7 Raumzugriffsstrukturen

Ausgedehnte Objekte

In Nichtstandard-Datenbankssytemen, wie z.B. in Geographischen Datenbanksystemen, werden nicht nur Punktobjekte, sondern auch *ausgedehnte Datenobjekte* gespeichert:

- *Linien*
- *Linienzüge*
- *Flächen.*

Eine für Geographische Datenbanken besonders wichtige Objektklasse stellen *Flächen mit Löchern* dar. Sie werden durch *einfache Polygone mit Löchern (EPL)* repräsentiert.

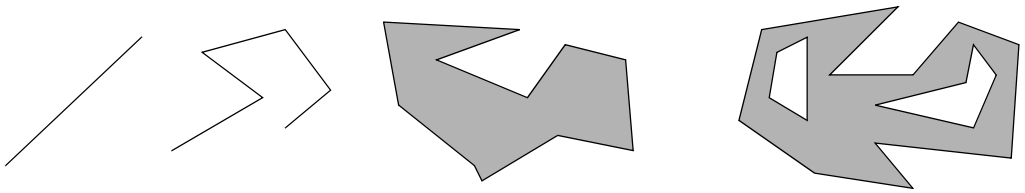
7.1 Grundlagen

- ***Einfaches Polygon***

  Polygon, bei dem sich kein Paar nicht-aufeinanderfolgender Kanten schneidet.
- ***Einfaches Polygon mit Löchern***

  Einfaches Polygon, aus dem disjunkte, einfache Polygone herausgeschnitten sind.

Im weiteren wird insbesondere auf die Verwaltung und Speicherung von einfachen Polygonen mit Löchern eingegangen.



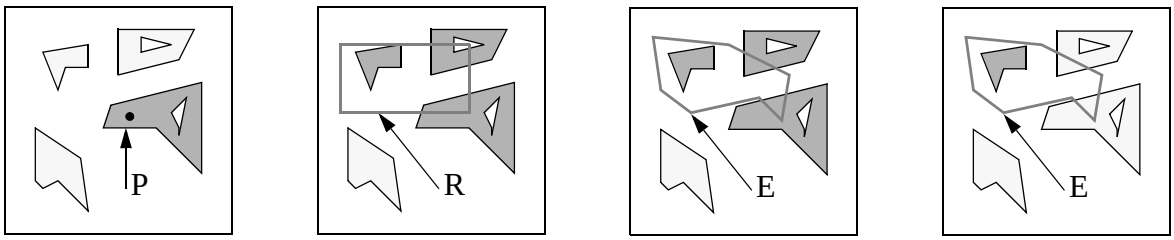
Linie Linienzug Fläche (einfaches Polygon) Fläche mit Löchern (EPL)

Anfragen

Eine Indexstruktur soll Anfragen effizient unterstützen. Für die Objektklasse der EPL gibt es eine Reihe von wichtigen *Basisanfragen*:

- *Point Query:* Gegeben ein Punkt P; finde alle EPL, die P enthalten.
- *Window Query:* Gegeben ein Rechteck R; finde alle EPL, die R schneiden.
- *Region Query:* Gegeben ein EPL E; finde alle EPL, die E schneiden.
- *Enclosure Query:* Gegeben ein EPL E; finde alle EPL, die in E enthalten sind.
- *Containment Query*: Gegeben ein EPL E; finde alle EPL, die E vollständig enthalten.

Auf diesen Basisanfragen können dann komplexere Anfragen in einem Geoinformationssystem aufsetzen.



Point Query Window Query Region Query Enclosure Query

Approximationen

Einfache Polygone mit Löchern (die ja eine beliebige Komplexität besitzen können) lassen sich nicht direkt mittels Indexstrukturen abgespeichern (z. B. weil man sonst keine minimale Anzahl von Einträgen pro Seite garantieren könnte).

⇒ Eine *Approximation* der EPL über eine einfachere Objektklasse wird notwendig.

Eine solche Approximation sollte *konservativ* sein, d.h. das approximierte Objekt sollte vollständig in der Approximation enthalten sein. Eine Reihe von Approximationen wurde vorgeschlagen, die in Abschnitt 7.4 behandelt werden. Im folgenden betrachten wir die einfachste dieser Approximationen, nämlich das *achsenparallele minimal umgebende Rechteck* (MUR).

Zweistufige Anfragebearbeitung [BHKS 93]

Die Anfragebearbeitung von approximiert organisierten Objekten muß zweistufig ablaufen:

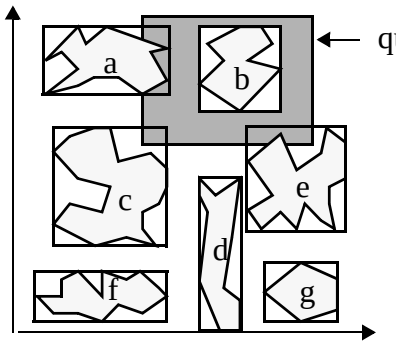
1. Filterschritt

Zunächst werden die Objekte über die Indexstruktur eingelesen, die gemäß der Approximation die Anfrage erfüllen. Man erhält somit eine *Kandidatenmenge*, die Obermenge zur eigentlichen Lösungsmenge ist.

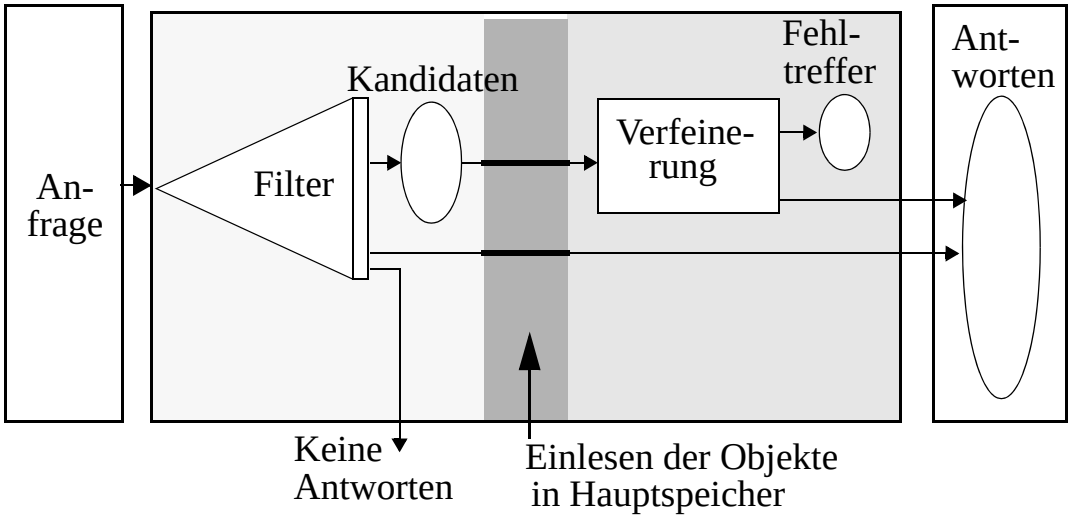
2. Verfeinerungsschritt

Im zweiten Schritt wird die exakte Darstellung der Kandidaten herangezogen und untersucht, ob das EPL tatsächlich die Anfrage erfüllt.

Beispiel: Window-Query



→ a und b sind sicher *Antworten*
→ f, d und g sind sicher keine Antworten
→ c und e sind *Kandidaten*
→ c ist ein *Fehltreffer* (*false hit*, d. h. ein Kandidat, der keine Antwort ist)



7.2 Von Punkt- zu Rechteckzugriffsstrukturen

Mit den bisher vorgestellten Indexstrukturen *(Punktzugriffsstrukturen)* können nicht ohne weiteres achsenparallele Rechtecke abgespeichert werden, sie können jedoch als Basis für *Raumzugriffsstrukturen* benutzt werden. Es gibt drei Techniken, um von Punkt- zu Rechteckzugriffsstrukturen zu kommen (siehe [SK 88]):

- Punkttransformation
- Clipping
- Überlappende Regionen.

