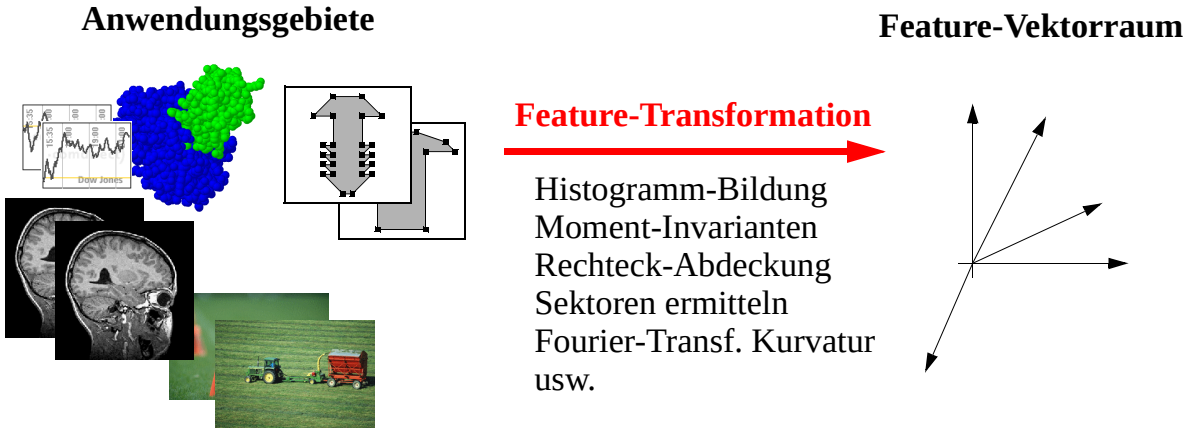
Kapitel 3 Algorithmen zur Ähnlichkeitssuche

- Vorangegangenes Kapitel:
  Wie kann man für verschiedene Applikationen Ähnlichkeit formal so definieren daß der applikationsspezifische, intuitive Ähnlichkeitsbegriff am besten wiedergegeben wird?
  → *Effektivität* der Ähnlichkeitssuche.
- Dieses und die folgenden Kapitel:
  Wie kann man unter Berücksichtigung der vorgegebenen Ähnlichkeitsmaße schnell die zu einem Anfrageobjekt ähnlichen (bzw. ähnlichsten) Datenbankobjekte finden?
  → *Effizienz* der Ähnlichkeitssuche.

Für die Effizienz in Datenbanken ist der Einsatz geeigneter Indexstrukturen entscheidend. Die Entwicklung spezieller Indexstrukturen für einzelne Anwendungsgebiete oder Ähnlichkeitsmaße ist jedoch im allgemeinen zu aufwendig. Man versucht deshalb mit wenigen Arten von Indexstrukturen möglichst viele Anwendungsgebiete abzudecken.
→ Transformation der Objekte in Vektoren (*Featuretransformation*)

3.1 Featuretransformation

3.1.1 Prinzip



- Aus den Objekten werden numerische Features extrahiert, die Objekte charakterisieren, z.B. Länge, Breite, Kurvatur-Parameter usw. ⇒ Feature-Vektoren
- Wichtigste Eigenschaft der Feature-Transformation:
  Ähnlichkeit der Objekte entspricht geringem Abstand der Feature-Vektoren

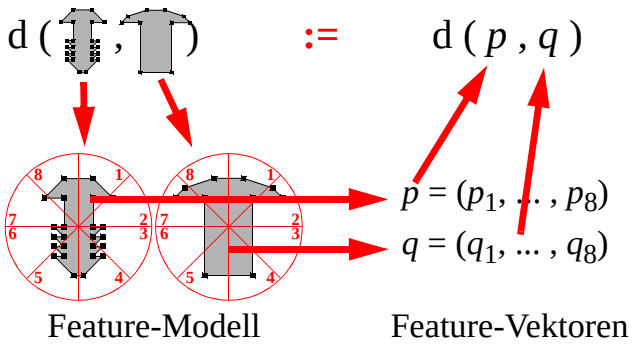
- Hierdurch werden Ähnlichkeitsanfragen im Objektraum generell in Nachbarschaftsanfragen (Range Queries, Nearest Neighbor Queries, etc.) im Featureraum übertragen. Verschiedene Abstandsnormen wie z.B.
  — Euklidische Norm
  — Maximumsnorm
  — Ellipsoid-Distanz usw.
- Diese werden durch geeignete *multidimensionale Indexstrukturen* effizient unterstützt
- Kann keine sinnvolle Featuretransformation angegeben werden, dann kann alternativ dazu auch die Metrik-Eigenschaft des Abstandsmaßes zur Indexierung und Anfragebearbeitung ausgenutzt werden (*metrische Indexstruktur*, siehe später)

3.1.2 Verfahrensklassen

- Verfahren mit featurebasiertem Abstandsmaß
  — entweder lassen sich die Objekte direkt als Vektoren interpretieren (z.B. Sequenzen)
  — oder Featurevektoren werden anwendungsspezifisch ermittelt
- Verfahren, die ein featurebasiertes Abstandsmaß als Filterschritt nutzen
- Verfahren mit metrischem (aber nicht featurebasierten) Abstandsmaß
- Sonstige Verfahren

Verfahren mit featurebasiertem Abstandsmaß:

- Die Ähnlichkeit der Objekte ist *unmittelbar* durch den Abstand der zugeordneten Featurevektoren *definiert*.
- Für die Anfragebearbeitung ist kein Verfeinerungsschritt erforderlich. Die aus dem Index gewonnen Treffer sind sichere *Ergebnisse* der Anfrage.



Verfahren, die ein featurebasiertes Abstandsmaß als Filterschritt nutzen

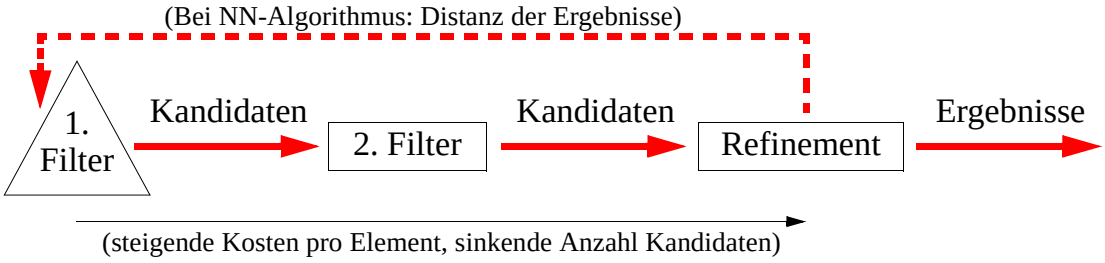
- Das *eigentliche* Ähnlichkeitsmaß kann z.B.
  — eine Metrik,
  — ein weiteres featurebasiertes Abstandsmaß oder sogar
  — ein nicht distanzbasiertes Ähnlichkeitsmaß sein
- Für die Anfragebearbeitung brauchen wir eine *Multi-Step-Architektur*:
  — die aus dem Index ermittelten Vektoren sind keine Ergebnisse sondern Kandidaten

— die Kandidatenmenge sollte jedoch klein gegenüber der Menge aller Objekte der Datenbank sein (*Filter-Selektivität* σ_F)

