbeitung:
  - Ein oder mehrere (kaskadierende) Filterschritte schränken die Kandidatenmenge sukzessive ein
  - Verfeinerungschritt testet auf Korrektheit der Kandidaten



– Zusammenpassen von Filter und Refinement
  - Idealfall: Filterdistanz ist obere oder unter Schranke (upper/ lower bound) der exakten Distanz => es kann garantiert werden, dass keine exakten Treffer verloren gehen (no false dismissals/drops)
  - Sonst: Ergebnisse u.U. nicht vollständig!!!
  - Lower Bounding Filter F_{LB}

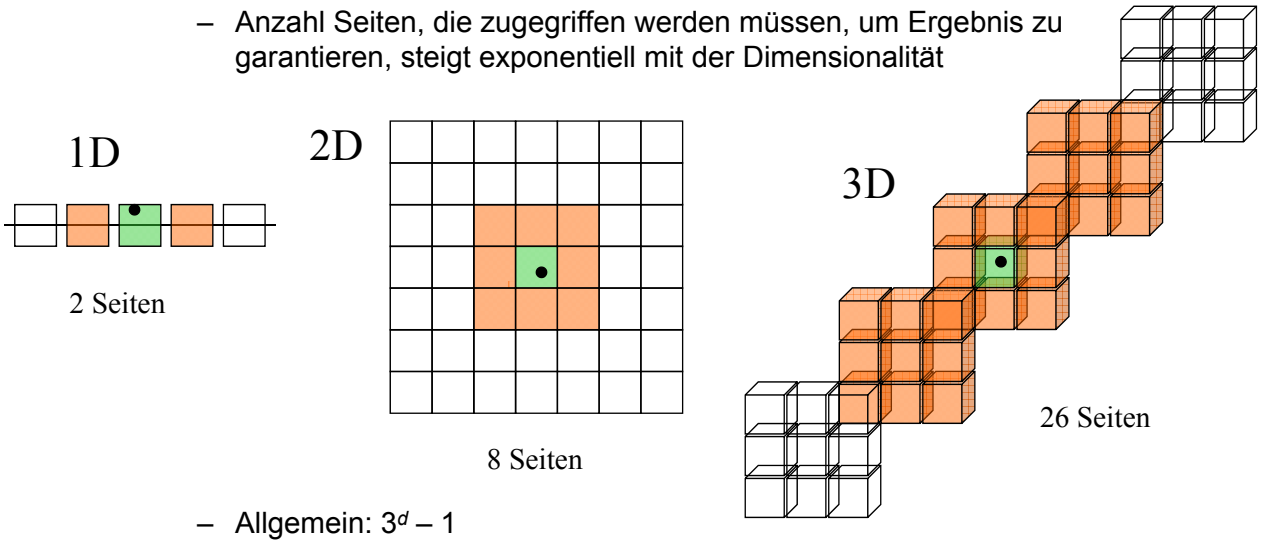
$$\forall x, y \in DB : dist_{F_{LB}}(F_{LB}(x), F_{LB}(y)) \leq dist(x, y)$$

    – Konservative Approximation (d.h. Obermenge) der Ergebnismenge
    – Objekte können evtl. bereits aufgrund der Filterdistanz als exakte Fehltreffer (true drops) identifiziert werden
  - Upper Bounding Filter F_{UB}

$$\forall x, y \in DB : dist_{F_{UB}}(F_{UB}(x), F_{UB}(y)) \geq dist(x, y)$$

    – Progressive Approximation (d.h. Untermenge) der Ergebnismenge
    – Objekte können evtl. bereits aufgrund der Filterdistanz als exakte Treffer (true hits) identifiziert werden

- Curse of Dimensionality (Vektordaten)
  - Anfrageleistung von Indexstrukturen verschlechtern sich mit zunehmender Dimension
    - Anzahl Seiten, die zugegriffen werden müssen, um Ergebnis zu garantieren, steigt exponentiell mit der Dimensionalität



    - Allgemein: $3^d - 1$
    - Seitenregionen überlappen sich stärker
    - Folge: oft muss komplettes Directory gelesen werden

2.3 Bereichsanfragen

- Allgemeines
  - Eigenschaften
    - Benutzer gibt Anfrageobjekt *q* und maximale Distanz ε vor
    - Ergebnis enthält alle Objekte, die höchstens eine Distanz von ε zu *q* haben
  - Formal

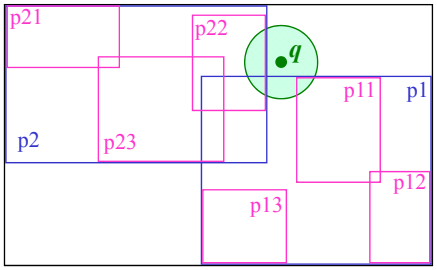
$$RQ(q,\varepsilon) = \{o \in DB \mid dist(q,o) \leq \varepsilon\}$$

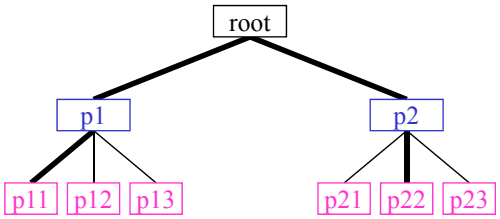
- Basisalgorithmus (sequential scan)

```
RQ-SeqScan(DB, q, ε)
    result = ∅;
    FOR i=1 TO n DO
        IF dist(q, DB.getObject(i)) ≤ ε THEN
            result := result ∪ getObject(i);
    RETURN result;
```

– Algorithmus mit Index: Tiefensuche

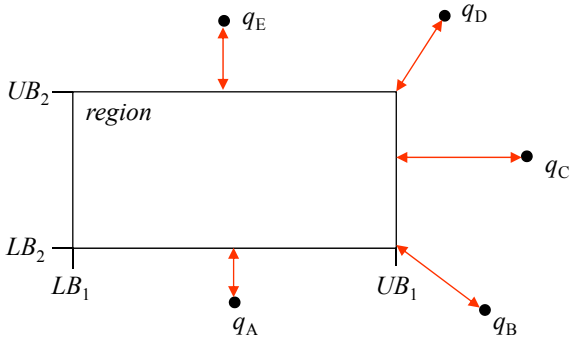
```
RQ-Index(pa, q, ε)          // pa = Diskadress z.B. der Wurzel des Indexes
    result = ∅;
    p := pa.loadPage();
    IF p.isDataPage() THEN
        FOR i=0 TO p.size() DO
            IF dist(q, p.getObject(i)) ≤ ε THEN
                result := result ∪ getObject(i);

    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;
```





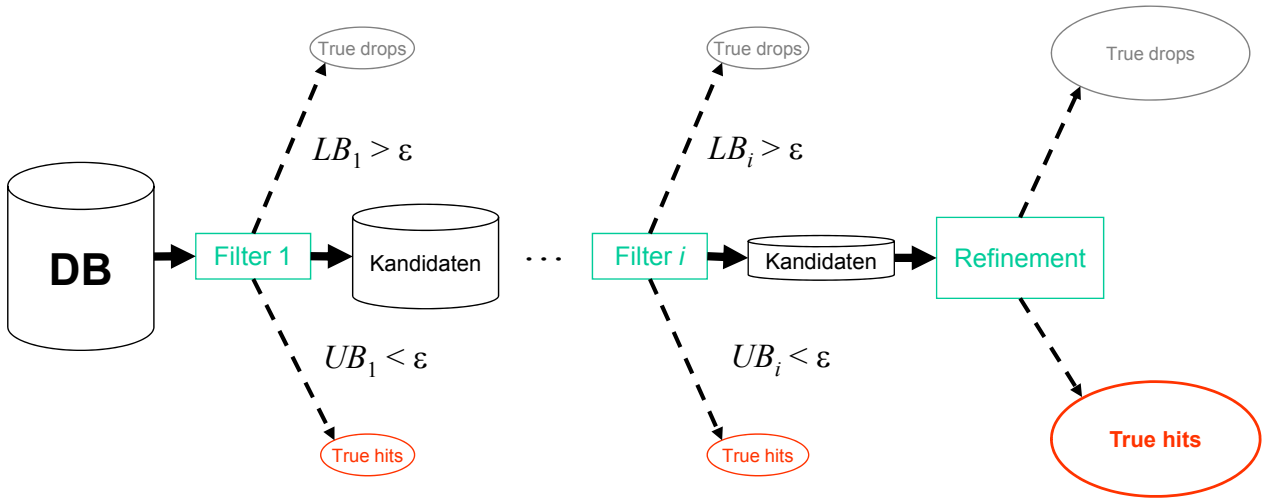
- MINDIST
  - Test ob Queryregion sich mit Seitenregion schneidet
  - Minimale Distanz zwischen Anfragepunkt und allen Punkten der Seitenregion (=> Lower Bound!!!)
  - Beispiel: Berechnung der MINDIST für L_2-Norm

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



- Mehrstufiger Algorithmus: Filter-/Refinement
  - Lower Bounding Filterdistanz LB
  - Upper Bounding Filterdistanz UB

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

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

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

- Oft nur Lower Bounding Distanzen

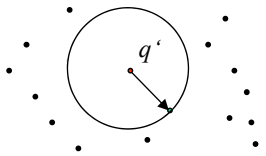
=> Anzahl der Kandidaten größer, da keine true hits

2.4 Nächste Nachbarn Anfragen

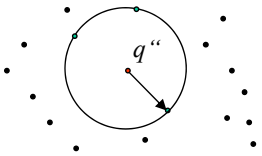
2.4.1 Nächste Nachbarn Anfrage

– Allgemeines
  - Eigenschaften
    – Benutzer gibt Anfrageobjekt *q* vor
    – Ergebnis enthält das Objekt, das die geringste Distanz zu *q* hat
    – Mehrdeutigkeiten müssen sinnvoll behandelt werden (mehrere nächste Nachbarn, oder nichtdeterministisch ein Objekt)
  - Formal

$$NN(q) = \{o \in DB \mid \forall x \in DB : dist(q,o) \leq dist(q,x)\}$$



Eindeutiges Ergebnis



Mehrdeutiges Ergebnis

– Basisalgorithmus (sequential scan): nichtdeterministisch

```
NN-SeqScan(DB, q)
    result = ∅;
    stopdist = +∞;
    FOR i=1 TO n DO
        IF dist(q, DB.getObject(i)) ≤ stopdist THEN
            result := getObject(i);
            stopdist = dist(q, DB.getObject(i));
    RETURN result;
```

– Algorithmus mit Index: Einfache Tiefensuche
  - Unterschied zur Range-Query
    – Nächste Nachbar kann beliebig weit vom Anfragepunkt weg liegen
    – Gestalt der Query zunächst unbekannt
    – Es kann zunächst nicht anhand der Seitenregion entschieden werden, ob eine Seite gebraucht wird
    – Ob eine Seite gebraucht wird, hängt auch von dem Inhalt der anderen Seiten ab

- Kennt man NN-Distanz, würde Range Query ausreichen
- Kennt man ein beliebiges Objekt, kann man dessen Abstand als obere Schranke für die NN-Distanz nutzen
- Kennt man mehrere Objekte, kann man den geringsten Abstand als obere Schranke für die NN-Distanz nutzen

• Umformulierung des RQ-Algorithmus:
- Verwende als ε die kleinste Distanz zu den bisher gefunden Nachbarn

```
Globale Variable: stopdist = +∞;

NN-Index-Simple-TS(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(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-Index-Simple-TS(p.childPage(i), q)
  RETURN result;
```

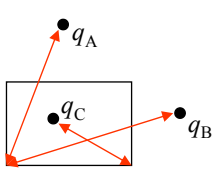

• Nachteil des einfachen Tiefensuch-Algorithmus
- Initialisierung: stopdist = +∞
- Dadurch: Start mit beliebigem Pfad
- Folge: die ersten gefundenen Objekte sind meist sehr weit vom Anfrageobjekt entfernt => stopdist ist wenig selektiv
- Verbesserung: beginne Pfad, der möglichst nah zum Anfrageobjekt liegt

– Algorithmus mit Index: Tiefensuche nach [RKV 95]

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

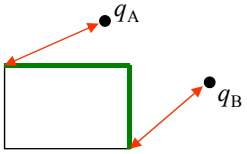
• Vermeidet langsame Einschränkung des Suchraums durch
- Verwendung der Seitenregionen zur Abschätzung der NN-Distanz
- Priorisierung der Tiefensuche nach Distanz der Seitenregion zur Query

• 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



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

– MINMAXDIST
  » MBRs als Seitenregionen: maximale NN-Distanz noch besser abzuschätzen
  » Auf jeder Kante des MBR muss ein Punkt liegen (sonst ist MBR nicht minimal)
  » Intuition: „nächstliegende Kante, weitester Punkt“



$$MINMAXDIST(region, q) = \sqrt{\min_{0<k\le d}(|q_i - rm_i|^2 + \sum_{\substack{i\neq k \\ 0<i\le d}} |q_i - rM_i|^2)}$$

wobei $$rm_i = \begin{cases} region.LB_i & \text{if } q_i \le \frac{region.LB_i + region.UB_i}{2} \\ regionUB_i & \text{else} \end{cases}$$

$$rM_i = \begin{cases} region.LB_i & \text{if } q_i \ge \frac{region.LB_i + region.UB_i}{2} \\ regionUB_i & \text{else} \end{cases}$$