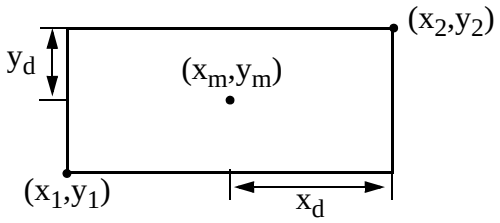
Punkttransformation

Bei der *Punkttransformation* werden die minimal umgebenden n-dimensionalen Rechtecke in 2n-dimensionale Punkte überführt und in einer 2n-dimensionalen Punktzugriffsstruktur abgespeichert. Es lassen sich zwei Transformationen unterscheiden:

- *Mittentransformation*

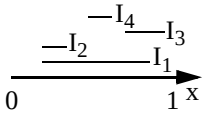
  Das Rechteck wird durch den Mittelpunkt (x_m,y_m) und die jeweilige halbe Ausdehnung x_d und y_d beschrieben.
- *Eckentransformation*

  Das Rechteck wird durch diagonal gegenüberliegende Eckpunkte (x_1,y_1) und (x_2,y_2) repräsentiert.

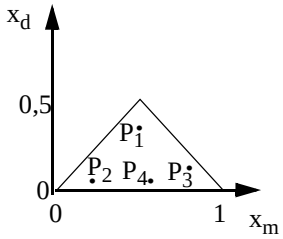


Transformation des Datenraumes und der Anfragebereiche (für 1-dimensionale Intervalle):

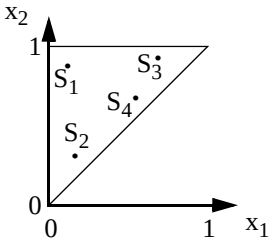
Transformation von Intervallen I_j



in 2-dim. Punkte:

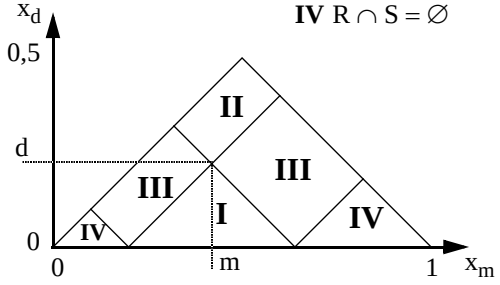


$P_j = (x_m,x_d)$
(Mittentransformation)



$S_j = (x_1,x_2)$
(Eckentransformation)

I $R \subseteq S$
II $R \supseteq S$
III $R \cap S \neq \emptyset$
IV $R \cap S = \emptyset$



Anfrageräume zur Suche von Intervallen R bzgl. eines Intervalls S = (m,d) (Mittentransformation)

Eigenschaften

- Bei der Eckentransformation liegen die meisten Daten auf einer Diagonalen durch den Datenraum.

  ⇒starke Korrelation der Daten
- Bei der Mittentransformation verlaufen die Anfragegrenzen nicht mehr orthogonal zur Datenraumpartitionierung.

  ⇒komplexere Implementierung der Anfrage

  ⇒mehr Datenseiten werden geschnitten
- Die geometrischen Verhältnisse gehen bei der Punkttransformation durch die Einbeziehung der Rechtecksausdehnung verloren.

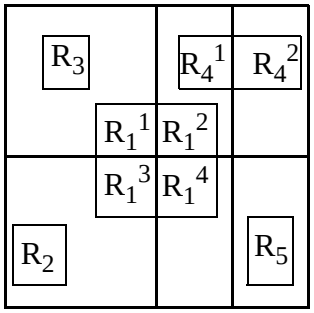
  ⇒Raumbezogene Anfragen verlieren dadurch deutlich an Effizienz.
- Die Punkttransformation ist auf jede Punktzugriffsstruktur anwendbar. So müssen z.B. die Einfüge- oder Lösch-Operationen nicht geändert werden.

Anwendung

Der Leistungsvergleich in Abschnitt 6.1 zeigt, daß der *Buddy-Baum* für die Punkttransformation gut geeignet ist (siehe Verteilung “Diagonal”). Der *LSD-Baum* wurde speziell als Indexstruktur für mehrdimensionale Punkte entwickelt, die durch Punkttransformation aus Rechtecken entstanden sind.

Clipping

Die Idee ist, daß ein Rechteck in jede Datenregion eingefügt wird, die es schneidet.



5 Rechtecke
9 Datensätze

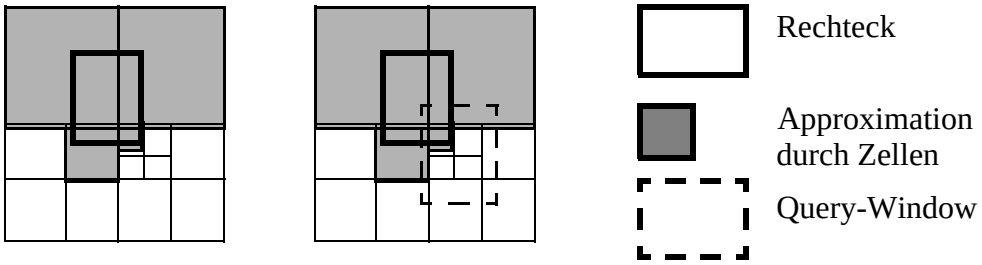
Eigenschaften:

- Die Zahl der Datensätze in der Indexstruktur steigt stärker als die Zahl der gespeicherten Rechtecke. Dieses gilt insbesondere, wenn die Rechtecke im Verhältnis zum Datenraum groß sind oder die Partitionierung schon sehr fein geworden ist.
- Die Indexstruktur sollte Überlaufseiten zulassen, da ein Gebiet, in dem sich mehr Rechtecke überlappen, als in eine Datenseite passen, nicht weiter partitioniert werden kann.
- Einfügen und Löschen werden erheblich aufwendiger.
- Bereichsanfragen degenerieren in der Leistung aufgrund der hohen Zahl von mehrfach eingelesenen identischen Rechtecken.

Anwendung

Mit Hilfe des Clipping lassen sich Rechtecke folgendermassen in einem *PR-Quadtree* abspeichern. Es werden die Zellen des PR-Quadtrees berechnet, die ein gegebenes Rechteck minimal umgeben.

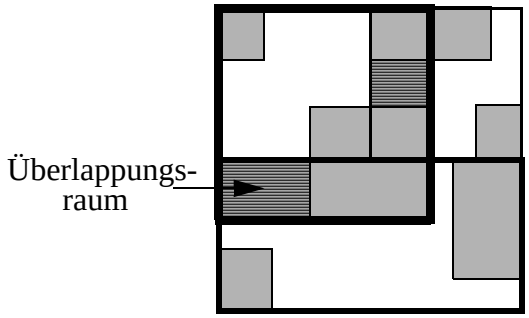
Alle diese Zellen erhalten einen Eintrag für das Rechteck.



In obigem Beispiel wird das gespeicherte Rechteck für die gegebene Window-Query viermal gefunden.

Überlappende Regionen

Die Kernidee der Technik der überlappenden Regionen ist, daß die Partitionierung des Datenraumes nicht mehr disjunkt ist. Somit können sich Seitenregionen überlappen und ein Clipping der Rechtecke wird überflüssig.



Hauptproblem: Überlappung der Directoryregionen.