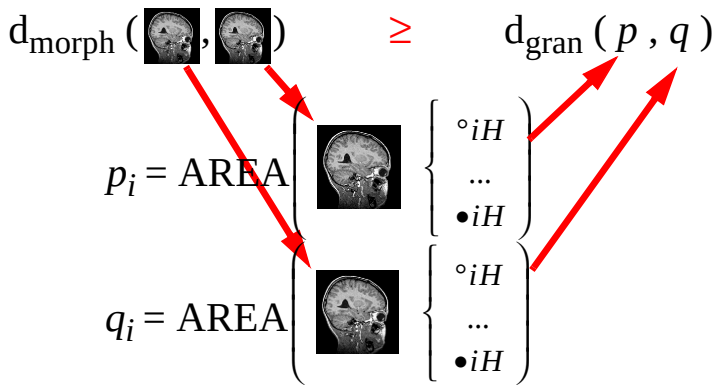
$$\sigma_F = \frac{\text{Anzahl der produzierten Kandidaten}}{\text{Anzahl der Datenbankobjekte}}$$

— für die Kandidaten wird das eigentliche Abstandsmaß berechnet, was i.A. teurer als der Filterschritt ist, aber auch selektiver

- Kaskadierte Filterschritte möglich:
  — erster Filter ermittelt Kandidatenmenge aus dem Index
  — weitere Filter schränken die Kandidatenmenge möglichst weiter ein
  — Verfeinerungsschritt bildet abschließenden Test auf Korrektheit der Ergebnisse
- Multi-Step-Architektur wirkt sich auch auf Algorithmen zur Anfragebearbeitung aus:
  — Range-Queries: relativ unbeeinflusst
  — Nearest-Neighbor-Queries: Zusammenspiel von Filter und Refinement



- Zusammenpassen von Filter und Refinement-Step:
  — Im Idealfall weist man nach, daß die Filterdistanz eine untere Schranke (*lower bound*) der Objektdistanz ist. Dann ist (entsprechende Algorithmen vorausgesetzt) garantiert, daß durch den Filter keine Anfrageergebnisse verloren gehen (*no false dismissals*). Die Ergebnismenge ist *vollständig*.
  — Gilt die *lower bounding property* nicht, dann können Anfrageergebnisse verloren gehen, d.h. die Ergebnismenge ist *nicht vollständig*. In diesem Fall muß *experimentell* nachgewiesen werden, daß der *Recall* der Anfragebearbeitung hoch ist, d.h. daß nicht zu viele Ergebnisse fallen gelassen werden.
- Beispiel



3.2 Multidimensionale Indexstrukturen

[BBK 01] Böhm C., Berchtold S., Keim D.A.: *Searching in High-dimensional Spaces: Index Structures for Improving the Performance of High-Dimensional Indexing*, to appear in ACM Computing Surveys, 2001.

3.2.1 Generelle Prinzipien vieler Indexstrukturen

- Jedem Knoten des Baums ist zugeordnet:
  — eine *Seite* des Hintergrundspeichers
  — eine *Region* des Datenraums
- Es gibt zwei Typen von Seiten (=Knoten):
  — Datenseiten sind Blattknoten und speichern Punktdaten
  — Directoryseiten sind innere Knoten und speichern *Directory-Einträge*, bestehen aus
    - *Verweis* zu Kindseite (Adresse auf dem Hintergrundspeicher) und
    - Beschreibung der *Region* der Kindseite
- Physische vs. logische Seiten:
  — Ursprüngliche Idee: Eine *physische* Seite des Hintergrundspeichers wird verwendet.
  — Kleinste Informationseinheit, die zwischen Plattenspeicher und Arbeitsspeicher übertragen werden kann.
  — physische Seiten meist zu klein
  fasse aufeinanderfolgende physische Seiten zu einer *logischen Seite* zusammen

  — Die meisten Indexstrukturen verwenden aber eine *einheitliche* Seitengröße für alle Seiten eines Indexes, um Probleme wie Freispeicherverwaltung zu vermeiden
  — Aus der Seitengröße ergibt sich auch die *Kapazität* (maximale Anzahl Einträge) der Directory- und Datenseiten:

$$C_{\text{Data}} := \left\lfloor \frac{|\text{Seite}| - |\text{Verwaltungsoverhead}|}{|\text{Datensatz}|} \right\rfloor \qquad C_{\text{Dir}} := \left\lfloor \frac{|\text{Seite}| - |\text{Verwaltungsoverhead}|}{|\text{Directoryeintrag}|} \right\rfloor$$

  — Meist werden die Seiten nicht zu 100% gefüllt, damit Platz für neu eingefügte Datensätze bleibt. Meist wird eine minimale Speicherauslastung, z.B. 40% definiert.
  — Der durchschnittliche Anteil besetzter Einträge wird als *Speicherauslastung* (*storage utilization*), die durchschnittliche Anzahl besetzter Einträge als *effektive Kapazität* bezeichnet mit:

$$C_{\text{eff,Data}} = su_{\text{Data}} \cdot C_{\text{Data}}$$

- Jeder Punkt-Datensatz wird in genau einer Datenseite gespeichert (keine Duplikate)
- Durch die Regionen wird gewährleistet, daß räumlich benachbarte Punkte möglichst auf denselben Datenseiten oder in denselben Teilbäumen gespeichert werden.
- Verschiedene Möglichkeiten für die Gestalt der Seitenregionen:
  — achsenparallele Rechtecke, die meist minimal um die Punktemenge gespannt werden (*minimum bounding rectangle, MBR*) am weitesten verbreitet (R-tree, X-tree ...)

— Kugeln (SS-tree)
— Zylinder (TV-tree)
— Kombinationskörper (SR-tree)

- Die Seitenregion umfaßt immer geometrisch alle auf einer Datenseite bzw. in dem entsprechenden Teilbaum unter einer Directoryseite gespeicherten Punkte.
  Dies ist Voraussetzung um die Vollständigkeit des Anfrageergebnisses zu gewährleisten (*konservative Approximation* bzw. *lower bounding property*)
- Bäume sind meist *balanciert*: Abstand zwischen Wurzel und alle Blätter gleich groß
- Indexstrukturen sind *dynamisch*:
  Insert- und Delete-Operationen sind effizient, möglichst in O (log *n*)
  Typische Vorgehensweise:
  — Auf einen Pfad beschränkte *Tiefensuche* nach einer geeigneten *Datenseite*:
  Nach bestimmten räumlichen Kriterien wird jeweils die beste Kindseite gewählt
  — Wenn auf der Datenseite noch Platz ist, wird der Punkt dort gespeichert
  — Manche Indexstrukturen: Restrukturierungsversuche ohne Anlegen neuer Seite
  — Sonst wird im Rahmen einer Überlaufbehandlung die Datenseite aufgeteilt (*Split*)
  - verschiedene Kriterien (*minimale Überlappung, toter Raum*)
  - verschiedene Algorithmen (*linear, quadratisch*)

- neuer Knoten wird im Elternknoten eingetragen
- durch Überlauf von Elternknoten kann Split im Extremfall bis zur Wurzel laufen