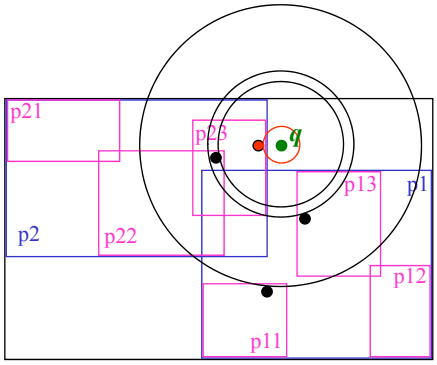
– Für andere Geometrien (nicht MBRs) sind MINDIST und MAXDIST analog definierbar; MINMAXDIST allerdings nicht
– Abschätzung von stopdist durch Minimum aus stopdist und MINMAXDIST (bzw. MAXDIST) aller bisher bekannten Seitenregionen (pruningdist)
– Vor dem rekursiven Abstieg: sortieren der Kindseiten nach MINDIST (experimentell als bestes Prioritätsmaß ermittelt)

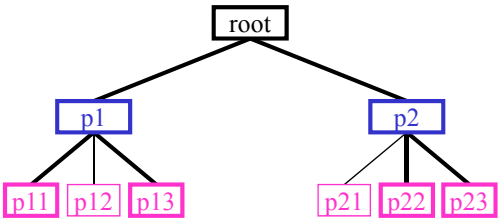
– Algorithmus:

Globale Variablen: stopdist = +∞; pruningdist = +∞;

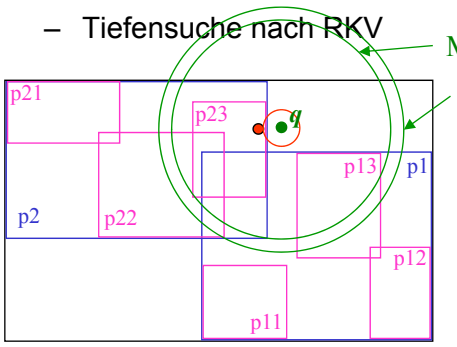
```
NN-Index-RKV-TS(pa, q)          // pa = Diskadress z.B. der Wurzel des Indexes
  result = ∅;
  p := pa.loadPage();
  IF p.isDataPage() THEN
    FOR i=0 TO p.size() DO
      IF dist(q, p.getObject(i)) ≤ stopdist THEN
        result := getObject(i);
        stopdist = dist(q, p.getObject(i));
    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

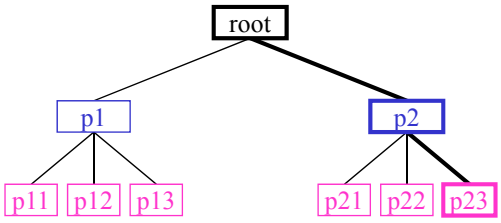
- Ablaufbeispiel
  - Einfache Tiefensuche



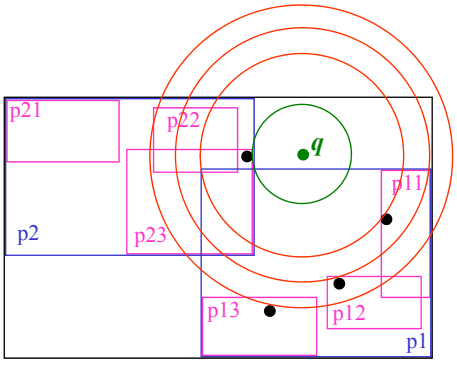


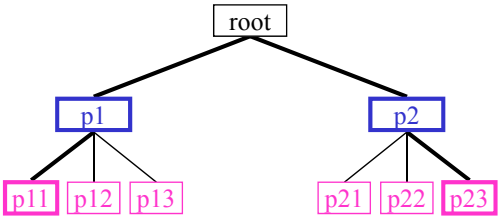
  - Tiefensuche nach RKV





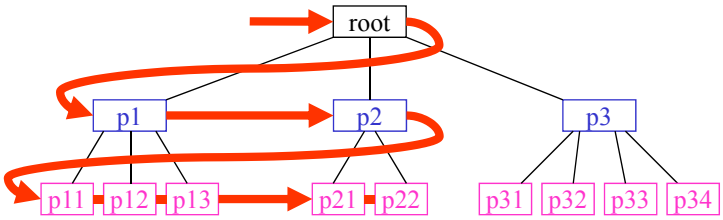
- Fazit:
  - Priorisierung mit MINDIST bewirkt Reduktion der Seitenzugriffe von 7 auf 3
  - MINMAXDIST verbessert Pruning-Distanz, verhindert hier aber keine Seitenzugriffe
  - Trotz Priorisierung: Tiefendurchlauf kann prinzipiell stark fehlgeleitet werden wenn z.B. eine Seite auf dem ersten Level sehr 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

– Algorithmus mit Index: Breitensuche



- Abgesehen von MINMAXDIST-Abschätzung stehen Punktdistanzen erst auf Blatt-Ebene zur Verfügung
  - Viele Zugriffe auf Directoryseiten
  - Die erste Punktdistanz hätte viele dieser Zugriffe schon verhindern können
- Speicherintensiv
  - Worst-case: gesamte letzte Directory-Ebene muss im RAM gehalten werden

– Algorithmus mit Index: Prioritätssuche nach [HS 95]

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

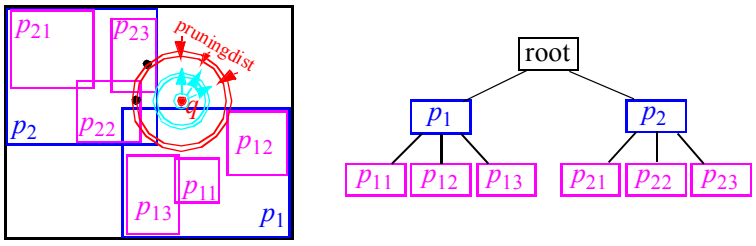
- Statt rekursivem Durchlauf: Liste der aktiven Seiten (active page list APL)
  - Seite p ist aktiv genau dann wenn folgende Bedingungen erfüllt sind:
    - » p wurde noch nicht geladen
    - » Elternseite von p wurde bereits geladen
    - » MINDIST(q, p.getRegion()) ≤ pruningdist
  - APL wird mit Wurzel 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: stopdist = +∞; pruningdist = +∞;

```
NN-Index-HS(pa, q)              // pa = Diskadress z.B. der Wurzel des Indexes
    result = ∅;
    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.getFirst().da.loadPage();
        apl.deleteFirst();
        IF p.isDataPage() THEN

            (* wie bisher *)

        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));
    RETURN result;
```


- 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

- 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)

- **Optimalität des Verfahrens**

  [Berchtold, Böhm, Keim, Kriegel. ACM Smp. Principles of database Systems (PODS), 1997]

  - Prioritätssuche nach [HS 95] ist optimal bzgl. der Anzahl der Seitenzugriffe
  - Beweis (Überblick):
    - » Lemma 1: jeder korrekte Algorithmus muss mind. die Seiten laden, die von der NN-Kugel um q berührt werden
    - » Lemma 2: das Verfahren greift auf Seiten in aufsteigendem Abstand von q zu
    - » Lemma 3: keine Seite s wird zugegriffen, mit MINDIST(q, s) > NN-Distanz(q)

  - **Lemma 1:** Ein korrekter NN-Algorithmus muss mind. die Seiten s laden, die MINDIST(q, s) ≤ NN-Distanz(q) erfüllen.

    Beweis: Angenommen eine Seite s mit MINDIST(q, s) ≤ NN-Distanz(q) wird nicht geladen. Dann kann diese Seite Punkte enthalten (als Datenseite; Directoryseiten können im entspr. Teilbaum Punkte speichern), die näher am Anfragepunkt liegen als der nächste Nachbar. Der nächste Nachbar ist also nicht als solcher validiert, da über Punkte in einem Teilbaum keine Infos bekannt sind, außer dass sie in der entsprechenden Region liegen.

    □

  - **Lemma 2:** Das Verfahren greift auf die Seiten des Index aufsteigend sortiert nach MINDIST zu.

    Beweis: Die Seiten werden in aufsteigender Reihenfolge aus der APL entnommen. Es muss also nur sichergestellt werden, dass nach Entnahme von Seite s keine Seiten s' mehr in APL eingefügt werden, mit MINDIST(q,s') < r := MINDIST(q,s). Alle Seiten, die nach Entnahme von s in APL eingefügt werden, sind entweder Kindseiten von s oder Kindseiten von Seiten s'' mit MINDIST(q,s'') ≥ r. Da die Region einer Kindseite in der Region der Elternseite vollständig eingeschlossen ist, ist die MINDIST einer Kindseite nie kleiner als die der Elternseite. Daher haben alle später eingefügten Seiten eine MINDIST ≥ r.

    □

  - **Lemma 3:** Das Verfahren greift auf keine Seite s zu, mit MINDIST(q,s) > NN-Distanz(q).

    Beweis: Nach Lemma 2 können nach Zugriff auf Seite s nur Punkte p gefunden werden, mit dist(q, p) > MINDIST(q, s). Wäre vor Zugriff auf s ein Punkt p mit dist(q, p) < MINDIST(q, s) gefunden worden, dann wäre s aus der APL gelöscht worden bzw. der Algorithmus hätte vor der Bearbeitung von p angehalten.

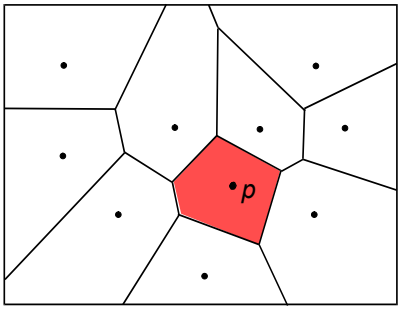
    □

  - Aus Lemma 1-3 ergibt sich, dass der Algorithmus nach [HS 95] optimal bzgl. der Anzahl der Seitenzugriffe ist.

– Hybrider Algorithmus: Voronoi-Diagramme

[Berchtold, Ertl, Keim, Kriegel, Seidl. Proc. Int. Conf. Data Engineeering (ICDE), 1998]

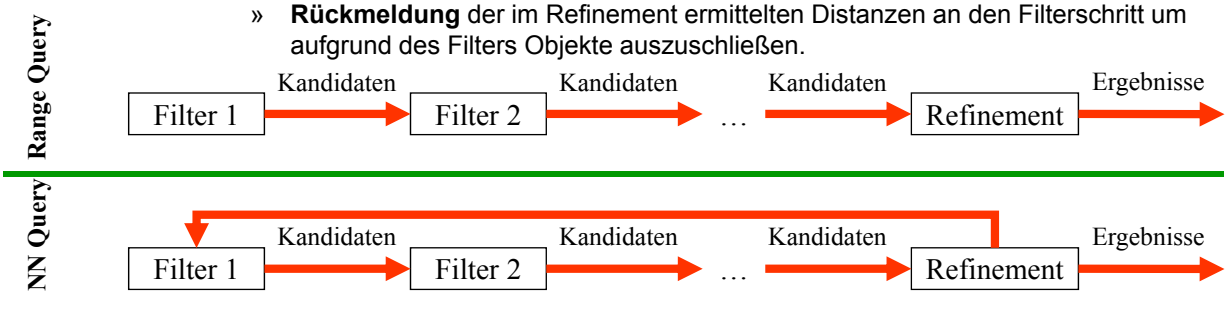
- Nur für Vektordaten!!!
- Idee:
  - Berechne für jeden Punkt *p* den Teil des Datenraumes in dem *p* der nächste Nachbar ist (Voronoi-Zellen)
  - Speichere Voronoi-Zellen in DB
  - NN-Anfrage entspricht Punktanfrage mit *q* auf den Voronoi-Zellen
    => Punkt *p*, in dessen Voronoi-Zelle *q* liegt, ist der NN von *q*



- Problem:
  - Voronoi-Zellen sind konvexe Polygone
  - Ab $d > 2$ sehr komplex (große Anzahl Eckpunkte)
- Lösung: Approximation der Voronoi-Zellen (z.B. mit MBR)
  - Nur Filterschritt, da MBRs sich überlappen können, und q in mehrere dieser MBRs liegen kann
    => Verfeinerung der Kandidaten mit exakten Punktdistanzen

– Mehrstufiger Algorithmus: Filter/Refinement

- 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 sequentiell ausgewertet werden.
    - » Bei NN-Anfragen (NNQ) nicht so leicht möglich. Grund: Der (erste) Filter ist nicht in der Lage eine NN-Kandidatenmenge (als kleine Teilmenge der Datenbank) zu identifizieren, die garantiert den exakten NN enthält. (Unter Beschränkung auf den LB-Filter)
    - » 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.



- Im folgenden:
  - Verschiedene Auswertungsstrategien
  - Ein Filterschritt (leicht generalisierbar auf mehrere Filterschritte)

- 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 (siehe Kapitel 2.4.3)
    - » 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 stopdist überschreitet
    - » Verfeinere das erhaltene Objekt und passe ggf. die stopdist 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