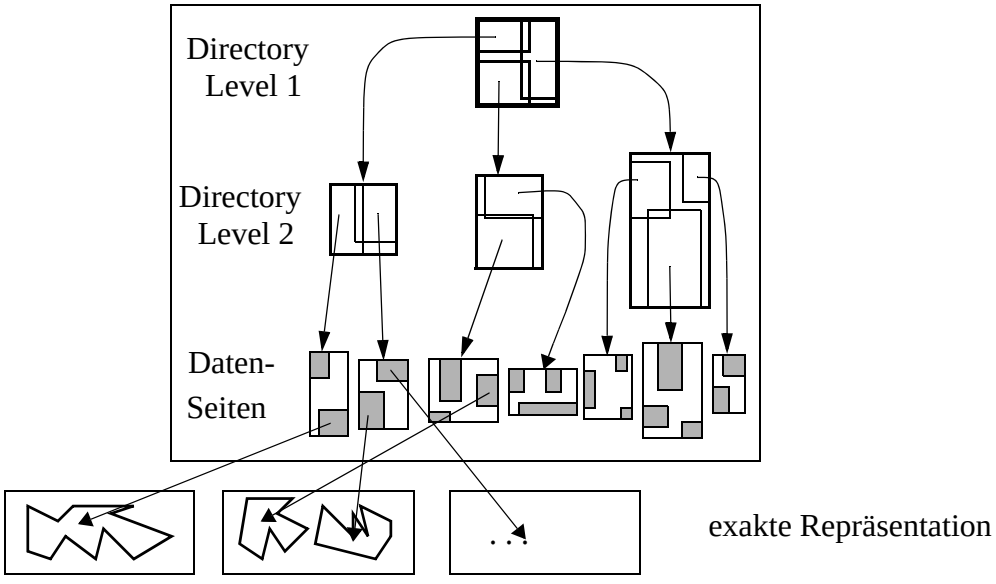
- Fällt eine Anfrage in einen Überlappungsraum, müssen mehrere Pfade im Suchbaum untersucht werden.

  $\Rightarrow$ Die Überlappung sollte möglichst klein gehalten werden.

Anwendung

Die Technik überlappender Regionen wird beim *R-Baum* und dessen Varianten verwendet.

Beispiel:



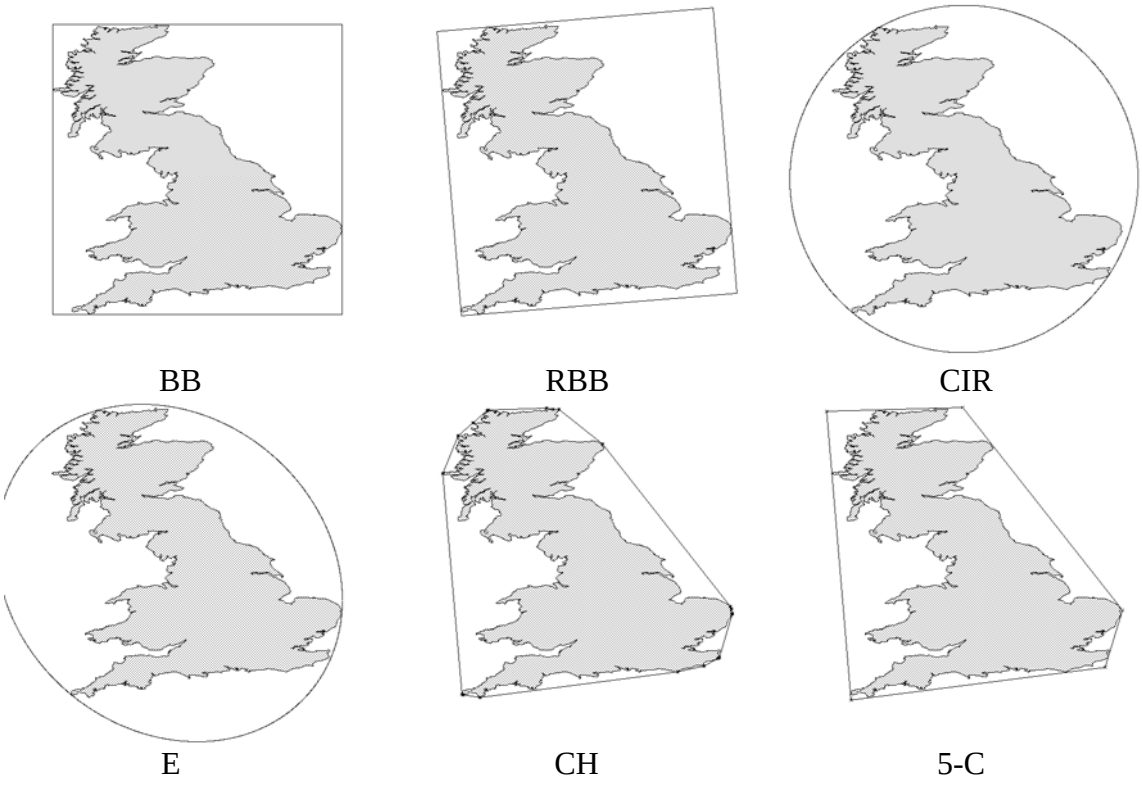
7.3 Approximation

Idee

- möglichst viele Objekte, die keine Antworten sind, ohne Zugriff auf ihre exakte Repräsentation ausscheiden
- den aufwendigen Verfeinerungsschritt nur für möglichst wenige Objekte durchführen

Verschiedene Approximationen

- *achsenparalleles, minimal umgebendes Rechteck (BB).*
- *minimal umgebender Kreis (CIR)* [Oos 90].
- *konvexe Hülle (CH)* [Gün 89].
- *gedrehtes, minimal umgebendes Rechteck (RBB)*
- *minimal umgebende Ellipse (E)*
- *minimal umgebende konvexe Vier- und Fünfecke (4-C bzw. 5-C)*



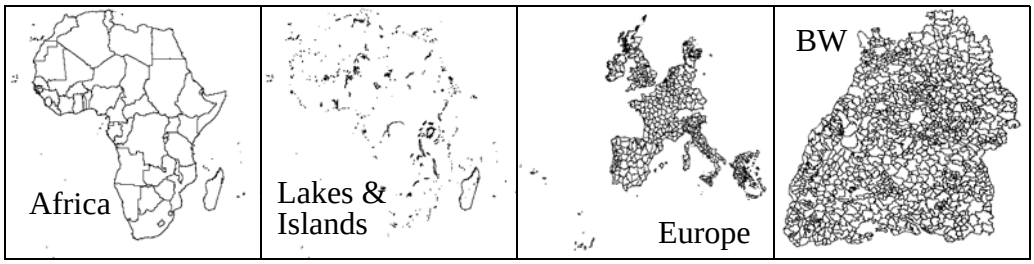
Approximationsgüte

$$Q_{\text{Appr}} = \frac{A(Appr[O])}{A(O)} \times 100\%$$

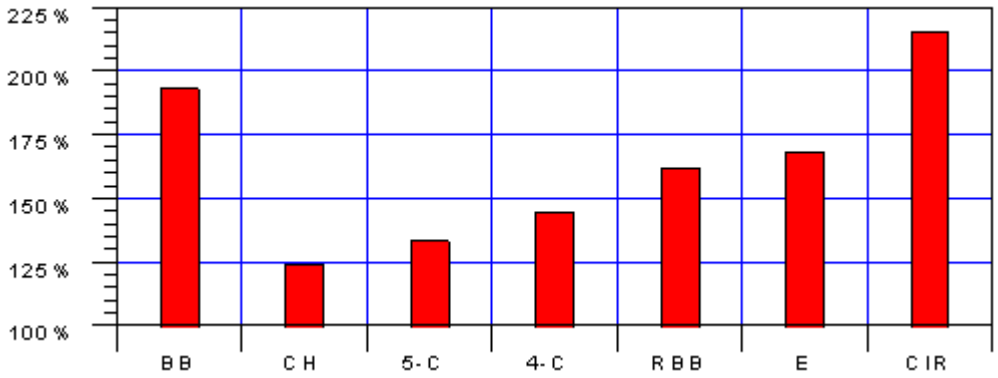
- A: Fläche, O: Objekt, Appr[O]: Approximation von O
- Q_{Appr} = 100% wenn Approximation optimal, d. h. Appr[O] = O
- je kleiner Q_{Appr}, umso besser die Approximation

Experimentelle Untersuchung der Approximationsgüte [BKS 93]

Verwendete Daten



Durchschnittsergebnisse über alle Landkarten



Interpretation

- CH hat im Unterschied zu 5-C beliebig viele Parameter, trotzdem approximiert 5-C fast gleich gut.
- RBB: 1 Parameter mehr als BB ⇒ 31% Verbesserung.
- 5-C: 6 Parameter mehr als BB⇒ 60% Verbesserung.