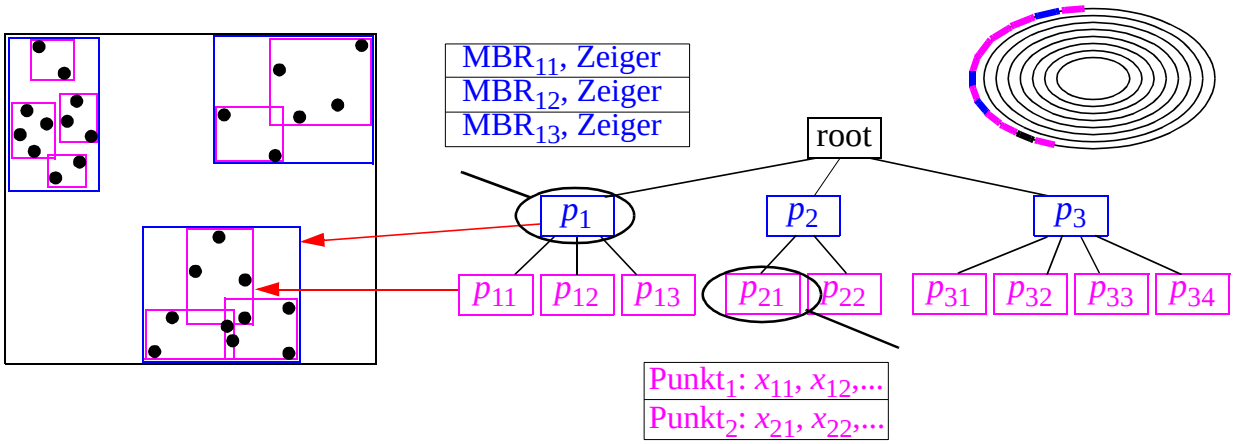
3.2.2 Beispiel: R*-Baum

Der R*-Baum ist ein balancierter Baum, der zur Speicherung von 2D Rechteckdaten entworfen wurde. Er wird jedoch häufig auch zur Speicherung höherdimensionaler Punktdaten verwendet und diente als Grundlage für wichtige spezialisierte Indexstrukturen für hochdimensionale Daten wie z.B. den TV-tree oder den X-tree.

- Balanzierte Indexstruktur mit Daten- und Directoryseiten einheitlicher Größe
- Seitenregionen: · Minimal umgebende achsenparallele Rechtecke (MBR)
- Insert-Strategie: · Vergrößerung des Overlap
  · Vergrößerung des Volumen
  · Volumen
- Überlaufbehandlung: Re-Insert-Konzept:
  · Teil der Punkte wird gelöscht und in den Baum neu eingefügt
- Split-Kriterien: · Umfang/Oberfläche
  · Überlappungsvolumen

· (Toter Raum, nicht sinnvoll bei Datenseiten mit Punktdaten)

- Split-Algorithmus: erste Phase: Ermittlung der ·Dimension
  zweite Phase: Ermittlung der ·Splitebene
  jeweils durch Sortieren.



3.3 Exakte Anfragen

3.3.1 Allgemeines

- Charakteristik:
  Benutzer gibt Anfrageobjekt q vor;
  System ermittelt, ob sich Datenbankobjekte auf derselben Position befinden.
- Formale Definition:

$$\text{ex}_q(\varepsilon) := \{o \in DB \mid o = q\}$$

- Nichtdeterministische Version:

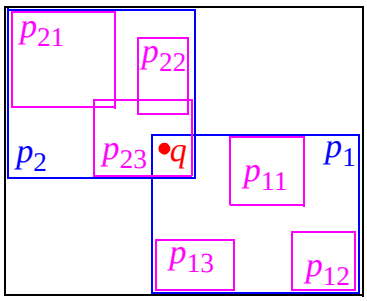
$$\text{ex}_q(\varepsilon) := \text{SOME } o \in DB \mid o = q$$

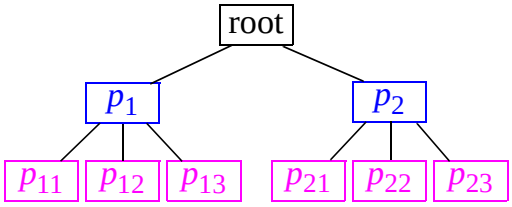
3.3.2 Basisalgorithmus ohne Index

```
result := ∅ ;
for i:=1 to n do
    if q = database [i] then
        result := result ∪ database [i] ;
```

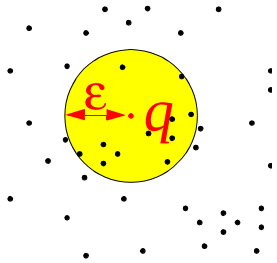
3.3.3 Basisalgorithmus mit Multidimensionalem Index (Tiefensuche)

```
function ExactMatchQuery (q: Point, pa: DiskAddress): Set of Point
    result := ∅ ;
    p := LoadPage (pa) ;
    if IsDataPage (p) then
        for i := 0 to p.num_objects do
            if q = p.object [i] then
                result := result ∪ p.object [i] ;
    else
        for i := 0 to p.num_objects do
            if IsPointInRegion (q, p.region [i]) then
                result := result ∪ ExactMatchQuery (q, p.childpage [i]) ;
    return result ;
```





3.4 Bereichsanfragen



3.4.1 Allgemeines

- Charakteristik:
  Benutzer gibt Anfrageobjekt q und maximale Distanz ε vor;
  System ermittelt alle Datenbankobjekte, die von q höchstens die Distanz ε haben
- Formale Definition:

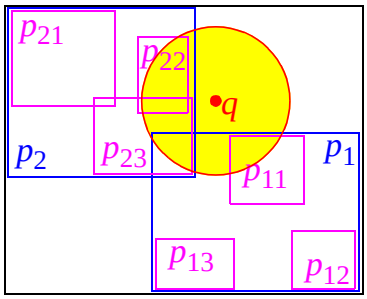
$$\text{sim}_q(\varepsilon) := \{o \in DB \mid d(q,o) \leq \varepsilon\}$$

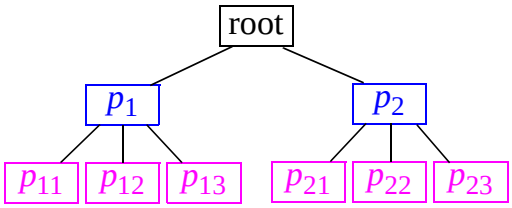
3.4.2 Basisalgorithmus ohne Index:

```
result := ∅ ;
for i:=1 to n do
    if distance (q, database [i]) ≤ ε then
        result := result ∪ database [i] ;
```

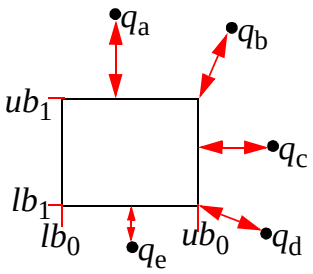
3.4.3 Algorithmus mit multidimensionalem Index: Tiefensuche (rekursiv)

```
function RangeQuery (q: Point; ε: Real; pa: DiskAddress): Set of Point
    result := ∅ ;
    p := LoadPage (pa) ;
    if IsDataPage (p) then
        for i := 0 to p.num_objects do
            if distance (q, p.object [i]) ≤ ε then
                result := result ∪ p.object [i] ;
    else
        for i := 0 to p.num_objects do
            if MINDIST (q, p.region [i]) ≤ ε then
                result := result ∪ RangeQuery (q, p.childpage [i]) ;
    return result ;
```





- Der Test, ob sich die Queryregion mit einer Seitenregion schneidet, wird am einfachsten mit Hilfe der Distanz zwischen dem Anfragepunkt und der Seitenregion entschieden. In diesem Fall wird die *minimale* Distanz zwischen dem Anfragpunkt und einem Punkt der Seitenregion benötigt (MINDIST).