```
NN-MultiStep-Optimal(DB, q)
    Ranking = initialisiere Ranking bzgl. q auf Filterdistanz      // Kapitel 2.4.3
    result = ∅;
    stopdist = +∞;
    REPEAT
        p = Ranking.getNext();
        filterdist = dist_Filter(p, q);
        IF dist_Filter(p, q) ≤ stopdist THEN
            IF dist(q, p) ≤ stopdist THEN
                result = p;
                stopdist = dist(q, p);
    UNTIL filterdist > stopdist;
    RETURN result;
```


2.4.2 k-nächste Nachbarn (*k*-NN) *Anfragen*

– Allgemeines
- Eigenschaften
  – Benutzer gibt Anfrageobjekt *q* und Anzahl *k* vor
  – Ergebnis enthält die *k* nächsten Nachbarn von *q*
  – Mehrdeutigkeiten müssen wiederum sinnvoll behandelt werden
- Formal
  – Deterministisch
  kleinste Menge *NN*(*q*,*k*) ⊆ DB mit mindestens *k* Objekten, sodass
  $$\forall o \in NN(q,k), \forall o' \in DB - NN(q,k) : dist(q,o) < dist(q,o')$$
  – Nicht-deterministisch
  Menge *NN*(*q*,*k*) ⊆ DB mit exakt *k* Objekten, sodass
  $$\forall o \in NN(q,k), \forall o' \in DB - NN(q,k) : dist(q,o) \leq dist(q,o')$$
  – Klar: NN(*q*,1) ≡ NN(*q*)

– Basisalgorithmus (sequential scan): nichtdeterministisch

```
NN-SeqScan(DB, q, k)
    result = LIST OF (dist:REAL, p:OBJECT) ORDERED BY dist DESCENDING;
    result = [];
    FOR i=1 TO k DO
        result.insert(dist(q, getObject(i), getObject(i));
    FOR i=k+1 TO n DO
        IF dist(q, DB.getObject(i)) ≤ result.getFirst().dist THEN
            result.deleteFirst();
            result.insert(dist(q, getObject(i), getObject(i));
    RETURN result;
```

- Bemerkung:
  - Liste *result* wird als Heap anstelle einer sortierten Liste implementiert
    - » Grund: Die Liste *result* braucht nicht vollständig sortiert sein – es reicht sicherzustellen, dass das erste Element in *result* immer den größten Distanzwert hat. Ermöglicht Einfügen in O(log(k)) .

– Algorithmen mit Index
  - Grundsätzlich lassen sich alle Algorithmen zur NN-Suche auf *k*-NN-Suche erweitern, egal ob Index-basiert oder mehrstufig
    - Pruningdistanz ist entsprechend immer die Distanz zum aktuell gefundenen *k*-NN (erstes Element in result-Liste)
    - Beim Algorithmus nach [RKV] kann MINMAXDIST zu einer Seite nur wie Distanz zu einem Punkt gewertet werden und nicht als Gesamt-Pruningdistanz => MINMAXDIST lohnt sich i.A. nicht für *k*-NN Anfragen

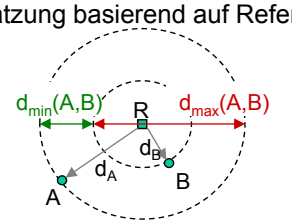
– Algorithmen mit Multi-Step Architektur
  - Alle drei Alternativen leicht erweiterbar
    - Auswertung mit Bereichsanfrage
      - » *k*-NN Anfrage statt NN Anfrage im Filter und Refinement anpassen (siehe Übung)
    - Unmittelbare Verfeinerung
      - » erweitere *k*-NN-Algorithmus statt NN-Algorithmus um entsprechende Aufrufe
    - Auswertung nach Priorität
      - » benutze *k*-NN-Distanz als Abbruchkriterium (siehe Übung)

      Bemerkung: Hier gilt die Verfeinerungsoptimalität nur unter der Annahme der Beschränkung auf einen LB-Filter (Grund siehe später)

– Verfeinerungsoptimale *k*-NN Anfrage

- Grundsätzlich:
  - Unterscheidung der Optimalität bzgl.
    - I/O Kosten (Seitenzugriffe im Index) = Kosten im Filterschritt
    - CPU Kosten für die Berechnung der exakten Distanz = Kosten im Verfeinerungsschritt
    - » Optimalität eines mehrstufigen Anfragealgorithmus hängt von der im Filterschritt zur Verfügung stehenden Distanzinformation ab, d.h. mehr Information im Filterschritt
      ⇒ höhere Selektivität des Filters
      ⇒ weniger Seitenzugriffe, weniger Verfeinerungen
  - Ziel:
    - » Kandidatenmenge im Filterschritt durch Berücksichtigung weiterer Filterkriterien oder zusätzlicher Information weiter reduzieren
  - Motivation:
    - » Exakte Distanzberechnung sehr teuer im Vergleich zur Filterdistanzberechnung
      ⇒ Oft lohnt es sich den Filter durch zusätzliche Filterinformationen zu verbessern
    - » In vielen Applikationen können Ähnlichkeitsdistanzen sowohl nach unten als auch nach oben hin effizient abgeschätzt werden

  - Beispiele für die Ermittlung von unterer bzw. oberer Distanzabschätzung:
    - » Abschätzung basierend auf Referenzpunkten (z.B. M-tree)



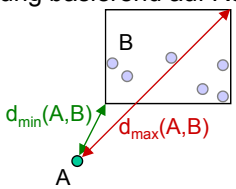
$d_{min}(A,B) = |d_A - d_B|$

$d_{max}(A,B) = d_A + d_B$

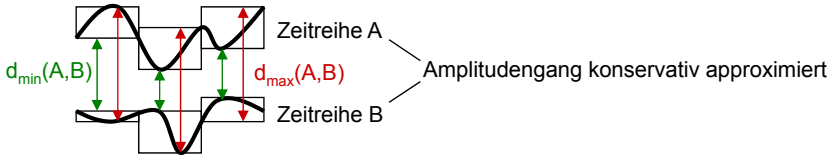
$d_{min}(A,B) \leq d(A,B) \leq d_{max}(A,B)$

LB exakt UB

    - » Abschätzung basierend auf Regionen (z.B. R-tree)

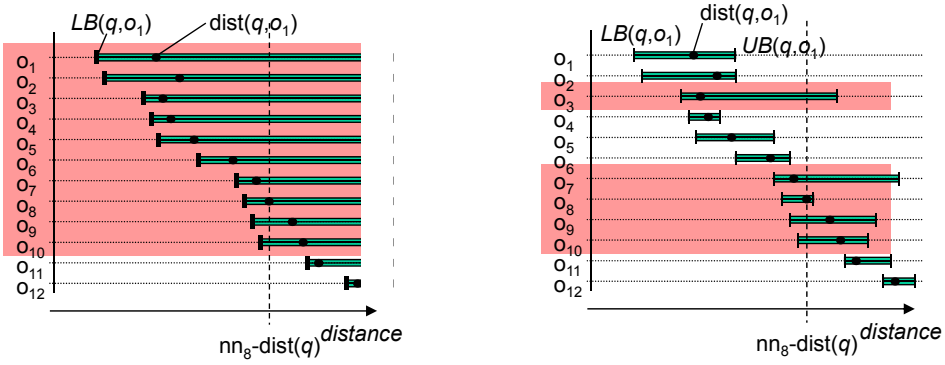


    - » Individuelle Abschätzung (Applikations/Daten abhängig)



  - Idee:
    - » zusätzlich zur unteren Distanzabschätzung (LB) wird auch obere Distanzabschätzung (UB) im Filterschritt miteinbezogen
      ⇒ optimale Auswertung mit unterer und oberer Distanzabschätzung

- LB-basierte *k*-NN-Suche vs. (LB+UB)-basierte *k*-NN-Suche



  - nn_k-dist(q) bezeichne die Distanz des k-nächsten Nachbarn von dem Anfrageobjekt q
  - Alle Objekte o deren Filterdistanzen die Eigenschaft $LB(q,o) \leq nn_k\text{-dist}(q) \leq UB(q,o)$ erfüllen, müssen verfeinert werden.
  - Bei der LB-basierten *k*-NN-Suche müssen mehr Objekte verfeinert werden als bei der (LB+UB)-basierten *k*-NN-Suche
    - » (siehe Beispiel oben) LB-basierte *k*-NN-Suche muss 10 Objekte verfeinern, während die (LB+UB)-basierte *k*-NN-Suche nur 5 Objekte verfeinern muss
    - » Voraussetzung: Die Berechnung der exakten Distanz für die Resultatmenge ist nicht erforderlich

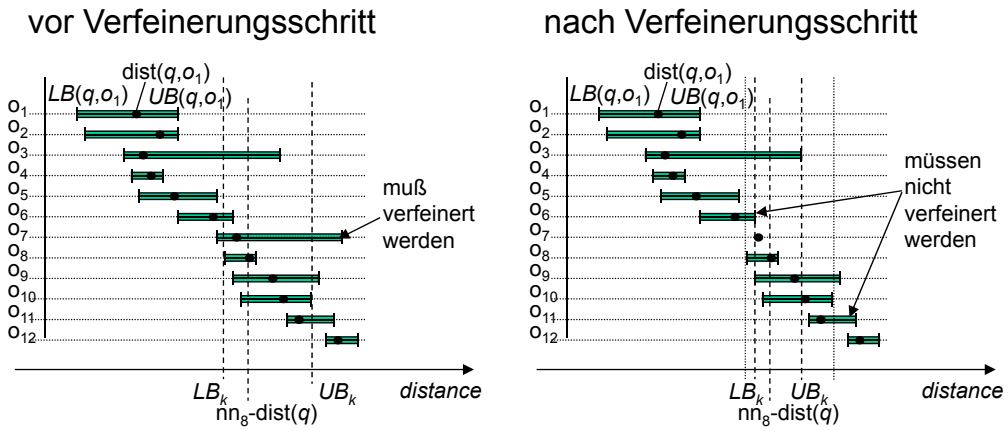
– Optimale (LB+UB)-basierte *k*-NN-Suche

[Kriegel, Kröger, Kunath, Renz. 10th Int. Symp. on Spatial and Temporal Databases (SSTD'07), 2007]

- Optimalität:

  Beweisbar: Algorithmus ist optimal bzgl.
  - » Anzahl der Seitenzugriffe (Filterschritt) und
  - » Anzahl der Verfeinerungen (Verfeinerungsschritt)
  - Beruht auf dem Prinzip der iterativen Verfeinerung (analog zu „Auswertung nach Priorität“ s. Folie 77):
    - » auf Filterebene läuft „Ranking Query“ ab (siehe Kapitel 2.4.3)
    - » Filtern aufgrund von unterer und oberer Schranke
    - » nach jedem Verfeinern wird erneut gefiltert
    - » Objekt nur dann anfordern, wenn unbedingt notwendig
    - » Objekt nur dann verfeinern, wenn unbedingt notwendig
- Vorteil
  - Einsparungen gegenüber dem Algorithmus „Auswertung nach Priorität“ durch zusätzliche Verwendung der oberen Distanzabschätzung UB(q,o)
- Nachteil
  - Komplexität des Ranking-Algorithmus (Speicher und/oder Zeit) bleibt
  - Kein exaktes „Ranking“ auf den *k* Ergebnisobjekten

- Prinzip:



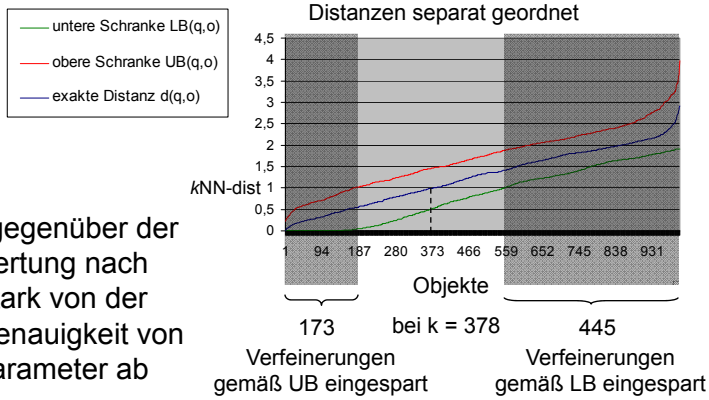
– konservative Approximation der *k*-NN-Distanz nn_k-dist(*q*) durch
  » LB_k = *k* kleinste LB-Filterdistanz
  » Ub_k = *k* kleinste UB-Filterdistanz
– Ziel: Nur diejenigen Objekte verfeinern, deren untere und obere Distanzabschätzung die *k*-NN-Distanz nn_k-dist(*q*) überdeckt
– Beweisbar: Es existiert immer mind. ein Kandidat, dessen untere und obere Distanzabschätzung sowohl LB_k als auch UB_k und somit auch die *k*-NN-Distanz nn_k-dist(*q*) überdeckt

– Algorithmus

k-NN-MultiStep-Optimal(DB, *q*)
Ranking = initialisiere Ranking bzgl. *q* auf LB-Filterdistanz; // Kapitel 2.4.3
result = ∅; *candidates* = ersten *k* Objekte aus dem Ranking; intitialisiere UB_k, LB_k aus *candidates*;
REPEAT
  // Schritt 1: hole nächsten Kandidaten
    if $LB_{next} \le LB_k$ then // LB_{next} = LB(*q*,o_{next}), wobei o_{next} = nächstes Obj. im Ranking
      p = Ranking.getNext();
      füge *p* zu *candidates* hinzu //, falls LB(*q*,*p*) ≤ UB_k; // LB_{next} evtl. Distanz zu MBR ???
    endif;
    aktualisiere LB_k, UB_k, LB_{next};
  // Schritt 2: Filtere true hits und true drops aus (Filter-Schritt)
    for each *c* ∈ *candidates* do
      if UB(*q*,*c*) ≤ LB_k then nehme *c* aus *candidates* und füge es zu *result* hinzu; // true hit
      if LB(q,c) > UB_k then entferne *c* von *candidates*; // true drop
    end for;
  // Schritt 3: Verfeinere Kandidaten (Verfeinerungs-Schritt)
    if |*result*|+|*candidates*| ≤ *k* und LB_{next} > UB_k then
      füge alle *c* ∈ *candidates* zu *result* hinzu; // Abbruchbedingung
    else
      verfeinere alle *c* ∈ candidates, die LB(q,c)≤LB_k≤UB_k≤UB(q,c) erfüllen, d.h.
      berechne d_{exakt}(*q*,*c*) und setze LB(*q*,*c*) = UB(*q*,*c*) = d_{exakt}(*q*,*c*);
    end if;
UNTIL(|*candidates*|=0 und LB_{next} > UB_k);
RETURN *result*;

- **Optimalität gemäß der Seitenzugriffe:**
  In *Schritt 1*: die Bedingung „$LB_{next} \leq LB_k$“ garantiert, dass nur die notwendigen Objekte angefordert werden $\Rightarrow$ minimale Seitenzugriffe
- **Optimalität gemäß der Anzahl der Verfeinerungen:**
  In *Schritt 3* die Bedingung „$LB(q,c) \leq LB_k \leq UB_k \leq UB(q,c)$“ garantiert, dass nur diejenigen Objekte verfeinert werden, die verfeinert werden müssen, d.h. für die gilt: $LB(q,c) \leq nn_k\text{-dist}(q) \leq UB(q,c)$ $\Rightarrow$ minimale Verfeinerungen
- Experimente auf Realdaten (NN-Suche auf Zeitreihen mit DTW-Distanz)

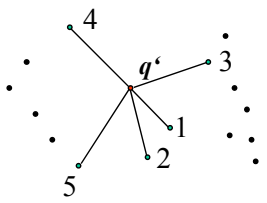
Bemerkung:
Effizienzgewinn gegenüber der einfachen „Auswertung nach Priorität“ hängt stark von der Approximationsgenauigkeit von UB und dem *k* Parameter ab



2.4.3 Nächste Nachbarn Ranking

– Allgemeines

- Eigenschaften
  - Benutzer gibt Anfrageobjekt *q* vor und initialisiert damit das Ranking
  - Benutzer kann mehrfach Funktion getNext() aufrufen, die ihm jeweils den 1., 2., usw. Nachbarn von *q* zurück gibt.
  - Mehrdeutigkeiten müssen wiederum sinnvoll behandelt werden
    - » Typischerweise nicht-deterministisch: der *k*-te Aufruf ergibt einen der *k*-NN



– Basisalgorithmen (siehe Übung!!!)

– Algorithmus mit Index

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

- Alle *k*-NN-Algorithmen können entsprechend erweitert werden
- Problem der rekursiven Algorithmen
  - Nachdem der i-te Nachbar gefunden ist, wird das Ergebnis an die Ranking-Ausgabe übergeben
  - Weitere getNext()-Aufrufe erfordern erneutes rekursives Suchen
- Vorteil der Prioritätssuche
  - Kompletter Zustand des Algorithmus ist in apl und result gespeichert
- Unterschied zum *k*-NN-Algorithmus
  - Unbeschränkte Ergebnisliste *result* in die jeder Punkt einer geladenen Datenseite eingefügt wird (***aufsteigend*** nach Distanz zu *q* sortiert).
  - Keine Pruningdistanz => Kindseiten verfeinerter Seiten in APL einfügen
  - Algorithmus stoppt (für den aktuellen getNext()-Aufruf) sobald erste Seite in APL größere MINDIST zu *q* hat als bestes Element in *result*
  - Dieses Element wird aus result gelöscht und ausgegeben
  - Nächster getNext()-Aufruf arbeitet mit aktuellen APL und *result* weiter
- Hoher Speicherplatzbedarf: im worst case gesamte DB in *result*

- Algorithmus