Vorteile genauerer Approximationen

- Zugriff auf die exakte Repräsentation für weniger Objekte (weniger Seitenzugriffe)
- Verfeinerungsschritt für weniger Objekte (weniger CPU-Zeit)
- z. B. Point-Queries: die Verbesserung der Verarbeitungszeit ist proportional zur Verbesserung der Approximationsgüte

Nachteile genauerer Approximationen

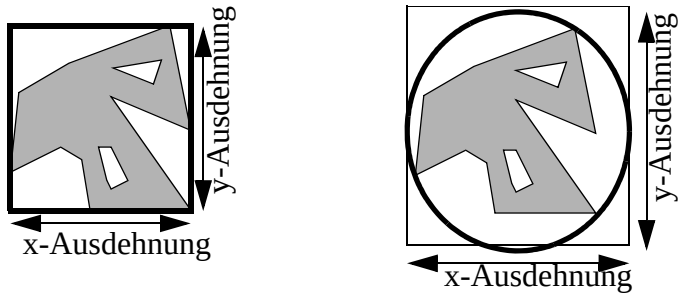
- höherer Speicherplatzbedarf auf Datenseiten, z. B. bei Annahme von 4 Bytes pro Parameter:

	CH	5-C	4-C	RBB	E	BB	CIR
Q_{Appr}	124	133	144	162	168	193	215
Speicherplatzbedarf in bytes	32 - 640	40	32	20	20	16	12

⇒ mehr Datenseiten nötig

- größere Ausdehnung der Approximationen

  z. B.



  $Ausdehnung_{Appr}$ = x-Ausdehnung x y-Ausdehnung, $Ausdehnung_{BB}=Ausdehnung_{CH}=100\%$

	5-C	4-C	RBB	E	CIR
Ausdehnung $_{Appr}$ (%)	121	144	151	122	142

$\Rightarrow$ größere Datenseiten-Regionen

- höherer Aufwand für die Berechnung der Approximationen und für Tests auf den Approximationen

Frage:

Wird der Overhead im Filterschritt durch größere Gewinne im Verfeinerungsschritt gerechtfertigt? Siehe die folgende Untersuchung der Performance von Point-Queries mit den betrachteten Approximationen.

Untersuchung von Point-Queries mit Approximationen

- Annahme: die gegebene Approximation ersetzt die BB-Approximation.
- Annahme: der Verfeinerungsschritt ist pro Objekt c-mal so aufwendig wie der Filterschritt (c > 2 für alle Testdaten).
- Leistungsmaß ist die Antwortzeit.

	Reduktion # Fehltreffer	**Overhead im Filterschritt**		**Break-Even Punkt für c =**		**Speicher-platz-bedarf** [bytes]
		100% erfolgreiche Query	60% erfolgreiche Query	100% erfolgreiche Query	60% erfolgreiche Query	
CH	36 %	70 %	62 %	1,94	1,72	32-640
5-C	31 %	44 %	39 %	1,42	1,26	40
4-C	25 %	43 %	38 %	1,72	1,52	32
RBB	16 %	30 %	28 %	1,88	1,75	20
E	13 %	18 %	15 %	1,38	1,15	20
CIR	- 11 %	5 %	6 %	/	/	12

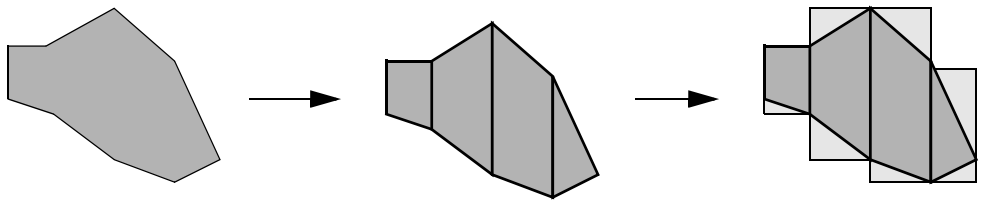
Zusammenfassende Bewertung

- Der Kreis ist nicht geeignet als Approximation.
- Die Ellipse besitzt den niedrigsten Break-Even-Punkt.
- Das 5-Eck liefert den besten Trade-Off zwischen Leistungsverbesserung, Speicherplatzbedarf und Break-Even-Punkt.
- Alle Leistungsverbesserungen wachsen mit c, d. h. mit der Komplexität der Objekte.

7.4 Dekomposition

Idee

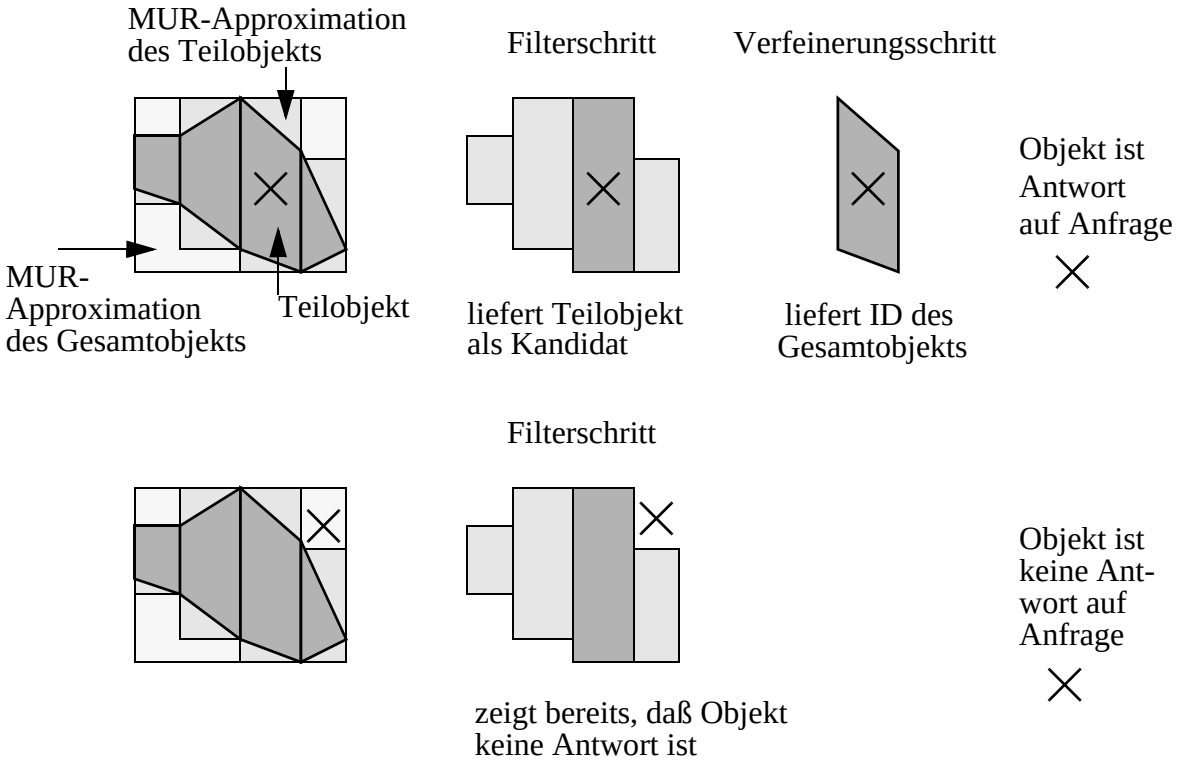
- Ziel: Tests auf der exakten Repräsentation der Objekte (im Verfeinerungsschritt) vereinfachen.
- Beobachtung: für das Ergebnis eines Tests ist i. A. nur ein kleiner Teil des Objekts relevant.
- Vorgehen: Dekomposition der EPL in einfache Teilobjekte und Approximation der Teilobjekte durch minimal umgebende Rechtecke.