- Berechnung der MINDIST für den euklidischen Abstand:

$$\text{MINDIST}(p,q) := \sqrt{\sum_{0<i\le d} \begin{cases} (lb_i - q_i)^2 & \text{if } q_i \le lb_i \\ 0 & \text{if } lb_i \le q_i \le ub_i \\ (q_i - ub_i)^2 & \text{if } ub_i \le q_i \end{cases}}$$

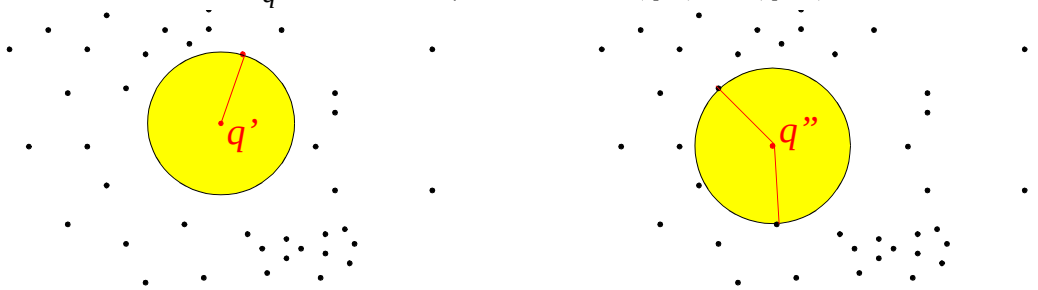
3.5 Nächste-Nachbar-Anfragen

3.5.1 Allgemeines

- Benutzer gibt Anfrageobjekt q vor;
  System ermittelt das Datenbankobjekt, das von q den geringsten Abstand hat

- *Mehrdeutigkeiten* können auftreten und müssen sinnvoll behandelt werden: entweder *mehrere* nächste Nachbarn, oder das Ergebnis ist nichtdeterministisch.
- Formale Definition:

$$\mathrm{NN}_q := \{o \in DB \mid \forall o' \in DB\ \mathrm{d}(q,o) \leq \mathrm{d}(q,o')\}$$



a) q' hat eindeutigen nächsten Nachb. b) q'' hat zwei nächste Nachbarn

- Nichtdeterministische Variante:

$$\mathrm{NN}_q := \text{SOME } o \in DB \mid \forall o' \in DB\ \mathrm{d}(q,o) \leq \mathrm{d}(q,o')$$

3.5.2 Basisalgorithmus ohne Index (nichtdeterministisch)

result := ⊥ ;
resultdist := +∞ ;

for *i*:=1 **to** *n* **do**
  if distance (*q*, *database* [*i*]) ≤ *resultdist* **then**
    result := *database* [*i*] ;
    resultdist := distance (*q*, *database* [*i*]) ;

3.5.3 Einfacher Tiefensuche-Algorithmus

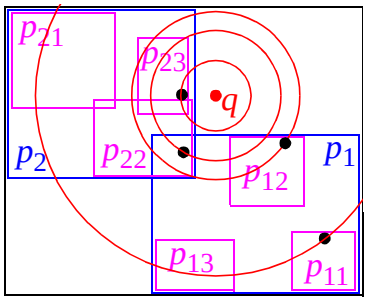
- Unterschied zu bisherigen Anfragealgorithmen:
  Da der nächste Nachbar prinzipiell beliebig weit vom Anfragepunkt entfernt sein kann, ist die Gestalt der Query zunächst unbekannt, d.h. es ist nicht wie bei der Range-Query leicht anhand der Seitenregion zu entscheiden ob eine Seite gebraucht wird. Ob eine Seite gebraucht wird, hängt *auch* davon ab was auf den anderen Seiten gespeichert ist.
- Wäre der Abstand zum nächsten Nachbarn (bzw. eine obere Schranke hierfür) bekannt, würde Range-Query ausreichen.
  — Kennt man *einen beliebigen* Punkt der Datenbank, dann kann man dessen Abstand als obere Schranke für die Nearest-Neighbor-Distance nutzen.
  — Kennt man *mehrere* Punkte der Datenbank, dann kann man den *geringsten* Abstand als obere Schranke für die Nearest-Neighbor-Distance nutzen.
- Der Algorithmus für Range-Queries wird deshalb so umformuliert, daß ε durch die Distanz zum besten, bisher gefundenen Nachbarn (*resultdist*) ersetzt wird.

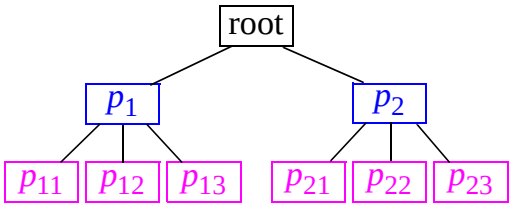
Die globalen Variablen *result* und *resultdist* werden wie oben mit ⊥ bzw. +∞ initialisiert

```
procedure SimpleNNQuery (q: Point; pa: DiskAddress)
    p := LoadPage (pa) ;
    if IsDataPage (p) then
        for i := 0 to p.num_objects do
            if distance (q, p.object [i]) ≤ resultdist then
                result := p.object [i] ;
                resultdist := distance (q, p.object [i]) ;
    else
        for i := 0 to p.num_objects do
            if MINDIST (q, p.region [i]) ≤ resultdist then
                SimpleNNQuery (q, p.childpage [i]) ;
```

- Nachteil des einfachen Tiefensuche-Algorithmus:
  - Fängt mit *resultdist* = +∞ an
  - Hierdurch: Start mit einem beliebigen Pfad, nicht mit einem Pfad, der möglichst nahe am Anfragepunkt liegt
  - Dadurch sind auch die ersten gefundenen Punkte sehr weit vom Anfragepunkt weg

  - D.h. das Verfahren schränkt seinen Suchraum nur sehr langsam ein. Viele Pfade werden unnötigerweise verfolgt.





3.5.4 Tiefensuche-Algorithmus von Roussopoulos, Kelley und Vincent

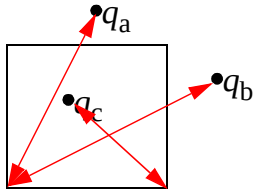
[RKV 95] Roussopoulos N., Kelly S., Vincent F.: *Nearest Neighbor Queries*, ACM SIGMOD Int. Conf on Management of Data, 1995

Dieser Algorithmus vermeidet die langsame Einschränkung des Suchraums durch

- Verwendung der Seitenregionen zur Abschätzung der NN-Distanz
- Priorisierung des Tiefensuche nach Abstand der Seitenregionen von q.