```
Globale Variablen:
result = LIST OF (dist:Real, obj:Object) ORDERED BY dist ASCENDING;
apl = LIST OF (dist:Real, da:DiskAdress) ORDERED BY dist ASCENDING

NN-Ranking(pa, q)
    result = [(+∞, dummy)];
    apl = [(0.0, pa)];
    WHILE NOT apl.isEmpty() AND apl.getFirst().dist ≤ result.getFirst().dist DO
        p = apl.getFirst().da.loadPage();
        apl.deleteFirst();
        IF p.isDataPage() THEN
            FOR i=0 TO p.size() DO  // Jedes Objekt einfügen
                result.insert( (dist(q, p.getObject(i)),p.getObject(i)) );

        ELSE                                    // p ist Directoryseite
            FOR i=0 TO p.size() DO  // Jede Seite einfügen
                apl.insert( (MINDIST(q, p.getRegion(i)),p.getChildPage(i)) );
    resultObject = result.getFirst().obj;
    result.deleteFirst();
    RETURN resultObject;
```

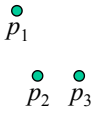
– Algorithmen mit Multi-Step Architektur
  - Nicht alle Varianten der NN-Algorithmen mit Multi-Step Architektur lassen sich zu Ranking Algorithmen erweitern
    – Auswertung mit Bereichsanfrage
      » Erweiterung nicht möglich, da kein ε ermittelbar
    – Unmittelbare Verfeinerung
      » Erweitere einen Ranking Algorithmus
    – Auswertung nach Priorität
      » Verwalte unbegrenzte Liste von Objekten statt result-Variable
  - Ranking-Algorithmus für Multi-Step *k*-NN-Anfragen wichtig, da die Resultate für weitere Filterschritte benötigt werden (bei Auswertung nach Priorität)
    – Ranking im Filterschritt notwendig, da die Ergebnismenge des Filters zunächst unbekannt. Ergebnismenge des Filters wird durch Ergebnise der Verfeinerungen bestimmt!!!

2.5 Reverse Nächste Nachbarn Anfragen

2.5.1 Allgemeines

- Eigenschaften
  – Benutzer gibt Anfrageobjekt *q* vor
  – Ergebnis enthält alle Objekte, die *q* als nächsten Nachbarn haben
  – Analog: Reverse *k*-nächste Nachbarn
  – Mehrdeutigkeiten (bei NN) entsprechend behandeln
- Formal
  – Reverse nächste Nachbarn $RNN(q) = \{o \in DB \mid q \in NN(o)\}$
  – Reverse *k*-nächste Nachbarn $RNN(q,k) = \{o \in DB \mid q \in NN(o,k)\}$
- Anwendungsbeispiel:
  – Standortsuche für neue Filiale (welche Kunden haben die neue Filiale als „nächsten Nachbarn“)

– Zusammenhang zwischen NN und RNN
  - NN ist keine symmetrische Relation
    – $y \in NN(x) \not\Leftrightarrow x \in NN(y)$
    – $y \in NN(x) \not\Leftrightarrow y \in RNN(x)$
  - RNN ist ein „eigenständiges Problem



	NN	RNN
p_1	$\{p_2\}$	$\{\}$
p_2	$\{p_3\}$	$\{p_3, p_1\}$
p_3	$\{p_2\}$	$\{p_2\}$

– Basisalgorithmus (sequential scan): nichtdeterministisch

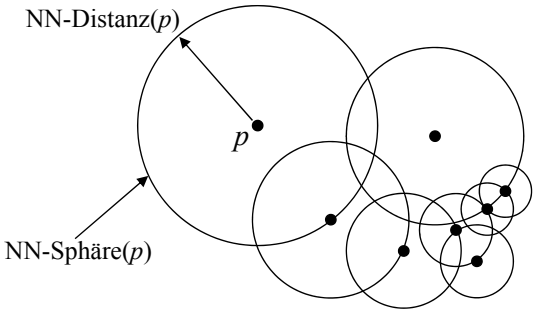
```
RNN-SeqScan(DB, q, k)
    resultSet = ∅;
    FOR i=1 TO n DO
        neighbors = NN-SequScan(DB, DB.getObject(i), k);
        IF q ∈ neighbors THEN
            resultSet.add(DB.getObject(i));
    RETURN resultSet;
```

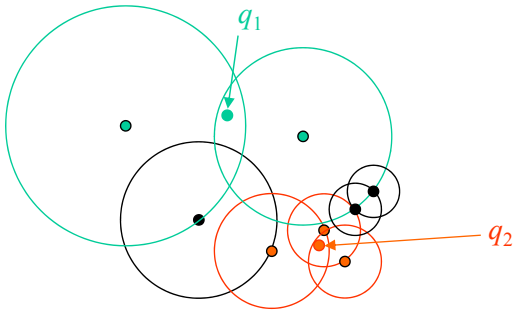
  - Offensichtliche Verbesserung
    – Statt NN-SequScan einen besseren NN-Algorithmus verwenden

– Index-basierte Methoden für die RkNN-Suche
  - Annahme:
    – Daten/Objekte in einer Baumartigen Indexstruktur, z.B. R-tree, organisiert
    – Suche erfordert hierarchischen Durchlauf der Directory-Seiten
  - Ziel
    – Bei der Suche möglichst früh Seiten auf höheren Index-Level ausschließen (d.h. effektive Pruning-Strategien)
      ==> möglichst starke Einschränkung des Suchraums
  - Pruning-Strategien
    – Generell gibt es zwei Index-basierte Pruning-Strategien für die RkNN-Suche
    – Self-Pruning Strategien:
      » Punkte/Seiten schließen sich selbst aus
      » Basieren auf k-NN-dist-Abschätzungen angewandt auf Punkte/Seitenregionen
      » Punkte/Seiten, deren k-NN-dist-Bereich den Anfragepunkt nicht enthalten können ausgeschlossen werden
    – Mutual-Pruning Strategien:
      » Punkte/Seiten schließen sich gegenseitig aus
      » Basieren auf Voronoi-Hyperebenen

2.5.2 Self-Pruning Strategien

- $p \in \text{RNN}(q) \Leftrightarrow \text{dist}(p, q) \leq \text{NN-Distanz}(p)$
- Materialisiere für alle Objekte die NN-Distanz
- Prüfe, ob $\text{dist}(p, q) \leq \text{NN-Distanz}(p)$ statt während der Anfrage eine NN-Query für alle Objekte zu berechnen

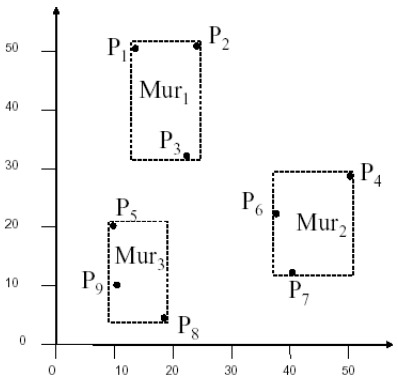




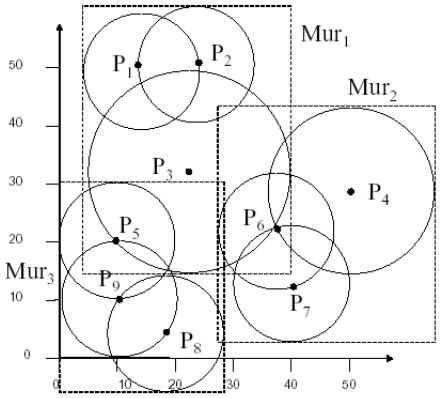
- Für Vektordaten
  - RNN-Query ist Punktanfrage bzgl. der NN-Sphären der Punkte (vgl. Voronoi-Ansatz zur NN-Query)
  - Speichere NN-Sphären in einem Index für ausgedehnte Objekte (z.B. R-Baum)

– RNN-Baum [Korn, Muthukrishnan. ACM Int. Conf. Management of Data (SIGMOD), 2000]

- RNN-Queries für Vektordaten
- Berechne NN-Distanz für alle DB-Punkte
- Speichere statt Punkte alle NN-Sphären der Punkte in R-Baum



Normaler R-Baum



RNN-Baum

- Algorithmus zur NN-Suche
  - Datenseiten enthalten Kreise, d.h. Objekte der Form (Punkt, Radius)
    - » Punkt = DB-Objekt (Mittelpunkt)
    - » Radius = NN-Distanz(Punkt)