Gesamt-Objekt | Teilobjekte | Teilobjekte mit MUR (in Zugriffsstruktur)

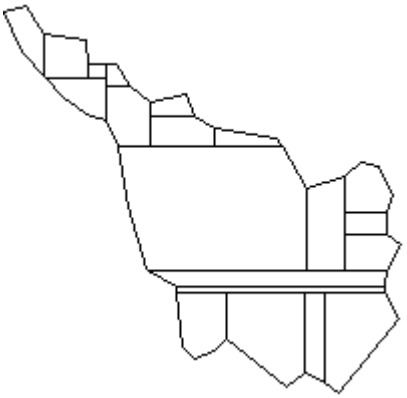
Anfragebearbeitung mit Dekomposition (Point Query)

- Der Filterschritt liefert mit Hilfe einer Raumzugriffsstruktur alle Teilobjekte, deren Approximation den Anfragepunkt enthalten.
- Über einen Algorithmus aus der Computational Geometry wird nun getestet, ob der Punkt tatsächlich im Kandidaten-Teilobjekt liegt.
- Wenn der Test dieses bestätigt, kann das zugehörige Gesamtobjekt in die Antwortmenge der Anfrage mit aufgenommen werden.



Verschiedene Dekompositionen

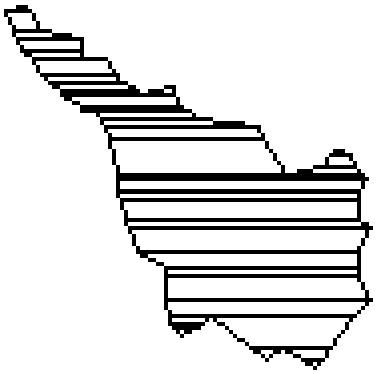
(n = Zahl der Eckpunkte des EPL)



konvexe Dekomposition

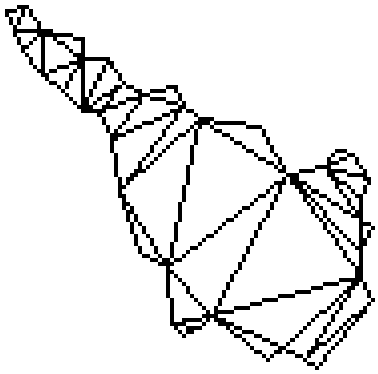
in konvexe Polygone

≈ *n*/2 Komponenten



Dekomposition in Trapeze

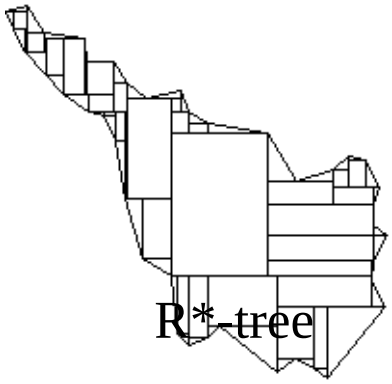
≈ *n* Komponenten



Triangulation

Dekomposition in Dreiecke

≈ *n* Komponenten



Heterogene Dekomposition

in Dreiecke und Rechtecke

≈ 1.8**n* Komponenten

Eigenschaften

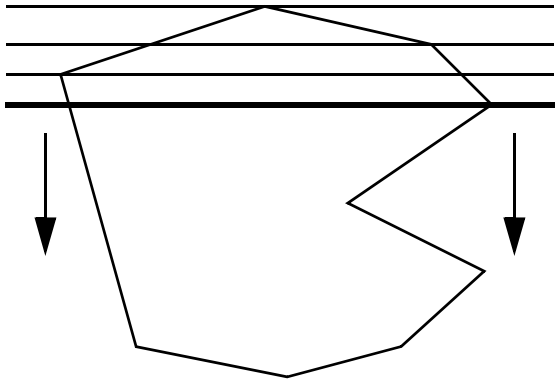
- lineare Anzahl einfacher Komponenten
- Die Typen der Komponenten sind so gewählt, daß sie gut durch MUR approximiert werden können.

Vorteile

- Die Teilobjekte lassen sich durch die MUR besser approximieren als das Gesamtobjekt.
  ⇒ Der Filterschritt arbeitet genauer, so daß sich die Kandidatenmenge verkleinert.
- Es entstehen einfachere Teilobjekte (weniger Kanten).
  ⇒ Der Verfeinerungsschritt kann effizienter durchgeführt werden.

Nachteile

- Rechen-Aufwand für die Dekomposition
  z. B. für die Zerlegung in Trapeze :



Sortiere die Eckpunkte nach Y-Koordinate.

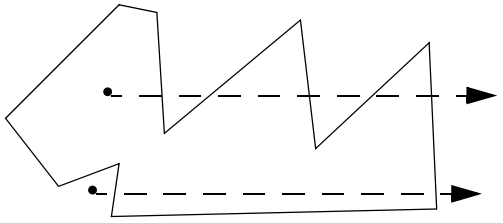
Für jedes Element der sortierten Liste definiere ein Trapez durch das aktuelle und das nächste Listenelement.

⇒ Laufzeit der Dekomposition in Trapeze ist O (n*log n), wobei n die Anzahl der Eckpunkte des EPL bezeichnet.

- Es sind mehrere Teilobjekte für ein EPL zu verwalten und zu speichern. Im obigen Fall der Dekomposition in Trapeze z. B. können es bis zu n-1 Teilobjekte sein.
  ⇒ Größerer Speicherbedarf.
  ⇒ Komplexere Handhabung von Objekten (Objekt-ID, Konsistenz).

Exkurs in die algorithmische Geometrie:

Beispiel: Test, ob Punkt in Polygon liegt

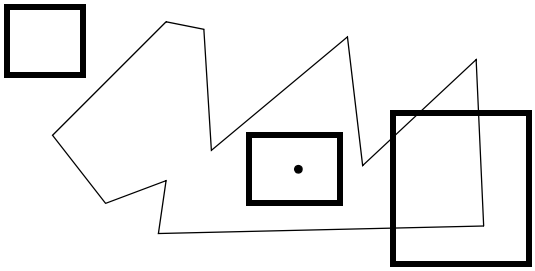


Zähle die Schnittpunkte.

Falls Anzahl
ungerade: Punkt liegt im Polygon
gerade: Punkt liegt nicht im Polygon.

Laufzeit O(n), n = Anzahl der Kanten

Beispiel: Test, ob Window ein Polygon schneidet



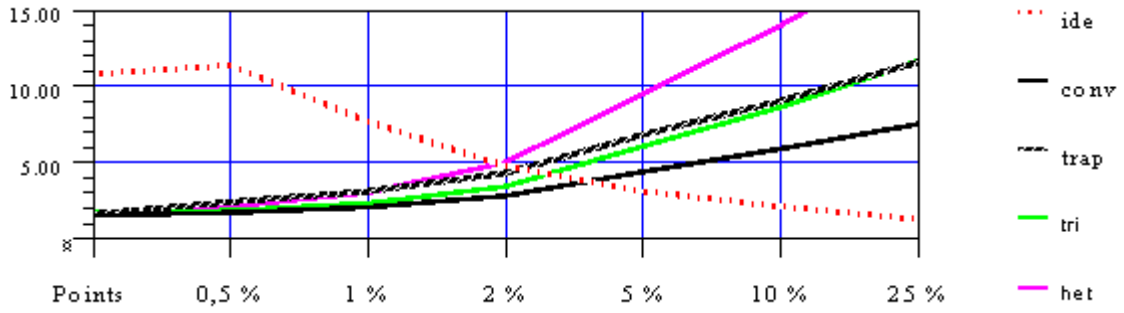
Schneidet eine der Kanten das Window?
Wenn ja: Antwort = ja
Wenn nein:

Nehme einen Punkt des Window.
Liegt er im Polygon?
Wenn ja: Antwort = ja.
Wenn nein: Antwort = nein.

Laufzeit O(n), n = Anzahl der Kanten

Experimentelle Untersuchung von Window-Queries [KHS 91]

(CPU-Zeit in msec pro Antwort)



Größe des Query-Window

Ergebnisse

Dekompositionen mit linearer Anzahl von Komponenten

- sind sehr gut für selektive Queries (kleine Antwortmenge)
- degenerieren für wenig selektive Queries (große Antwortmenge).