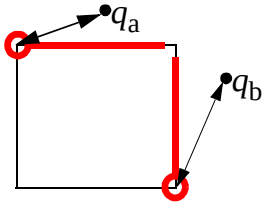
Zur Abschätzung der NN-Distanz werden zusätzlich zur MINDIST noch 2 weitere Distanzmaße für Seitenregionen definiert:

- MAXDIST:
  Anschaulich: Maximale Distanz zwischen p und einem beliebigen Punkt in der Seitenregion;
  Da in jeder Seite Punkte gespeichert werden, kann die NN-Distanz nicht schlechter als MAXDIST werden:



$$\text{MAXDIST} := \sqrt{\sum_{0 < i \le d} \max\{(q_i - ub_i)^2, (q_i - lb_i)^2\}}$$

- MINMAXDIST:
  Verwendet eine Indexstruktur Minimum Bounding Rectangles (MBR) als Seitenregionen, dann kann man die Eigenschaft ausnutzen, daß auf jeder Kante (für d>2 auf jedem (d–1)-dimensionalen Oberflächensegment) der Box ein Punkt liegt (sonst wäre das Rechteck nicht minimal).



- "Nächstliegende Kante, weitester Punkt"

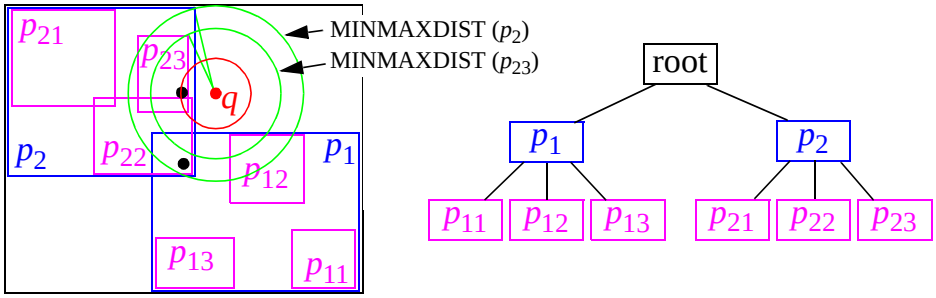
$$\text{MINMAXDIST}^2 := \min_{0 \le k < d} \left(|q_k - rm_k|^2 + \sum_{\substack{i \ne k \\ 0 \le i < d}} |q_i - rM_i|^2 \right)$$

$$\text{wobei } rm_k = \begin{cases} lb_k & \text{if } q_k \le \dfrac{lb_k + ub_k}{2} \\ ub_k & \text{otherwise} \end{cases} \quad \text{und } rM_i = \begin{cases} lb_i & \text{if } q_i \ge \dfrac{lb_i + ub_i}{2} \\ ub_i & \text{otherwise} \end{cases}$$

- Für andere Geometrien (z.B. Kreise) oder allgemeine (nicht *minimum bounding*) Rechtecke:
  — MINDIST und MAXDIST analog definierbar;
  — MINMAXDIST nicht definiert.
- Als Ausschlußkriterium wird nicht nur *resultdist* verwendet sondern das Minimum aus
  — *resultdist*
  — die MINMAXDIST aller bisher bekannter Seitenregionen
  (bzw. MAXDIST bei Seiten-Geometrien ohne MINMAXDIST)
- Vor dem rekursiven Abstieg: Sortieren der Kindseiten nach Priorität:
  MINDIST ist experimentell als geeignetstes Prioritätsmaß ermittelt worden.
- Die globale Variable *pruningdist* wird wie *resultdist* mit $+\infty$ initialisiert.

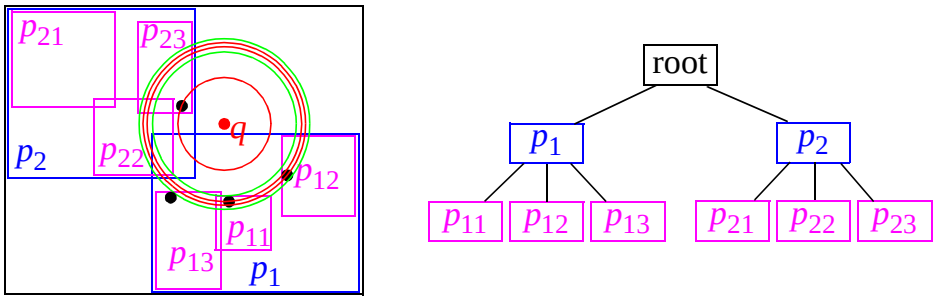
```
procedure SimpleNNQuery (q: Point; pa: DiskAddress)
    p := LoadPage (pa) ;
    if IsDataPage (p) then
        for i := 0 to p.num_objects do
            if distance (q, p.object [i]) ≤ resultdist then
                result := p.object [i] ;
                resultdist := distance (q, p.object [i]) ;
                if resultdist < pruningdist then
                    pruningdist := resultdist ;
    else
        for i := 0 to p.num_objects do
            if MINMAXDIST (q, p.region [i]) < pruningdist then
                pruningdist := MINMAXDIST (q, p.region [i]) ;
        quicksort (p.object, MINDIST);
        for i := 0 to p.num_objects do
            if MINDIST (q, p.region [i]) ≤ pruningdist then
                SimpleNNQuery (q, p.childpage [i]) ;
```


- Beispiel:



- In diesem Beispiel bewirkt die Priorisierung mit MINDIST eine Reduktion von 7 Seitenzugriffen auf 4 Seitenzugriffe.
- Das verbesserte Ausschlußkriterium mit MINMAXDIST verbessert zwar zeitweise die Pruning-Distanz, verhindert hier aber keine Seitenzugriffe.
- Trotz der Priorisierung kann es passieren, daß der Tiefendurchlauf stark fehlgeleitet wird, wenn z.B. eine Seite auf dem ersten Level sehr nah am Querypunkt liegt, ihre Kindseiten aber relativ weit weg.

- Dies ist ein prinzipielles Problem der Tiefensuche



- Bei Start mit p_2 hätte keine der Kind-Seiten von p_1 zugegriffen werden müssen

3.5.5 Breitensuche

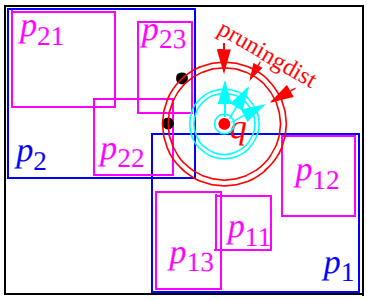
- Abgesehen von MINMAXDIST-Abschätzungen stehen Punkt-Distanzen erst zur Verfügung, wenn man an der Blatt-Ebene angekommen ist. Die erste Punktdistanz hätte aber bereits viele Zugriffe von Directory-Seiten verhindern können
- Speicherplatz-intensiv, weil man im schlechtesten Fall die gesamte letzte Directory-Ebene im Arbeitsspeicher halten muß (sofern nicht vorher aufgrund von MINMAXDIST-Abschätzungen Seiten verworfen werden können)

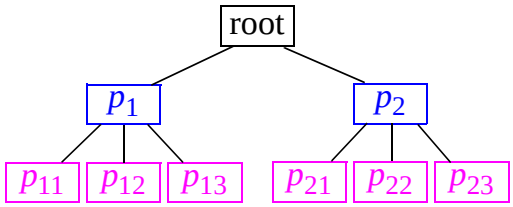
3.5.6 Prioritätssuche nach Hjaltason und Samet

[HS 95] Hjaltason G. R., Samet H.: *Ranking in Spatial Databases*, Int. Symp. on Large Spatial Databases (SSD), 1995.