```
RNN-Tree-Search(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(q, p.getObject(i).Point) ≤ p.getObject(i).Radius THEN
                result := result ∪ getObject(i).Point;

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


  - Vorteil
    - » Sehr gute Performanz (Pruning-Power) bei RNN-Anfragen
  - Nachteile
    - » *k* muss fest vorgegeben sein
    - » Nur für Vektordaten
    - » Hohe Überlappungen der Seitenregionen im Directory führt zu schlechter Performanz bei normalen NN-Anfragen
      Lösung: speichere einen RNN-Baum und einen konventionellen R-Baum
    - » Schlechte Performanz bei Einfügungen und Löschungen
      Beispiel: *Einfügen* von *p*

„Normaler“ Index	Bestimme NN(*p*) und füge Kreis (*p*, NN-Distanz(*p*)) in Index ein
Zusätzlich für RNN-Baum	Bestimme RNN(*p*)
	Erneuere NN-Sphären aller $o \in$ RNN(*p*)
	Erneuere Seitenregionen (von *p* und allen *o*) betroffener Datenseiten
	Erneuere rekursiv Seitenregionen der Vaterseiten

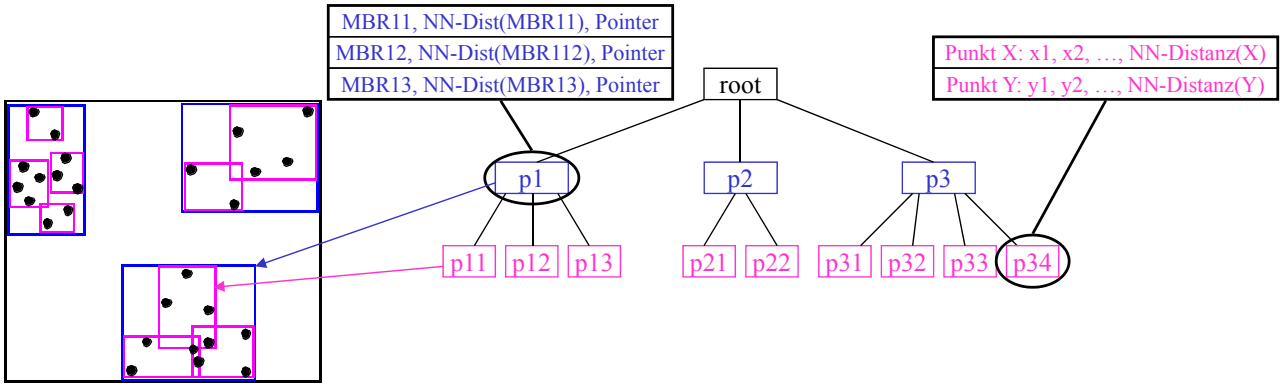
  - Varianten:
    - » Voronoi-Zellen statt NN-Sphären
    - » Andere Verfeinerungsreihenfolge (siehe NN-Algorithmen)

– RdNN-Baum [Yang, Lin. IEEE Int. Conf. Data Engineering (ICDE), 2001]

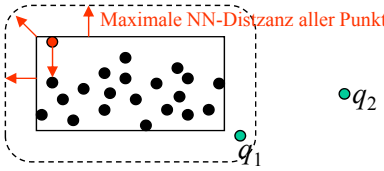
- Prinzip
  - Idee des RNN-Baum, ABER
  - Speichere die DB-Objekte statt NN-Sphären
  - Speichere zu jedem Punkt in der DB seine NN-Distanz
    - » Zu jedem Punkt zusätzlich einen Wert abspeichern
    - » Zu jeder Seitenregion *s* zusätzlich noch den Wert

$$\max_{child \in s} child.getNNDist()$$

abspeichern (Maximum aller NN-Distanzen in den Kinderseiten)



– Ausschluss von Directory-Seiten, wenn MINDIST(q,Seitenregion) größer ist, als die aggregierte NN-Distanz der Seitenregion



Query q_2: Seite kann ausgeschlossen werden; kein Punkt der Seite kann q_2 als NN haben

Query q_1: Seite kann nicht ausgeschlossen werden

- Algorithmus zur RNN-Suche

```
RdNN-Tree-Search(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(q, p.getObject(i)) ≤ p.getNNDist(i) THEN
        result := result ∪ getObject(i);

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

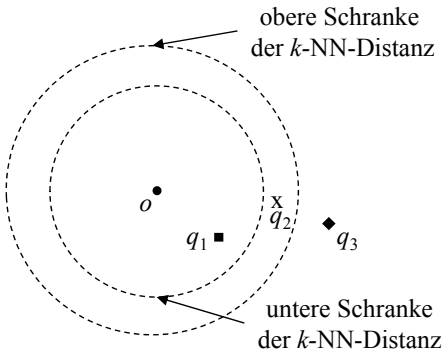
- Auch hier wieder alle möglichen anderen algorithmischen Lösungen zur NN-Suche anwendbar (Prioritätssuche, etc.)
- Vorteil
  - » Sehr gute Selektivität, damit gute Performanz bei Anfragen
  - » Seitenüberlappung wie bei „normalem“ R-Baum, daher kein extra Index für NN-/RQ-Anfragen nötig
- Nachteil
  - » *k* muss fest vorgegeben sein
  - » Nur für Vektordaten (aber: Konzept sehr leicht für M-tree erweiterbar)
  - » Weiterhin schlechte Performanz bei Einfügungen und Löschungen

– MRkNNCoP-Baum

[Achtert, Böhm, Kröger, Kunath, Pryakhin, Renz. ACM Int. Conf. Management of Data (SIGMOD), 2006]

- Vergleich bisheriger Verfahren
  - RNN-Tree/RdNN-Tree: Vorberechnung der NN-Distanz
    - » Wert für *k* ist fix und vorher bekannt
    - » Update-Problematik
    - » Dafür: erweiterbar auf metrische Daten (z.B. M-Tree)
  - Geometrische Suche
    - » Nur für Vektordaten
    - » Teurer Verfeinerungsschritt, schlechtere Selektivität
    - » Dafür: beliebiges *k*, keine Update-Problematik
- Idee:
  - Benutze die gute Selektivität der vorberechneten NN-Distanzen
  - Berechne für mehrere (am besten alle) Werte für *k* die *k*-NN-Distanzen vor
  - Problem: Speicherung aller Distanzen zu aufwendig
    - » Pro Objekt alle *k*-NN-Distanzen => Index wäre sehr hoch => hohe Kosten
  - Lösung: Approximiere die *k*-NN-Distanzen
    - » Approximation sollte untere Schranke (LB) der *k*-NN-Distanz sein => true drops, wir können Objekte (Seiten) frühzeitig ausschließen
    - » Zusätzliche Approximation als obere Schranke (UB) => true hits

- Bewertung von Objekt *o* (analog: Seiten) mit UB- und LB-Approximationen
  - dist(*o*,*q*) ≤ $LB_{k\text{-NN-Dist}}(o)$
    => *o* true hit, d.h. $o \in RNN(q, k)$
    » Beispiel: $q = q_1$
  - dist(*o*,*q*) ≥ $UB_{k\text{-NN-Dist}}(o)$
    => *o* true drop, d.h. $o \notin RNN(q, k)$
    » Beispiel: $q = q_3$
  - $UB_{k\text{-NN-Dist}}(o) \leq dist(o,q) \leq LB_{k\text{-NN-Dist}}(o)$
    => *o* Kandidat
    » Beispiel: $q = q_2$



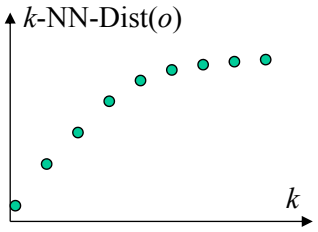
- Gegeben: für jedes Objekt *o* eine Sequenz der *k*-NN-Distanzen, <1-NN-Dist(*o*), 2-NN-Dist(*o*), …, k_{max}-NN-Dist(*o*)> für ein hinreichend großes k_{max}
- Frage: wie kann ich UB- und LB-Approximation dieser *k*-NN-Distanzen berechnen und kompakt speichern?

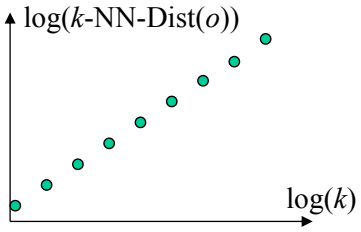
- Lösung aus der Theorie der Selbstähnlichkeit
  - Potentzgesetz gilt für Verhältnis zwischen
    » dem Radius einer Hyperkugel
    » der Anzahl an Objekten innerhalb der Hyperkugel

$$encl(\varepsilon) \propto \varepsilon^{d_f}$$

wobei *encl*(ε) = #Objekte innerhalb der Kugel
d_f = „Fraktale Dimension“

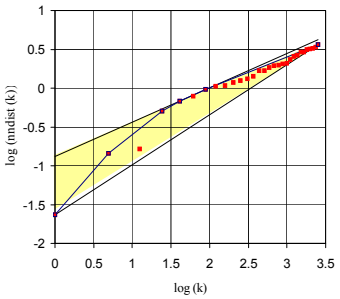
  - Übertragung auf k-NN-Sphäre:
    » ε = *k*-NN-Distanz
    » *encl*(ε) = *k*
  - Im log-log-Raum: $\log(k - NN - Dist(o)) \propto \frac{\log(k)}{d_f}$

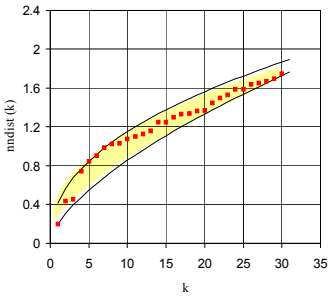


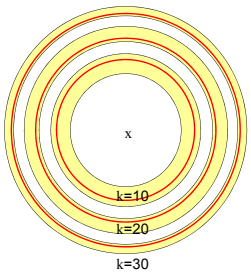


  - In der Realität verhalten sich die Distanzen nicht wie perfekte Linien im log-log-Raum
  - Trotzdem: im log-log-Raum können die *k*-NN-Distanzen mit einer Linie approximiert werden
  - Das ist erheblich billiger als alle *k*-NN-Distanzen zu speichern, oder andere Funktionen höherer Ordnung zu verwenden, um die Distanzen im normalen *k* / *k*-NN-Dist – Raum zu approximieren

- LB- und UB-Approximationen
  - UB-Approximation ist eine Linie im log-log-Space, sodass
    $$\forall k \leq k_{\max} : k - \mathrm{NN} - \mathrm{Dist}(o) \leq \mathrm{UB}_{k-\mathrm{NN}-\mathrm{Dist}}(o)$$
  - LB-Approximation ist eine Linie im log-log-Space, sodass
    $$\forall k \leq k_{\max} : k - \mathrm{NN} - \mathrm{Dist}(o) \geq \mathrm{LB}_{k-\mathrm{NN}-\mathrm{Dist}}(o)$$







- Jedem Objekt wird zugeordnet
  - Eine LB-Approximation der *k*-NN-Distanzen
  - Eine UB-Approximation der *k*-NN-Distanzen
- Jeder Seite im Index wird zugeordnet
  - Eine LB-Approximation der LB-Approximationen der Kindseiten
  - UB-Approximation wird nicht gespeichert, da zu wenig selektiv

- Vorteil:
  - Beliebiges *k*
  - Für allgemein metrische Daten (M-Tree) oder Vektordaten (z.B. X-Tree)
  - Durch UB- und LB-Approximationen höhere Filterselektivität als Geometrisches Verfahren => weniger Kandidaten die verfeinert werden müssen

- Nachteil
  - Updateproblematik
  - $k_{\max}$ muss bekannt sein (ABER i.d.R. kein Problem)
  - Teure Verfeinerung nötig (ABER i.d.R. deutlich weniger Kandidaten)

- ***Filter***-Algorithmus für allgemein metrische Daten (M-tree)

Knoten Node = (RoutingObj, CovRadius)
MINDIST(q,Node) = max{dist(q,Node.RoutingObj) – CovRadius, 0}

```
MRkNNCoP-Tree-Search(DB, q)      // DB als MRkNNCoP-Tree organisiert
    result = ∅;
    candidates = ∅;
    queue = LIST OF (dist:Real, obj:Object) ORDERED BY dist ASCENDING;
    queue = [(0.0, DB.root)];
    WHILE NOT queue.isEmpty() DO
        p = queue.first().Object;
        IF p.isDataPage() THEN
            FOR i=0 TO p.size() DO
                IF dist(q, p.getObject(i)) ≤ LB_k-NN-Dist(p.getObject(i)) THEN
                    result := result ∪ getObject(i);
                ELSE IF dist(q, p.getObject(i)) ≤ UB_k-NN-Dist(p.getObject(i)) THEN
                    candidates := candidates ∪ getObject(i);