Vergleich der bisherigen Ansätze zur Dekomposition

- zwei Ansätze: Identität (d. h. keine Dekomposition der Objekte) und Zerlegung in einfache Komponenten (im folgenden Vergleich z. B. in Trapeze)
- Komplexität der Komponenten: Zahl ihrer Eckpunkte

	Anzahl der Komponenten	Komplexität der Komponenten	Leistung bei Queries: selektive	wenig selektive
Identität	1	n	schlecht	gut
einfache Komponenten	n	4	gut	schlecht

⇒ Benötigt wird eine gute *Balance* zwischen Anzahl und Komplexität der Komponenten.

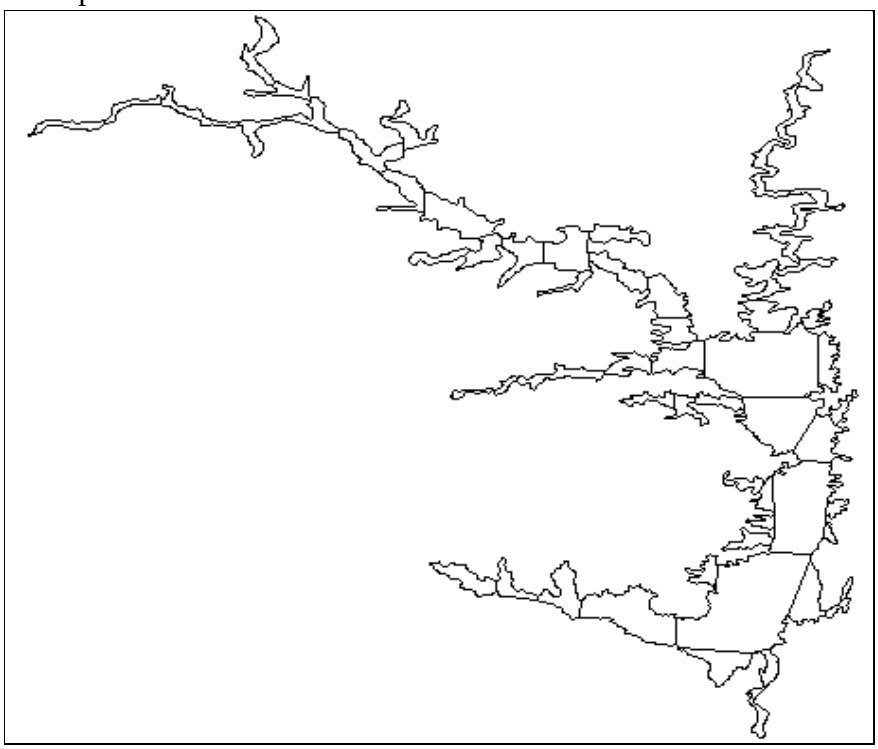
Balancierte Dekomposition [SK 93]

- Siehe obiger Vergleich: das Produkt aus Anzahl der Komponenten und ihrer Komplexität ist O(n).
- Wurzelkriterium für balancierte Dekomposition: die Komplexität der bei einer Dekomposition erhaltenen Komponenten soll im Intervall

$$[c\sqrt{n},\ 2c\sqrt{n} + 1]$$

liegen, wobei c eine Konstante ist (z. B. c = 1) und n die Anzahl der Eckpunkte des Gesamtobjekts bezeichnet.

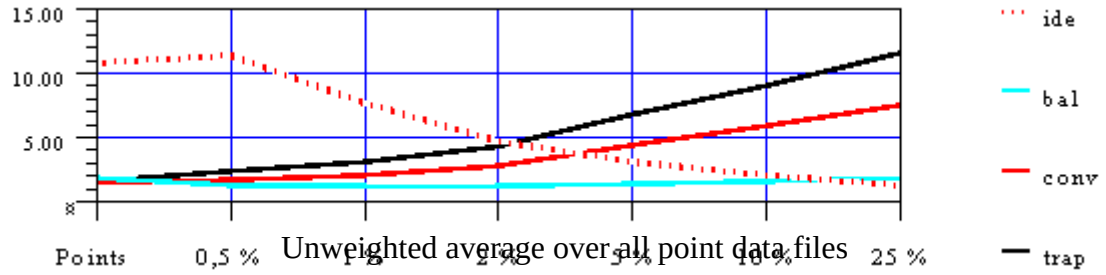
- Beispiel: Volta See



n = 5449

57 Komponenten

Experimentelle Untersuchung der balancierten Dekomposition



Unweighted average over all point data files

Durchschnittliche Antwortzeit für verschiedene Window-Größen **(**in msec pro Antwort)

==> Die balancierte Dekomposition

- degeneriert nicht für wenig selektive Window-Queries
- verbessert die Leistung im Vergleich zur Identität um einen Faktor von bis zu 10.

7.5 Literatur

Literatur (Raumzugriffsstrukturen)

[BKSS 90] Beckmann N., Kriegel H.-P., Schneider R., Seeger B.: *'The R*-tree: An Efficient and Robust Access Method for Points and Rectangles'*, Proc. ACM SIGMOD Int. Conf. on Management of Data, Atlantic City, N.J., 1990, pp. 322-331.

[Gün 89] Günther O.: *'The Design of the Cell Tree: An Object-Oriented Index Structure for Geometric Databases'*, Proc. IEEE 5th Int. Conf. on Data Engineering, Los Angeles, CA., 1989, pp. 598-605.

[Gut 84] Guttman A.: *'R-trees: A Dynamic Index Structure for Spatial Searching'*, Proc. ACM SIGMOD Int. Conf. on Management of Data, Boston, MA., 1984, pp. 47-57.

[Oos 90] Oosterom P. J. M.: *'Reactive Data Structures for Geographic Information Systems'*, PhD-thesis, Dept. of Computer Science at Leiden University, Netherland, 1990.

[PSTW 93] Pragel B. U., Six H.-W., Toben H., Widmayer P.: *'Towards an Analysis of Range Query Performance in Spatial Data Structures'*, Proc. ACM SIGMOD Principles of Database Systems, Washington, 1993, pp. 214-221.

[SK 88] Seeger B., Kriegel H.-P.: *'Techniques for Design and Implementation of Efficient Spatial Access Methods'*, Proc. 14th Int. Conf. on Very Large Databases, Los Angeles, CA., 1988, pp. 360-371.

Literatur (Anfragebearbeitung)

[BHKS 93] Brinkhoff T., Horn H., Kriegel H.-P., Schneider R.: *'A Storage and Access Architecture for Efficient Query Processing in Spatial Database Systems'*, Proc. 3rd Int. Symp. on Large Spatial Databases, Singapore, 1993, in: Lecture Notes in Computer Science, Vol. 692, Springer, 1993, pp. 357-376.

[BKS 93] Brinkhoff T., Kriegel H.-P., Schneider R.: *'Comparison of Approximations of Complex Objects used for Approximation-based Query Processing in Spatial Database Systems'*, Proc. 9th Int. Conf. on Data Engineering, Vienna, Austria, 1993, pp. 40-49.

[BKS 93] Brinkhoff T., Kriegel H.-P., Seeger B.: *'Efficient Processing of Spatial Joins Using R-trees'*, Proc. ACM SIGMOD Int. Conf. on Management of Data, Washington DC, 1993, pp. 237-246.

[BKSS 94] Brinkhoff T., Kriegel H.-P., Schneider R., Seeger B.: *'Multi-Step Processing of Spatial Joins'*, Proc. ACM SIGMOD Int. Conf. on Managment of Data, Minneapolis, MN, 1994, pp. 197-208.

[Kri 91] Kriegel H.-P., Heep P., Heep S., Schiwietz M., Schneider R.: *'An Access Method Based Query Processor for Spatial Database Systems'*, Proc. Int. Workshop on Database Management Systems for Geographical Applications, Capri, Italy, 1991, in: Geographic Database Management Systems, Springer, 1992, pp. 273-292.