- Statt eines rekursiven Durchlaufs durch den Index hält der Algorithmus explizit eine *Liste der aktiven Seiten* (*active page list APL*).
- Definition: Eine Seite p ist *aktiv* genau dann wenn folgende Bedingungen erfüllt sind:
  — p wurde noch nicht geladen
  — die Elternseite von p wurde bereits geladen
  — die Distanz (MINDIST) zwischen der Region *p.region* und dem Anfragepunkt übersteigt nicht die Pruningdistanz (Distanz des bisherigen nächsten Nachbarn)
- Anfangs wird *APL* mit der Wurzel des Index initialisiert.
- Der Algorithmus entnimmt in jedem Schritt die Seite aus der APL, die die höchste Priorität (d.h. die geringste MINDIST vom Anfragepunkt q) hat.
- Die entnommene Seite wird geladen und verarbeitet:
  — Datenseiten werden wie bisher verarbeitet
  — Directoryseiten: Kindseiten mit MINDIST $\leq$ *pruningdist* werden in *APL* eingefügt
  — Ändert sich *pruningdist*, dann werden die betroffenen Seiten aus *APL* gelöscht (Alternativ können diese Seiten auch nur als “gelöscht” markiert bzw. auch ohne explizite Markierung später ignoriert werden)

- 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}

- Beobachtung:
  - Seiten werden nach aufsteigendem Abstand (blaue Kreise) geordnet zugegriffen
  - *pruningdist* (rote Kreise) wird kleiner, sobald nähergelegener Datenpunkt gefunden
  - die Anfragebearbeitung stoppt, wenn sich beide Kreise treffen

- Algorithmus:

```
apl: list of (dist: Real, pa: DiskAdr) ordered by dist ascending := ⟨(0.0, root)⟩ ;
while not empty (apl) and apl [0].dist ≤ resultdist do
    p := LoadPage (apl [0].pa) ;
    delete_first (apl) ;
    if (IsDataPage (p) ) then
        (* wie üblich *)
    else
        for i := 0 to p.num_objects do
            h := MINDIST (q, p.region [i]) ;
            if h ≤ resultdist then
                insert ((h, p.childpage [i]), apl) ;
```

- Speicherbedarf:
  - Wie bei der Breitensuche kann es passieren, daß der gesamte unterste Directorylevel in der *APL* steht (siehe Beispiel, 4. Zeile; p_{21} könnte allerdings durch MINMAX-DIST (p_{11}) ausgeschlossen werden)
  - Im Gegensatz zur Breitensuche ist dieser Fall aber unwahrscheinlicher.
  - Die Speicherplatz*komplexität* (worst case) ist jedenfalls mit O(n) schlechter als die der Tiefensuche mit O(log n)

Optimalität des Verfahrens

[BBKK 97] Berchtold, Böhm, Keim, Kriegel: *A Cost Model for Nearest Neighbor Search in High-Dimensional Space*, ACM Symposium on Principles of Database Systems, 1997.

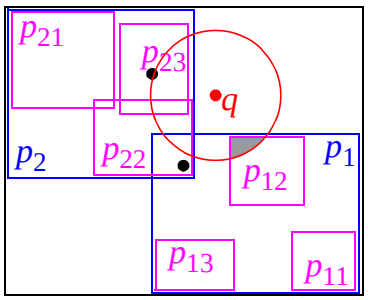
Im folgenden wollen wir zeigen, daß die Prioritätssuche optimal in Bezug auf die Anzahl der Seitenzugriffe bei einem gegebenen Index ist. Dieser Beweis besteht aus drei Teilen:

- Lemma 1 sagt aus, daß jeder korrekte Algorithmus zur Bestimmung des nächsten Nachbarn *mindestens* die Seiten einladen muß, die von der Kugel um q geschnitten werden, die den nächsten Nachbarn von q *berührt* (die *NN-Kugel*)
- Lemma 2 stellt sicher, daß das Verfahren die Seiten, wie bereits vorher angemerkt, in aufsteigendem Abstand vom Anfragepunkt zugreift
- Lemma 3 folgert, daß der Algorithmus keine Seite mit einer MINDIST zugreift, die die Distanz des nächsten Nachbarn (von q) überschreitet

Lemma 1. Ein korrekter nearest neighbor Algorithmus muß mindestens die Seiten laden, deren Seitenregion MINDIST vom Anfragepunkt aufweist, die den Abstand zwischen dem nächsten Nachbarn und dem Anfragepunkt nicht überschreitet.

Beweis. Angenommen, eine Seite wird nicht geladen, obwohl sie eine MINDIST aufweist, die kleiner als die Distanz des nächsten Nachbarn ist. Dann kann diese Seite Punkte enthalten (als Datenseite bzw. bei Directoryseiten können im entsprechenden Teilbaum Punkte gespeichert sein), die näher am Anfragepunkt liegen als der nächste Nachbar. Der

nächste Nachbar ist also nicht als solcher validiert, da über die Punkte in einem Teilbaum keine Information bekannt ist, außer daß sie in der entsprechende Region liegen.



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

Beweis. Die Seiten werden in aufsteigender Reihenfolge aus der *APL* entnommen. Die Aussage des Lemmas ist also bewiesen, wenn sichergestellt ist, daß nach Entnahme einer Seite p keine Seiten mehr in APL eingefügt werden, deren MINDIST kleiner als $r :=$ MINDIST (p,q) ist. Alle Seiten, die nach Entnahme von p eingefügt werden sind entweder Nachkommen von p oder Nachkommen von Seiten, deren MINDIST $\geq r$ ist. Da die Region eines Nachkommen in der Region des Vorfahren vollständig eingeschlossen ist, ist die MINDIST eines Nachkommens nie kleiner als die des Vorfahren. Deshalb haben alle später eingefügten Seiten eine MINDIST $\geq r$.

Lemma 3. Der Algorithmus greift auf keine Seite zu, deren Seitenregion eine MINDIST aufweist, die die Distanz zwischen q und seinem nächsten Nachbarn überschreitet

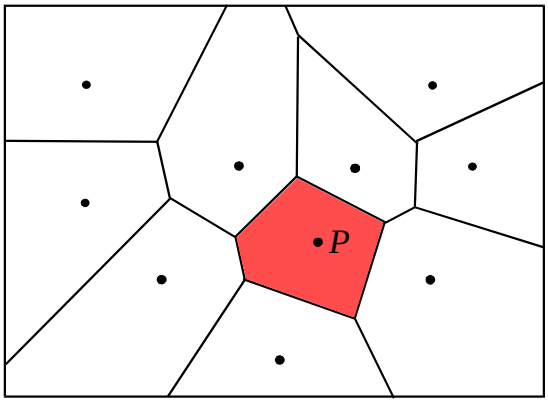
Beweis. Da die Seiten in aufsteigender Reihenfolge zugegriffen werden, können nach Zugriff auf eine Seite p nur Punkte gefunden werden, deren Abstand vom Anfragepunkt größer als MINDIST (p,q) ist (gemäß der Argumentation in Lemma 2). Wäre vor Zugriff auf p ein Punkt gefunden worden, der einen geringeren Abstand aufweist, dann wäre p aus der *APL* gelöscht worden bzw. der Algorithmus hätte vor Bearbeitung von p angehalten.