        ELSE                    // p ist Directoryseite
            FOR i=0 TO p.size() DO
                IF MINDIST(q, p.getRegion(i)) ≤ UB_k-NN-Dist(p.getRegion(i)) THEN
                    queue.insert((MINDIST(q, p.getRegion(i)), p.childPage(i)));
```


2.5.3 Mutual Pruning Strategien

Idee:

- Ausschluss von Seiten/Objekten mittels Distanzabschätzungen (ohne Vorberechnung, zur Laufzeit der Anfrage ermittelt)
- Für mindestens k Objekte ist die Distanz zu Objekt o höchstens so groß wie die Distanz zwischen Anfrageobjekt q und o
  => Objekt o $\notin$ RkNN(q)
- Distanzabschätzungen über Indexseitenregionen

Vorteile:

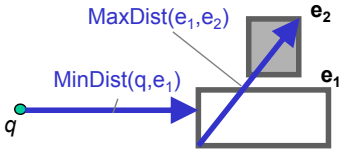
- Keine Vorberechnung notwendig
  → keine Update-Problematik mehr
- Parameter k zur Anfragezeit frei wählbar

– Min/Max-Distanz-Ansatz

[Achtert, Kröger, Kriegel, Renz, Züfle, EDBT, 2009]

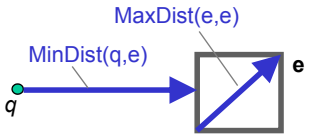
– Vorraussetzung: Indexseite speichert Anzahl der Objekte |e| in Seite e

Ausschluß von Seite e durch Seite e‘ (k≥1):



$MaxDist(e_1,e_2) < MinDist(q,e_1)$
$\Rightarrow e_1$ kann keine RkNN(q)-Ergebnisse enthalten wenn $|e_2| \geq k$

Seite e schließt sich selbst aus (k≥1):



$MaxDist(e,e) < MinDist(q,e)$
$\Rightarrow$ e kann keine RkNN(q)-Ergebnisse enthalten wenn $|e|-1 \geq k$

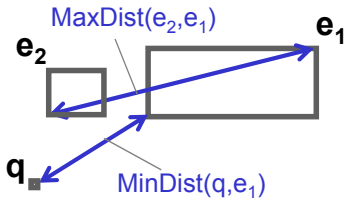
$MinDist(e_1,e_2)$: Distanz zwischen den beiden voneinander am nächsten liegenden Punkten $p_1 \in e_1$ und $p_2 \in e_2$

$MaxDist(e_1,e_2)$: Distanz zwischen den beiden voneinander entferntesten Punkten $p_1 \in e_1$ und $p_2 \in e_2$

– Vorteile:
- » Keine Vorberechnung mehr (keine Update-Problematik)
- » Parameter k zur Anfragezeit frei wählbar
- » Pruningkonzept auf allg. metrische Daten anwendbar

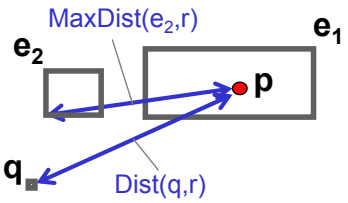
– Problem:
- » Abhängigkeiten zwischen Distanzen unberücksichtigt

Min/Max-Dist Pruning:



$MinDist(q,e_1) < MaxDist(e_2,e_1)$

„Optimales“ Pruning:



$\forall p \in e_1$: $Dist(q,r) > MaxDist(e_2,r)$

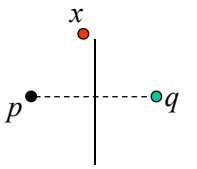
- » Die Distanz $MinDist(q,e_1)$ schätzt die Distanz dist(q,p) zwischen q und einem Objekt p in Seitenregion e_1 (untere Distanzabschätzung)
- » Die Distanz $MaxDist(e_2,e_1)$ schätzt die Distanz dist(o,p) zwischen einem Objekt o in Seitenregion e_2 und einem Objekt p in Seitenregion e_1 (obere Distanzabschätzung)
- » Beide Distanzen dist(q,p) und dist(o,p) hängen von der Lage des Objektes p in Region e_1 ab => dist(q,p) abhängig von dist(o,p)
- » Diese (Abhängigkeits-) Information geht bei der Verwendung von $MaxDist(e_2,e_1)$ und $MinDist(q,e_1)$ verloren

→ geringeres Pruningpotential

– Geometrische RNN-Suche (Filter/Verfeinerung)

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

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

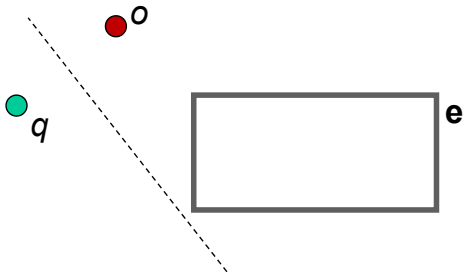


Voronoi-Ebene $E(p,q)$ zwischen p und q

  » Voronoi-Ebene $E(p,q)$: für alle Punkte $e \in E$ gilt: dist(q,e) = dist(p,e)

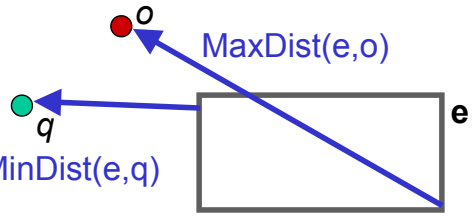
– „Geometrisches“ Pruning ist optimal bzgl. der Pruning-Stärke

geometrisches Pruning:



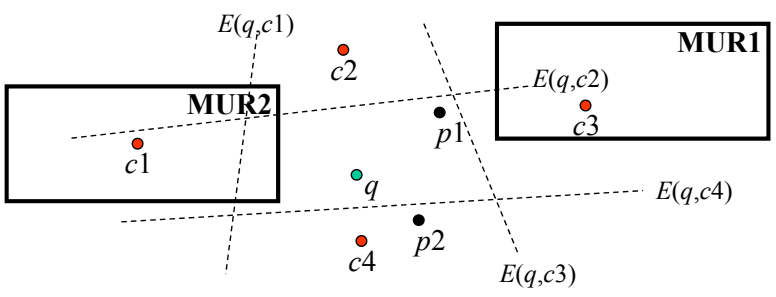
e hinter der Hyperebene
=> prune e

Min/Max-dist Pruning:



MaxDist(e,o) > MinDist(e,q)
=> prune e nicht!

- Algorithmus: Filter-Schritt (Skizze)
  – 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 NN-Anfrage berechnet werden
- Beispiel



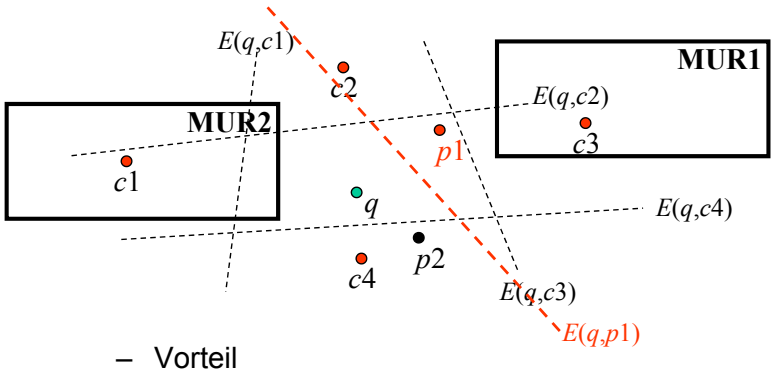
Bisherige Kandidaten:

{$c1$, $c2$, $c3$, $c4$}

Inhalt des Rankings (ungeordnet):

MUR1: nicht verfeinern
MUR2: verfeinern
$p1$: verfeinern
$p2$: nicht verfeinern

- Verfeinerung von $p1$
  - » Streiche $c2$ und $c3$ aus Kandidatenliste (liegt nun hinter $E(q,p1)$)
  - » MUR2 muss weiterhin verfeinert werden



Bisherige Kandidaten:
{c1, c4, p1}

Inhalt des Rankings (ungeordnet):
MUR2: verfeinern

- Vorteil
  - » k kann beliebig sein (Ausschlusskriterium: Objekt/Seite muss hinter k Ebenen liegen
  - » Keine vorberechneten Distanzen, daher keine Update-Problematik und bessere Speicherkomplexität
- Nachteil
  - » Nur für Vektordaten
  - » Teurer Verfeinerungsschritt (eine NN-Anfrage pro Kandidat)
  - » Teilweise komplexe Ebenenverwaltung

- 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):

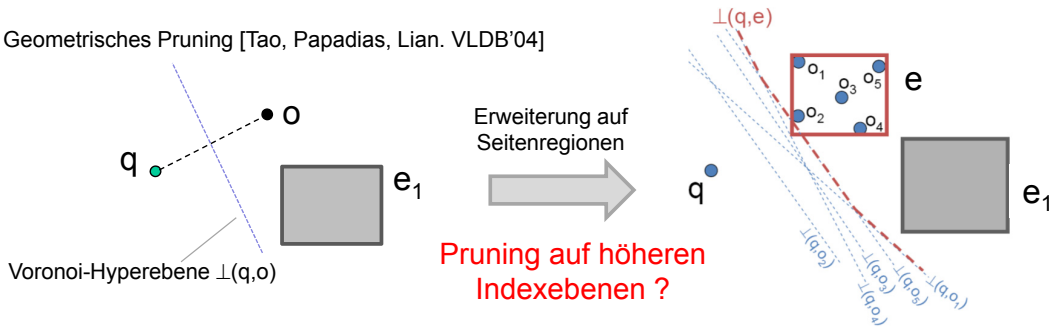
Algorithm TPL-filter(q) /* q is the query point */
1. initialize a min-heap H accepting entries of the form (e, key)
2. initialize sets S_{cnd}=∅, S_{rfn}=∅
3. insert (R-tree root, 0) to H
4. while H is not empty
5. (e, key)=de-heap H
6. if (**trim**(q, S_{cnd}, e)=∞) then S_{rfn}=S_{rfn}∪{e}
7. else // entry may be or contain a candidate
8. if e is data point p
9. S_{cnd}=S_{cnd}∪{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)≠∞) then insert (p,$dist(p,q)$) in H
13. else S_{rfn}=S_{rfn}∪{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 ($mindist(N_i^{resM}, q)$=∞) then S_{rfn}=S_{rfn}∪{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]

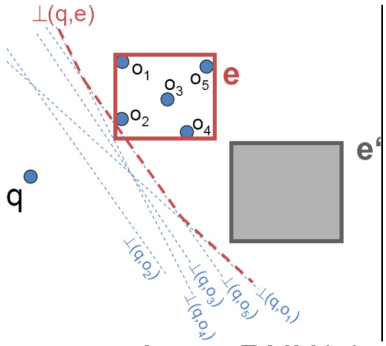
- Weitere Eigenschaft der Geometrischen RNN-Suche
  - Ausschluß (Pruning) von Punkten/Seiten nur aufgrund anderer Punkte
  - Ausschlußkriterium könnte auch vollständig auf Directory-Ebene angewandt werden, aber wie?
- Erweiterung des geometrischen Pruning-Konzepts auf Basis von Voronoi-Hyperebenen auf Seitenregionen



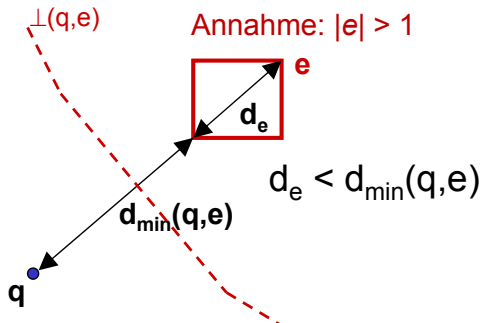
- Frage: Effiziente Repräsentation von Voronoi-Hyperebenen zwischen einem Punkt und einer Seitenregion ?
- Idee: Konservative Approximation der Hyperebenen
  Eigenschaft: ∀p∈⊥(q,e): dist(q,p) = MaxDist(e,p)

– Erweiterung der Geometrischen RNN-Suche (gilt nur für Vektordaten!!!) [Kriegel, Kröger, Renz, Züfle: SSDBM, 2009]
  – Ziel: Pruning auf höheren Indexebenen
  – Definition von Hyper-Ebenen zwischen Anfragepunkt und Seitenregion e
  – Bilde konservative Approximation $\mathcal{H}$ aller Hyperebenen bzgl. aller Punkte innerhalb der Seitenregion e
  – Anzahl der konservativ approximierten Hyperebenen kann zum Ausschluß von Seitenregionen (Self/Mutual Pruning) verwendet werden (insbesondere auch für RkNN-Suche mit k>1)

Seite e schließt Seite e‘ aus (k=1):



Seite e schließt sich selbst aus (k=1):

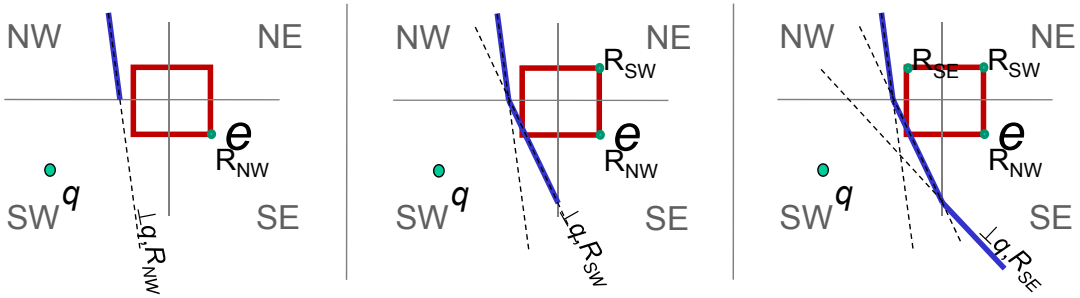


$\forall o \in e'$: $o \notin RNN(q)$, da $\exists o_i \in e$: e’ hinter Hyperebene $\perp(q,o_i)$

- Berechnung der konservativen Approximation aller Hyperebenen zwischen Anfragepunkt q und Seitenregion e
  – Aufsplittung des Datenraums in 2^d Partitionen orientiert am Mittelpunkt der Seitenregion E (z.B. 4 Partitionen NW, NE, SE und SW in 2D Raum)
  – Für jede Partition P: Wähle Referenzpunkt $R \in e$, sodaß gilt:

    $\forall p \in P, \forall e \in E: d(p,e) \leq d(p,R)$

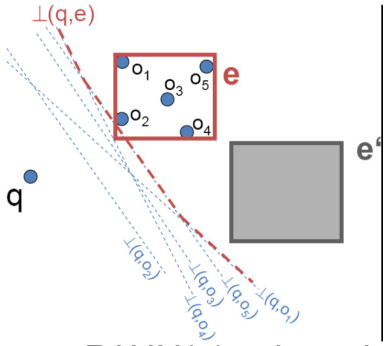
    Bemerkung: Referenzpunkt eindeutig durch die gewählte Partitionierung
  – Für jede Partition P bilde Hyperebene zwischen q und Referenzpunkt zu P



  – Die konservative Hyperebenen-Approximation wird aus 2^d-1 Hyperebenen gebildet

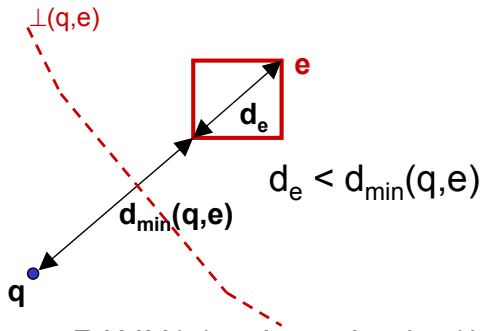
- Erweiterung der Geometrischen R*k*NN-Suche für k > 1
  - Verwende aR-Baum anstatt R-Baum, d.h. R-Baum mit zusätzlicher Angabe der Anzahl der Punkt-Objekte |e| die innerhalb einer Seitenregion e organisiert werden.
  - Menge der Punkt-Objekte auf die sich eine konservative Approximation $\mathcal{H}$ bezieht ist zur Anfragezeit bekannt.
    => Anzahl N der Objekte, die näher an einem Objekt o bzw. einer Seitenregion e‘ sind als der Anfragepunkt q lässt sich einfach ermitteln