[KHS 91] Kriegel H.-P., Horn H., Schiwietz M.: *'The Performance of Object Decomposition Techniques for Spatial Query Processing'*, Proc. 2nd Symp. on Large Spatial Databases, Zurich, Switzerland, 1991, in: Lecture Notes in Computer Science, Vol. 525, Springer, 1991, pp. 257-276.

[SK 93] Schiwietz M., Kriegel H.-P.: *'Query Processing of Spatial Objects: Complexity versus Redundancy'*, Proc. 3rd Symp. on Large Spatial Databases, Singapore, 1993.

Kapitel 8
Methoden für Ähnlichkeitsanfragen

Skript zur Vorlesung: Anfragebearbeitung und Indexstrukturen in Datenbanksystemen
Wintersemester 2012/13, LMU München

Übersicht

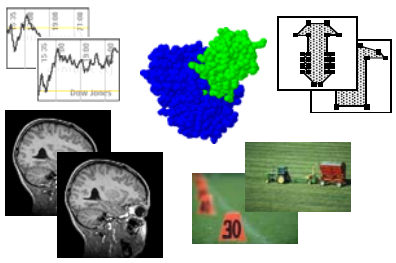
8.1 Feature-Basierte Ähnlichkeit

8.1.1 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
- Prinzipien:
  - Entwickle nicht für jedes der einzelnen Anwendungsgebiete/ Ähnlichkeitsmaße spezielle Indexstrukturen
  - Versuche mit wenigen Arten von Indexstrukturen möglichst viele Anwendungsgebiete abzudecken
- Indexstrukturen für:
  - Multidimensionale Vektoren
  - Allgemeine metrische Daten (beliebige Objekte, auf denen eine metrische Distanzfunktion definiert ist)

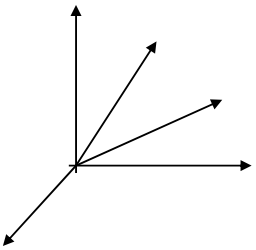
- Extrahiere charakteristische (numerische) Eigenschaften („Features“) aus den Objekten



Anwendungsgebiete (Objektraum)

Feature-Transformation

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



Feature-Vektor-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

8.1.2 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 $\forall x,y,z \in Dom: dist(x,z) \leq dist(x,y) + dist(x,z)$

– 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.

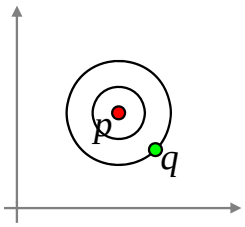
$\forall x,y \in OBJ: dist_{OBJ}(x,y) \equiv dist_{Dom}(T(x),T(y))$

– Distanzmaße in Vektorräumen

- Euklidische Norm (L_2):

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

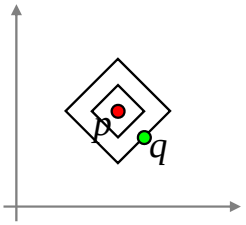
  Natürliches Distanzmaß



- Manhattan-Norm (L_1):

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

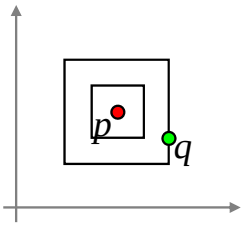
  Die Unähnlichkeiten der einzelnen Merkmale werden direkt addiert.



- Maximums-Norm (L_∞):

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

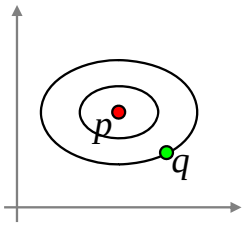
  Die Unähnlichkeit des am wenigsten ähnlichen Merkmals zählt.



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

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

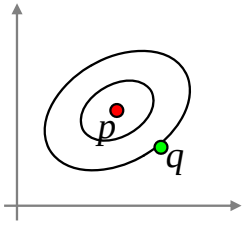
  Häufig sind die Wertebereiche der Merkmale deutlich unterschiedlich.
  Beispiel: Merkmal $M_1 \in [0.01 \ldots 0.05]$
  Merkmal $M_2 \in [3.07 \ldots 22.2]$
  Damit M1 überhaupt berücksichtigt wird, muss es höher gewichtet werden.



- Quadratische Form:

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

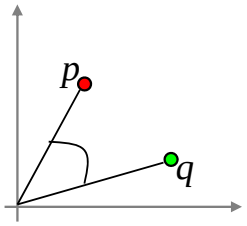
  Bisherige Abstandsmaße gewichten Merkmale nur getrennt.
  Besonders bei Farbhistogrammen müssen verschiedene Merkmale gemeinsam gewichtet werden.



- Cosinus-Distanz

  dist = cos(winkel(p,q))

  Berechnet den Cosinus des Winkels.
  Meist für sehr hochdimensionale
  Feature-Vektoren (z.B. Texte).



– Bemerkungen

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

LMU Database Systems

8.2 Algorithmische Paradigmen zur Anfragebearbeitung

8.2.1 Übersicht

- Typen von Ähnlichkeitsanfragen
  - Bereichsanfragen
  - Nächste-Nachbarn-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 die Auswertung des Anfrage-Prädikats)
  - Indexbasierte Suche
  - Mehrstufige Anfragebearbeitung

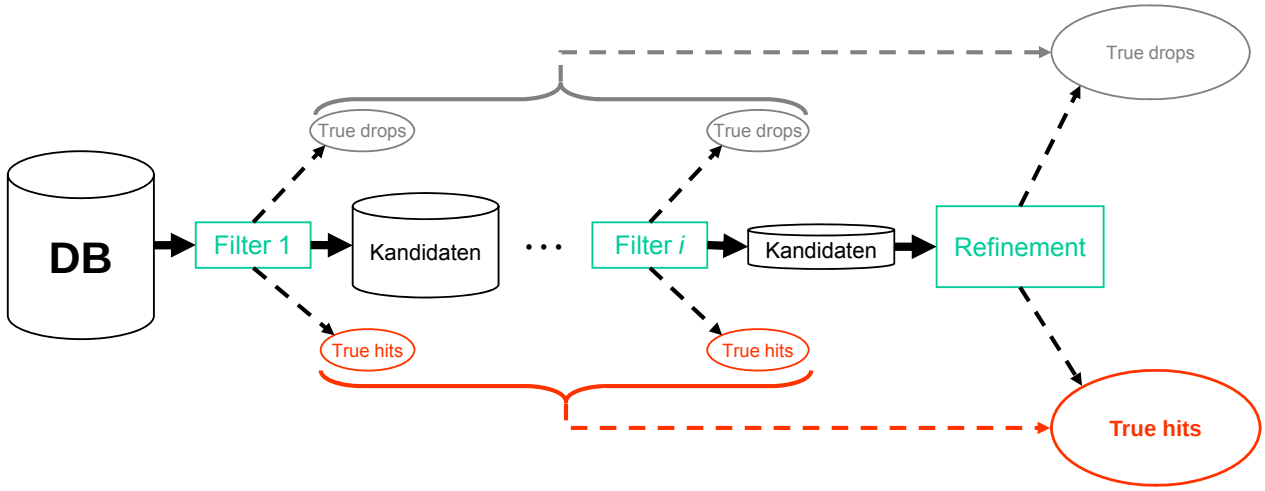
8.2.3 Mehrstufige Anfragebearbeitung

- Idee:
  - Verwendetes Distanzmaß ist sehr teuer (z.B. Edit-Distanz) oder nicht feature-basiert (z.B. Überlappungsfläche von Polygonen)
  - Vektorraum ist sehr hochdimensional (Curse of Dimensionality)
  - Benutze ein feature-basiertes (meist niedrig-dimensionales) 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 exakte Distanzmaß berechnet, was i.A. teurer, dafür selektiver als der Filterschritt ist

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



– Zusammenpassen von Filter und Refinement
  - Idealfall: Filterdistanz ist obere oder untere 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