Korollar. Aus Lemma 1-3 ergibt sich, daß der Prioritätsalgorithmus optimal in bezug auf die Anzahl der Seitenzugriffe ist.

3.5.7 Voronoi-Diagramme

[BEK+ 98] Berchtold S., Ertl B., Keim D. A., Kriegel H.-P., Seidl T.: *Fast Nearest Neighbor Search in High-dimensional Space*, Int. Conf. on Data Engineering (ICDE), 1998, pp. 209-218.

Die Idee dieses Algorithmus ist es, für jeden Punkt $P \in$ DB den Teil des Datenraumes vorzuberechnen, in dem P der nächste Nachbar ist. Dies entspricht dem Konzept der *Voronoi-Diagramme*. Alle (potentiellen Anfrage-) Punkte, die in dem eingefärbten Bereich liegen, haben P



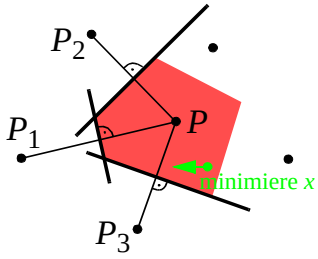
als nächsten Nachbarn. Statt die Punkte in der Datenbank zu speichern und NN-Algorithmen ablaufen zu lassen, kann man auch die Voronoi-Zellen in der Datenbank speichern und Punktanfragen stellen.

Problem:

Die Voronoi-Zellen sind konvexe Polygone, die besonders in Dimensionen $d > 2$ sehr komplex werden (große Anzahl Eckpunkte). Es ist daher schwierig, die Voronoi-Zellen direkt in der Datenbank zu speichern. Statt der Voronoi-Zellen kann man aber auch die minimal umgebenden Rechtecke (MBR) der Voronoi-Zellen oder andere Approximationen speichern. In diesem Fall ist die Punktanfrage natürlich nur ein Filterschritt. Da sich die MBRs der Voronoizellen überlappen, liefert eine Punktanfrage i.a. mehrere Kandidaten, die man durch Vergleich der tatsächlichen Punktabstände verfeinern kann.

Ermittlung der MBRs:

Die Ermittlung der Grenzen der MBRs läßt sich als ein lineares Optimierungsproblem darstellen. Für die Ermittlung der unteren Grenze ug_i in Dimension d_i ergibt sich als Zielfunktion des Optimierungsverfahrens die *Minimierung* von x_i unter den linearen Constraints, die die Voronoizelle vorgibt. Die Constraints sind Halbräume, die durch Ebenen in Normalform (Aufpunkt und Normalenvektor) begrenzt sind.



3.6 *k*-nächste-Nachbar-Anfragen

3.6.1 Allgemeines

- Der Benutzer spezifiziert einen Anfragepunkt q und eine Anzahl k.
  Das System ermittelt die k nächsten Nachbarn von q.
- formale Definition:
  kleinste Menge $NN_q(k) \subseteq DB$ mit mindestens k Objekten, so daß
  $$\forall o' \in NN_q \; \forall o' \in DB \setminus NN_q: d(q,o) < d(q,o')$$
- nichtdeterministische Variante:
  Menge $NN_q(k) \subseteq DB$ mit exakt k Objekten, so daß
  $$\forall o' \in NN_q \; \forall o' \in DB \setminus NN_q: d(q,o) \leq d(q,o')$$

3.6.2 Grundalgorithmus ohne Index (nichtdeterministisch):

```
result: list of (dist: Real, P: Point) ordered by dist descending := ⟨⟩ ;
for i:=1 to k do
    insert ((distance (q, database [i]), database [i]), result) ;
for i:=k + 1 to n do
    if distance (q, database [i]) ≤ result [0].dist then
        delete_first (result) ;
        insert ((distance (q, database [i]), database [i]), result) ;
```


Anmerkung: Die Ergebnisliste *result* ist hier *absteigend* geordnet, obwohl dies der Intuition aus Anwendersicht widerspricht (Intuition: erstes Element = bestes Element = erster NN). Der Grund ist, daß bei absteigender Ordnung Datenstrukturen verwendet werden können, die einen besonders effizienten Zugriff auf das *erste* Element erlauben.

3.6.3 Algorithmen mit Index

Grundsätzlich lassen sich sowohl die Tiefensuche-Algorithmen als auch der Prioritätsalgorithmen erweitern so daß sie k nächste Nachbarn suchen. Als Pruning-Distanz dient grundsätzlich die Distanz des am weitesten entfernten Eintrags in der Ergebnisliste *result*[0].*dist*. (Liste ist hier absteigend sortiert)

Algorithmus von Roussopoulos, Kelley und Vincent:

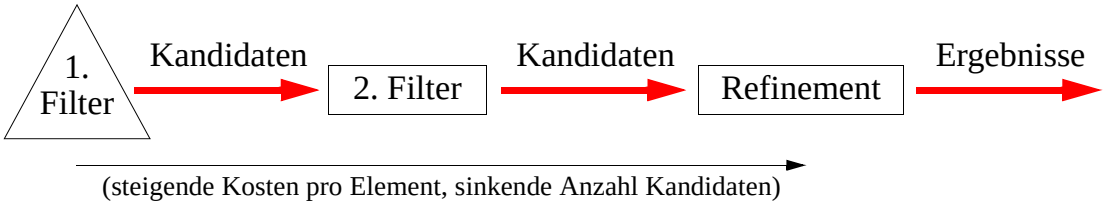
Die MINMAXDIST jeder einzelnen bekannten Seite kann grundsätzlich nur so gewertet werden, wie *ein* bekannter Punkt, nicht als Gesamt-Pruning-Distanz. Deshalb lohnt sich das MINMAXDIST-Konzept i.a. nicht für die k-NN-Suche.

3.7 NN-Suche mit mehrstufiger Anfragebearbeitung

- Häufig wird bei der Anfragebearbeitung eine Multistep-Architektur eingesetzt, die aus

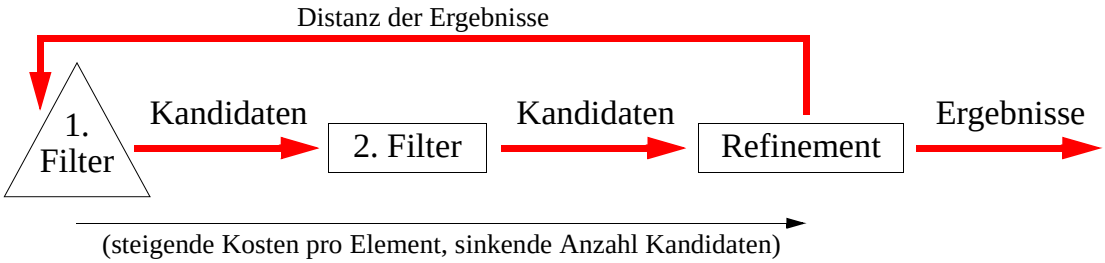
einem oder mehreren Filterschritten sowie einem Verfeinerungsschritt besteht