  - Prune Seitenregion e‘ (bzw. e‘ bei self pruning) falls N > k (bzw. N-1 > k)

Seite e schließt Seite e‘ aus (k>1):



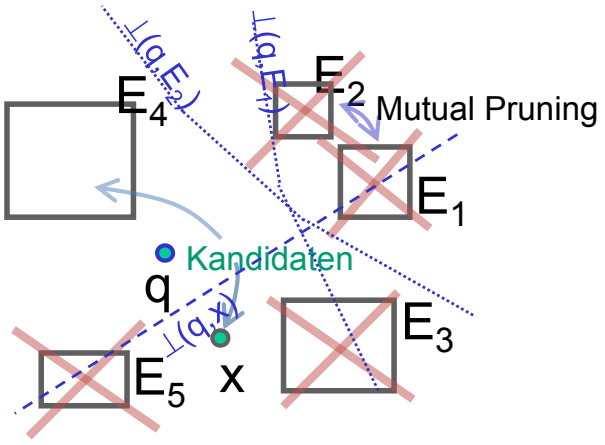
∀o∈e’: *o* ∉ R*k*NN(*q*), where *k* ≤ |e|

Seite e schließt sich selbst aus (k>1):



∀o∈e: o ∉ R*k*NN(q), where *k* ≤ |e-1|

- Beispiel:



- Bemerkung zum „Optimalen“ Pruning:
  - Die Suche mittels Mutual-Pruning ohne Vorberechnung ist weniger selektiv als mit vorberechneten NN-Distanzen
    ==> schlechteres Pruning-Verhalten als bei (reinen) Self-Pruning-Methoden

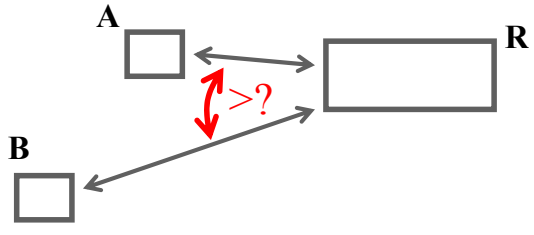
- Eigenschaften der Geometrischen R*k*NN-Suche
  - Vorteile:
    - » Vollständige Flexibilität bzgl. *k*
    - » Keine Zusätzliche Kosten für Änderungen im Index (Update-Kosten)
    - » Pruning-Filter selektiver als bei Min/Max-Dist-Ansatz
  - Nachteile:
    - » 2^d Hyperebenen pro Seitenregion müssen materialisiert werden => Overhead der Ebenenverwaltung (schlechte Performanz in höher-dimensionalen Räumen)
    - » Test ob Seitenregion geprunt werden kann ist teuer (2^d Ebenen-Pruning-Tests)
- Fazit:
  - Min/Max-Dist-Ansatz hat geringeres Pruningpotential als Geometrisches Pruning
  - Geometrisches Pruning eignet sich nur für niedrig-dimensionale Räume
- Gewünscht:
  - Verfahren mit „optimalen“ Pruningpotential, das auch für hoch-dimensionale Räume effizient funktioniert

  → Welche Beziehung gilt zwischen dem Min/Max-Dist-Basierten Ansatz und dem Geometrischen Pruning Ansatz?

- Generelle Problemstellung:

  Gegeben:

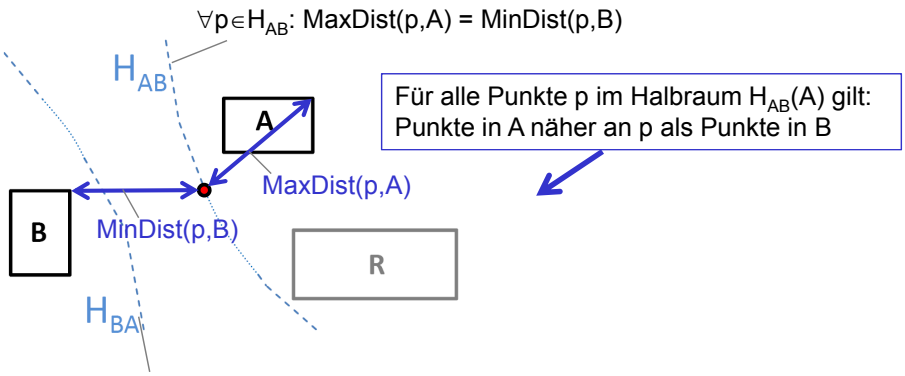
  Drei Punktobjekt-Approximationen A, B und R gegeben als achsenparellele Rechtecke



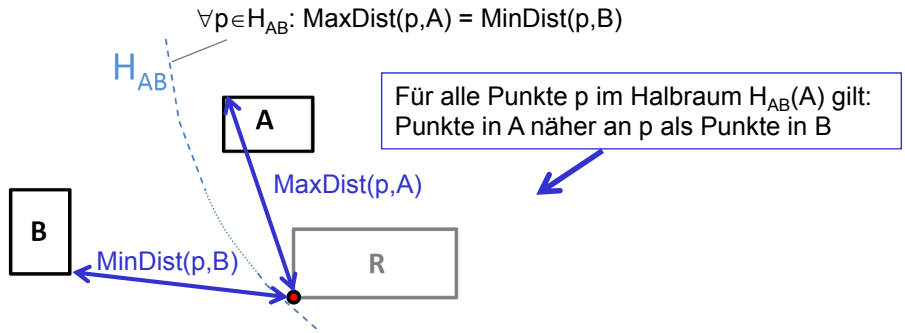
  Frage:

  Wie kann man effizient bestimmen ob die Objekte in R näher an den Objekten in A oder näher an den Objekten in B liegen

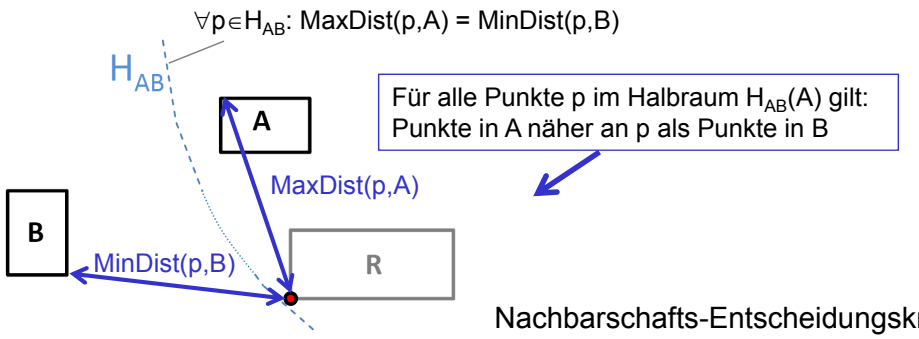
  → Nachbarschafts-Entscheidungskriterium [SIGMOD 10]



- Beobachtung:
  - Halbraum $H_{AB}(A)$ in der A liegt ist konvex
  - → alle Eckpunkte von R liegen in $H_{AB}(A)$ => R liegt vollständig in $H_{AB}(A)$



- Beobachtung:
  - Halbraum $H_{AB}(A)$ in der A liegt ist konvex
  - → alle Eckpunkte von R liegen in $H_{AB}(A)$ => R liegt vollständig in $H_{AB}(A)$
- Idee:
  - Finde Eckpunkt p von R bei dem MaxDist(p,A)-MinDist(p,B) den größten Wert hat
  - Falls dieser Wert kleiner 0 (d.h. p in $H_{AB}(A)$)
  - → Alle Ecken von R in $H_{AB}(A)$ => Region R vollständig in $H_{AB}(A)$

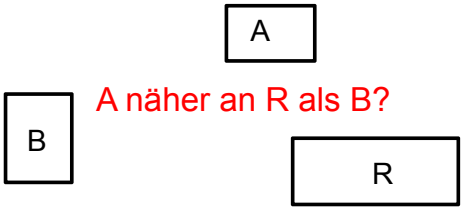


Nachbarschafts-Entscheidungskriterium:
Objekte in R näher an A falls

$$\sum_{i=1}^{d} \max_{r_i \in \{R_i^{min}, R_i^{max}\}} (MaxDist(A_i, r_i)^2 - MinDist(B_i, r_i)^2) < 0$$

- Beobachtung:
  - Halbraum $H_{AB}(A)$ in der A liegt ist konvex
  - → alle Eckpunkte von R liegen in $H_{AB}(A)$ => R liegt vollständig in $H_{AB}(A)$
- Idee:
  - Finde Eckpunkt p von R bei dem MaxDist(p,A)-MinDist(p,B) den größten Wert hat
  - Falls dieser Wert kleiner 0 (d.h. p in $H_{AB}(A)$)
  - → Alle Ecken von R in $H_{AB}(A)$ => Region R vollständig in $H_{AB}(A)$

- Herleitung
  - Idee: Umformung der Distanzrelation:



$$\forall a \in A, b \in B, r \in R : dist(a,r) < dist(b,r)$$

$$\Leftrightarrow \forall r \in R : MaxDist(A,r) < MinDist(B,r)$$ ← konservative Distanzabschätzung zu A und B

$$\Leftrightarrow \forall r \in R : \sqrt[p]{\sum_{i=1}^{d} MaxDist(A_i,r_i)^p} < \sqrt[p]{\sum_{i=1}^{d} MinDist(B_i,r_i)^p}$$

$$\Leftrightarrow \forall r \in R : \sum_{i=1}^{d} MaxDist(A_i,r_i)^p - \sum_{i=1}^{d} MinDist(B_i,r_i)^p < 0$$

$$\Leftrightarrow \max_{r \in R} \left(\sum_{i=1}^{d} (MaxDist(A_i,r_i)^p - MinDist(B_i,r_i)^p) \right) \leq 0$$

$$\Leftrightarrow \sum_{i=1}^{d} \max_{r_i \in R_i} (MaxDist(A_i,r_i)^p - MinDist(B_i,r_i)^p) < 0$$

Reduzierung des Problems auf die einzelnen Dimensionen

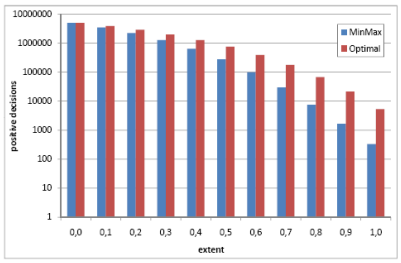
$$\Leftrightarrow \sum_{i=1}^{d} \max_{r_i \in \{R_i^{min}, R_i^{max}\}} (MaxDist(A_i,r_i)^p - MinDist(B_i,r_i)^p) < 0$$

Reduzierung des Problems auf Extremwerte in R_i (pro Dimension nur 2 Werte)

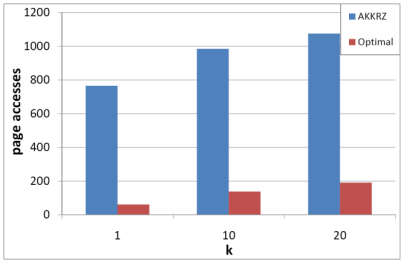
- Vorteil: Berechnungskomplexität O(d) (d = Dimensionalität)

=> geeignet auch für höherdimensionale Räume

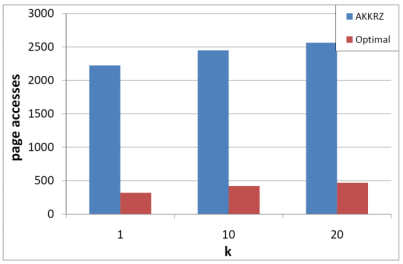
- Experimenteller Vergleich: Min/Max-Dist vs. Optimales Pruning:



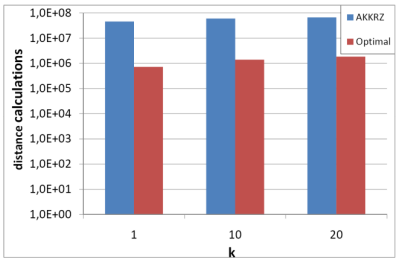
a) Anzahl der erkannten Ereignisse: Objekte in R näher an A als an B
Datensatz (synthetisch): 10 Mil. Triple (A,B,R) von Rechtecken (gleichverteilt im Raum $[0,1]^2$)



b) Anzahl der Seitenzugriffe für RkNN-Anfragen mit Index
Datensatz (synthetisch): 100K Punktobjekte (5D, gleichverteilt)



c) Anzahl der Seitenzugriffe für RkNN-Anfragen mit Index (I/O-Kosten)
Datensatz (real): 581K Punktobjekte (10D)



d) Anzahl der Distanzberechnungen für RkNN-Anfragen mit Index (CPU-Kosten), Datensatz (real): 581K Punktobjekte (10D)

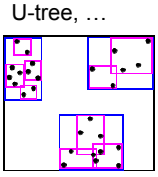
– Weiterer Anwendungsbereiche des Nachbarschaftskriteriums:

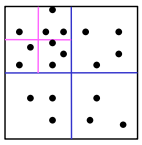
Im Prinzip lässt sich das Nachbarschaftskriterium überall dort einsetzen wo Objekte durch achsenparallele Rechtecke approximiert sind und Nachbarschaftsbeziehungen relevant sind

Beispiele:

Spatial Index:

R-tree, X-Tree, U-tree, …



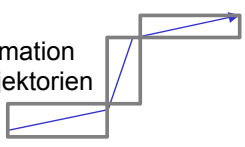


Quadtree

Anfragen:
- kNN, RkNN
- continuous kNN (RkNN)
- probabilistic kNN (RkNN)
- inverse ranking
- group Nearest Neighbor
- clustering
- usw.

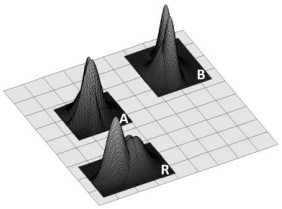
Spatio-Temporal Data:

Approximation von Trajektorien



Unsichere Daten:

Approximation von unsicheren Objekten



– Zusammenfassung

	Verfahren	Vorteile	Nachteile
self-pruning	**RNN-Tree**	Sehr gute Performanz, da keine Verfeinerung nötig	*k* fix; nur für Vektordaten; Updateproblematik; wenig selektiv bei normalen NN-Queries
self-pruning	**RdNN-Tree**	Sehr gute Performanz, da keine Verfeinerung nötig; auf allgemein metrische Daten erweiterbar	*k* fix; Updateproblematik
self-pruning	**MRkNNCoP-Tree**	variables *k (mit Einschränkung)*; für allgemein metrische Daten	Verfeinerung nötig, Updateproblematik
mutual-pruning	**Min/Max-Dist-basierte RkNN Suche**	variables *k*; geringe Kosten, für metrische Daten, erlaubt Pruning auf Directory-Ebene	schlechtere Filterselektivität
mutual-pruning	**Geometrische (Voronoi-basierte) RkNN Suche**	variables *k*;	Nur für Vektordaten; Verfeinerung nötig
mutual-pruning	**Erweiterte geometrische RkNN Suche**	variables *k*; erlaubt Pruning auf Directory-Ebene; optimale Filterselektivität	Nur für Vektordaten; teure Verwaltung der Hyperebenen
mutual-pruning	**optimal-pruning-basierte RkNN Suche**	variables *k,* geringe Kosten erlaubt Pruning auf Directory-Ebene; optimale Filterselektivität;	Nur für Vektordaten;

2.6 Skylineanfrage

- Gegeben:

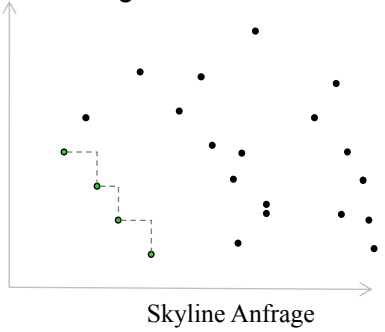
  Menge von Objekten mit positiven Attributwerten $a_i \in \Re^+$ $(1 \le i \le d)$.
- Gesucht:

  Ergebnis enthält alle Objekte die von keinem anderen Objekt *dominiert* werden, d.h. kein anderes Objekt ist „besser“ bzgl. aller Attribute.
- Formal:

  Vektor $x=(x_1,..,x_d) \in \Re^d$ dominiert Vektor $y=(y_1,..,y_d) \in \Re^d$ wenn gilt:

  $dom(x,y) \Leftrightarrow (\forall\ 1 \le i \le d\ x_i \le y_i) \wedge (\exists\ 1 \le i \le d\ x_i < y_i)$

  SkyLine(*DB*) = $\{o \in DB \mid \forall p \in DB : \neg dom(p,o)\}$



- Basisalgorithmus (sequential scan):