8.3 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

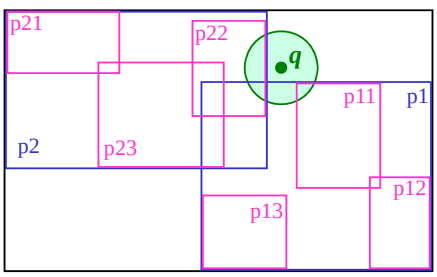
$$RQ(q,\varepsilon) = \{o \in DB \mid dist(q,o) \leq \varepsilon\}$$

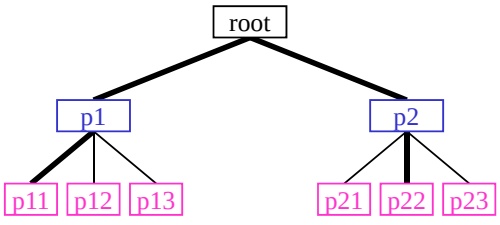
– Basisalgorithmus (sequential scan)

```
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

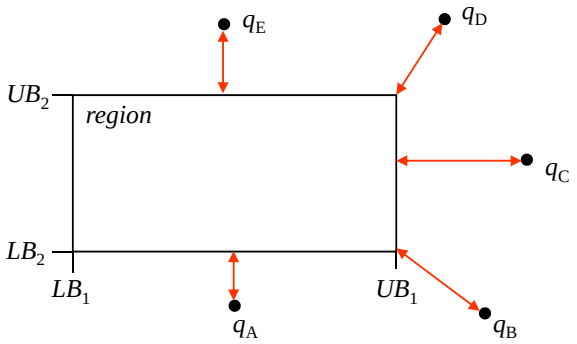
```
RQ-Index(pa, q, ε)          // pa = page address z.B. der Wurzel des Indexes
    result = ∅;
    p := pa.loadPage();
    IF p.isDataPage() THEN
        FOR i=0 TO p.size() DO
            IF dist(q, p.getObject(i)) ≤ ε THEN
                result := result ∪ getObject(i);

    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

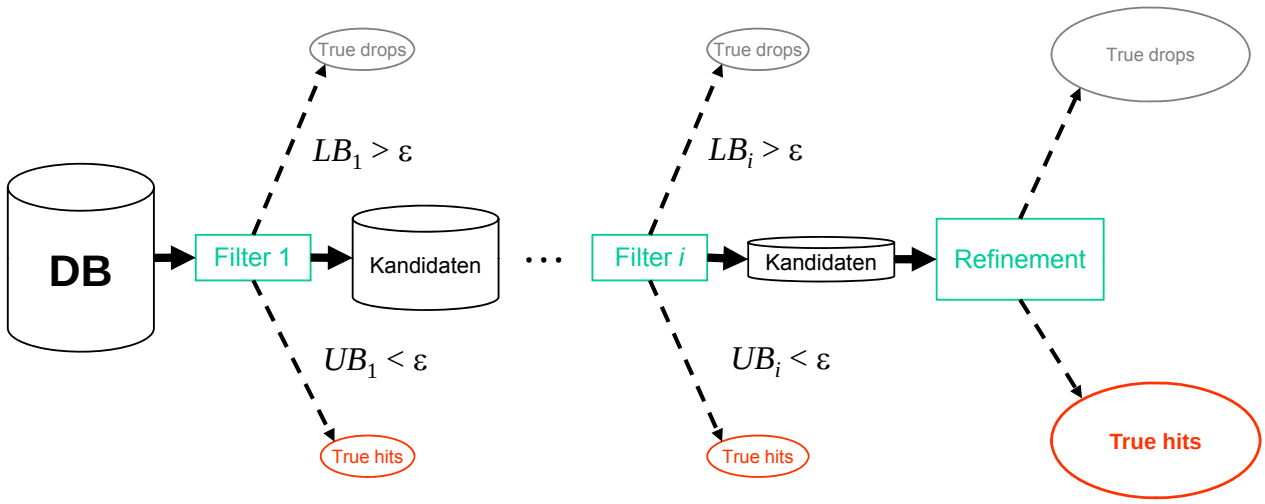
$$\text{MINDIST}(q, region) = \sqrt{\sum_{0<i\le d} \begin{cases} (region.LB_i - q_i)^2 & \text{if} \quad q_i \le region.LB_i \\ 0 & \text{if} \quad region.LB_i \le q_i \le region.UB_i \\ (q_i - region.UB_i)^2 & \text{if} \quad region.UB_i \le q_i \end{cases}}$$



– 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;
```



$LB_1 \leq LB_2 \leq \ldots \leq LB_i$

$UB_1 \geq UB_2 \geq \ldots \geq UB_i$

– Oft nur Lower Bounding Distanzen verwendet

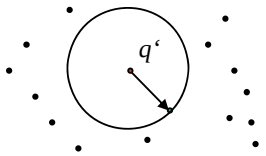
=> Anzahl der Kandidaten größer, da keine true hits

8.4 Nächste-Nachbarn-Anfragen

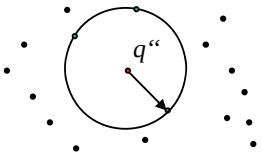
8.4.1 Nächste-Nachbarn-Anfrage

– 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 = ∅;
    pruningdist = +∞;
    FOR i=1 TO n DO
        IF dist(q, DB.getObject(i)) ≤ pruningdist THEN
            result := getObject(i);
            pruningdist = 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 gefundenen Nachbarn

```
Globale Variable: pruningdist = +∞;

NN-Index-Simple-TS(pa, q)          // pa = page address 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)) ≤ pruningdist THEN
                result := getObject(i);
                pruningdist = dist(q, p.getObject(i));
    ELSE                               // p ist Directoryseite
        FOR i=0 TO p.size() DO
            IF MINDIST(q, p.getRegion(i)) ≤ pruningdist THEN
                result := NN-Index-Simple-TS(p.childPage(i), q)
    RETURN result;
```


– Algorithmus mit Index: Prioritätssuche nach [HS 95]

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

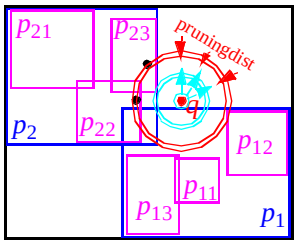
• Statt rekursivem Durchlauf: Liste der aktiven Seiten (active page list APL)
- Seite p ist aktiv genau dann, wenn folgende Bedingungen erfüllt sind:
  - » p wurde noch nicht geladen
  - » Elternseite von p wurde bereits geladen
  - » MINDIST(*q*, p.getRegion()) ≤ pruningdist
- APL wird mit Wurzel des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

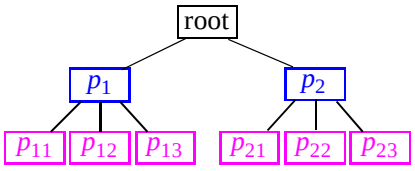
– Algorithmus:

Globale Variablen: pruningdist = +∞;

```
NN-Index-HS(pa, q)              // pa = page address 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.getFirst().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 unterster Directorylevel in APL stehen
  - » Dieser Fall ist allerdings unwahrscheinlicher als bei Breitensuche
  - » Speicherkomplexität O(n) (Tiefensuche O(log n)

- Optimalität des Verfahrens

  [Berchtold, Böhm, Keim, Kriegel. ACM Symp. 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).

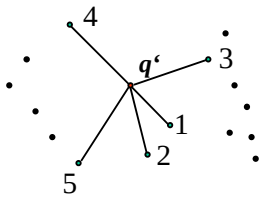
    Beweis: Nach Lemma 2 können nach Zugriff auf Seite s nur Punkte p gefunden werden, mit dist(q, p) > MINDIST(q, s). Wäre vor Zugriff auf s ein Punkt p mit dist(q, p) < MINDIST(q, s) gefunden worden, dann wäre s aus der APL gelöscht worden bzw. der Algorithmus hätte vor der Bearbeitung von p angehalten.

    □

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

8.4.3 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



– 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;
```


LMU Database Systems