```
Skyline-SeqScan(DB)
  result = ∅;
  FOR i=1 TO n DO
    o = getObject(i);
    IF o not-dominated-by(any other object in DB)THEN
      result := result ∪ o;
  RETURN result;
```

zusätzlicher sequentieller scan über DB

- Skyline-Anfrage Varianten: [Papadias et al., ToDS 2005]

  zum Beispiel:

  » *Constrained Skyline*:
  Ausgabe aller Skyline-Objekte deren Attribute zusätzlich eine bestimmte Bedingung (z.B. Hotelkosten zwischen 50 und 80 Euro) erfüllen.

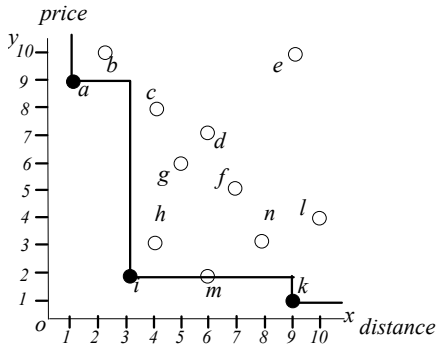
  » *Ranked Skyline*:
  Ausgabe der *k* ersten Skyline-Objekte, die bzgl. einer Präferenzfunktion (Kostenfunktion über alle Attribute) sortiert ausgegeben werden.

  » weitere Varianten:
  Group-by Skyline, *Dynamic Skyline*, *K-dominating Queries*, etc.

2.6.2 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:

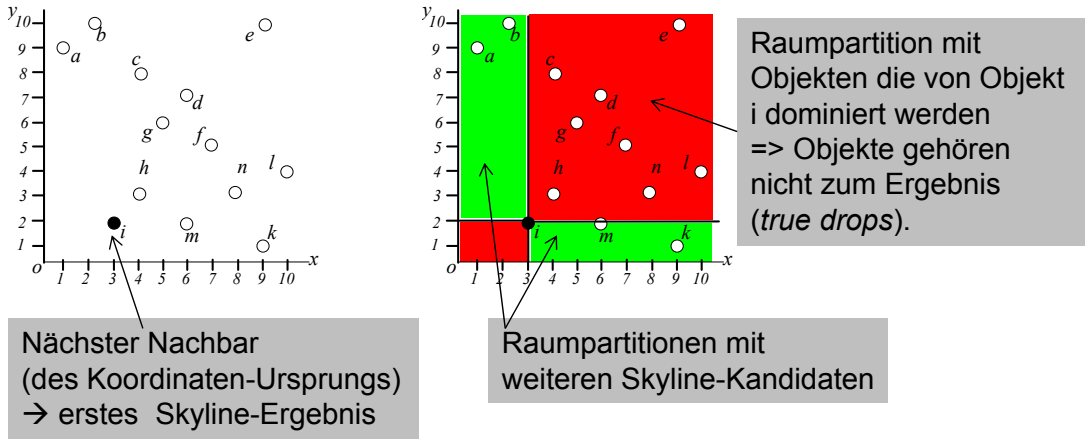
      - wenig Seitenzugriffe und Dominanzüberprüfung mittels Index (R-Baum).
      - Erste Ergebnisse werden frühzeitig ausgegeben durch iterative Verarbeitung.

– Nächste-Nachbarn-Skyline (NNS) Algorithmus:

[Kossman et al., VLDB 2002]

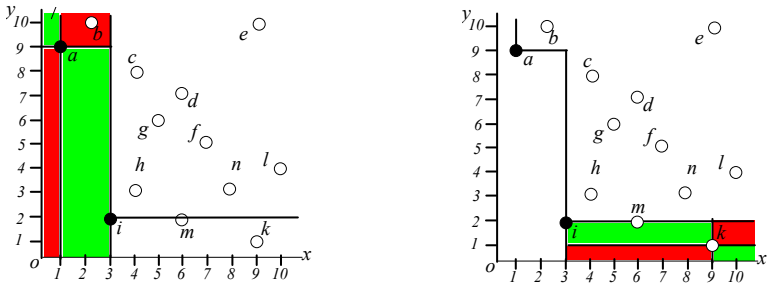
Prinzip:

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

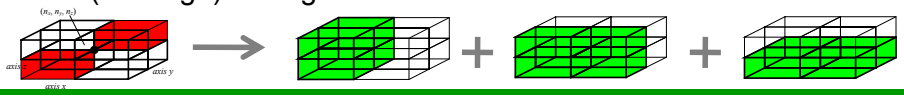


- Nächste-Nachbar-Suche kann durch R-Baum Index beschleunigt werden (z.B. Alg.: k-NN-Index-HS).

- 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:

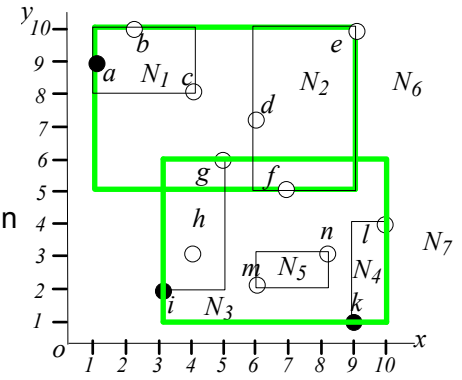
[Papadias et al., ToDS 2005]

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

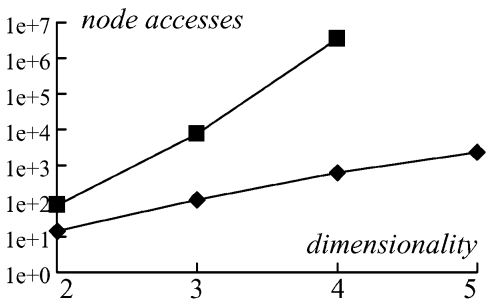


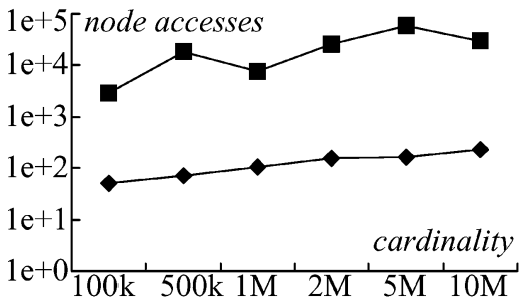
- Algorithmus:

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





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