- Um die Vollständigkeit des Anfrageergebnisses garantieren zu können, ist die Lower-Bounding-Property nachzuweisen, d.h. die im Filter ermittelte Distanz ist höchstens so groß wie die Distanz des Verfeinerungsschrittes
- Analog gilt bei mehreren Filtern $\text{dist}_{\text{Filter1}} \leq \text{dist}_{\text{Filter2}} \leq \ldots$
- *Bereichsanfragen* können dann durch einfache kaskadierte Anwendung (ohne weitere Modifikation) der Filterschritte und des Verfeinerungsschrittes ausgewertet werden:



- Für Nearest-Neighbor-Queries gilt dies nicht, denn der im (1.) Filterschritt ermittelte nächste Nachbar ist nicht notwendig auch das Element mit minimaler $\text{dist}_{\text{Refinement}}$.
- Bei einer geeigneten Filterfunktion ist es aber *wahrscheinlich*, daß das Element mit minimaler $\text{dist}_{\text{Refinement}}$ auch unter *den ersten* Elementen des Filterschrittes ist.
- Man benötigt eine Art “Rückmeldung” der im Verfeinerungsschritt ermittelten Distanz

an den Filterschritt.



- Analog zu den NN-Algorithmen ohne Multistep-Architektur gibt es verschiedene Auswertungsstrategien, die sich in Speicherplatz- und Zeitkomplexität unterscheiden.
- Im folgenden: Ein Filterschritt, ein Verfeinerungsschritt
  Verfahren leicht übertragbar auf Architekturen mit mehreren Filterschritten.

3.7.1 Auswertung mit Bereichsanfrage

[KSF+ 96] Korn F., Sidiropoulos N., Faloutsos C., Siegel E., Protopapas Z.: *Fast Nearest Neighbor Search in Medical Image Databases*. VLDB 1996, 215-226.

[KSF+ 98] Korn F., Sidiropoulos N., Faloutsos C., Siegel E., Protopapas Z.: *Fast and Effective Retrieval of Medical Tumor Shapes*. TKDE 10(6), 1998, 889-904.

Auf Filterebene wird zunächst eine Nearest-Neighbor-Query und dann eine Range-Query ausgeführt.

Algorithmus:

R := FilterNearestNeighborQuery (*Q*) ;
ε := RefinementDist (*R*,*Q*) ; (* hierbei wird evtl. vollständige Info von *R* geladen *)
S := FilterRangeQuery (*Q*,ε) ;
resultdist := ε ; *result* := *R* ;
for each $P \in S$ **do**
  if RefinementDist (*P*,*Q*) ≤ *resultdist* **then**
    resultdist := RefinementDist (*P*,*Q*) ; (* hierbei lade evtl. vollst. Daten von *P* *)
    result := *P* ;

Idee:

- Sobald die (Verfeinerungs-) Distanz ε eines beliebigen Punktes bekannt ist, weiß man,
  — die Verfeinerungsdistanz des (eigentlichen) nächsten Nachbarn ist höchstens ε
  — auch die Filterdistanz des (eigentlichen) nächsten Nachbarn ist höchstens ε, denn es gilt ja immer Filterdistanz ≤ Verfeinerungsdistanz
- Der nächste Nachbar liegt also immer in der Ergebnismenge einer Range-Query, wenn man als Range ε die *Verfeinerungsdistanz* eines beliebigen Punktes *R* anwendet
- Ein sehr gut geeigneter Punkt *R* ist z.B. der nächste Nachbar von *Q* bzgl. der Filterdistanz, da dieser Punkt wahrscheinlich auch eine geringe Verfeinerungsdistanz aufweist

- Unter der (hoffentlich kleinen) Ergebnismenge der Bereichsanfrage wird mit dem Grundalgorithmus der nächste Nachbar bestimmt. Alle Punkte der Ergebnismenge werden verfeinert (teuer: meist 1 zusätzlicher Plattenzugriff pro Element)

Vorteile:

- Einfacher Algorithmus
- Schlanke Schnittstelle zur Datenbank: Lediglich durch die Funktionen
  — FilterNearestNeighborQuery
  — FilterRangeQuery
  — evtl. Zugriff auf Punktinformationen speziell für den Verfeinerungsschritt
  keine laufende Interaktion zwischen der Anfragebearbeitung auf Filter- und Verfeinerungsebene erforderlich, nur Funktionsaufruf und Ergebnisübergabe

Nachteile:

- Leistung stark abhängig von der Filterselektivität.
  Bei relativ schlecht funktionierendem Filter großes ε, und dadurch:
  — große Ergebnismenge der Bereichsanfrage
  — hohe Kosten für Verfeinerung dieser Ergebnisse

3.7.2 Auswertung mit unmittelbarer Verfeinerung

Jeder Punkt, der nicht aufgrund seiner Filterdistanz ausgeschlossen werden kann, wird unmittelbar verfeinert. Zu diesem Zweck wird ein beliebiger NN-Algorithmus um die entsprechenden Aufrufe des Verfeinerungsschrittes erweitert.

Algorithmus:

```
procedure SimpleNNQuery (q: Point; pa: DiskAddress)
    p := LoadPage (pa) ;
    if IsDataPage (p) then
        for i := 0 to p.num_objects do
            if FilterDistance (q, p.object [i]) ≤ resultdist then
                (* evtl. vollständige Daten zu p.object [i] laden *)
                if RefinementDistance (q, p.object [i]) ≤ resultdist then
                    result := p.object [i] ;
                    resultdist := RefinementDistance (q, p.object [i]) ;
    else
        for i := 0 to p.num_objects do
            if MINDIST (q, p.region [i]) ≤ resultdist then
                SimpleNNQuery (q, p.childpage [i]) ;
```


Vorteile

- (je nach eingesetztem NN-Algorithmus) gute Speicherplatzkomplexität, da keine Kandidaten zwischengespeichert werden
- Einfache Erweiterung eines NN-Grundalgorithmus

Nachteile

- Sehr hohe Verfeinerungskosten (fast alle Punkte), wenn eine der Bedingungen erfüllt:
  — Filter mit rel. schlechter Selektivität
  — NN-Algorithmus konvergiert langsam
- Keine schlanke Schnittstelle zum Datenbanksystem:
  Erweiterung muß direkt in die Implementierung des NN-Algorithmus eingebaut werden

3.7.3 Auswertung nach Priorität

[SK 98] Seidl T., Kriegel H.-P.: *Optimal Multi-Step k-Nearest Neighbor Search*. ACM SIGMOD 1998, 154-165.

Auf Filterebene läuft eine Ranking-Query. Der Algorithmus ruft so lange die GetNext-Funktion dieses Rankings auf, bis die Filterdistanz des erhaltenen Objekts die beste Verfeinerungsdistanz überschreitet.

Algorithmus

```
FilterInitializeRanking (Q) ;
result := ⊥ ;
resultdist := +∞ ;
repeat
    P := FilterGetNext () ;
    fildist := FilterDistance (P,Q) ; (* liefert eigentlich FilterGetNext *)
    if RefinementDistance (P,Q) ≤ resultdist then
        result := P ;
        resultdist := RefinementDistance (P,Q) ; (* zwischenspeichern!!! *)
until fildist > resultdist ;
```

Vorteile

- Beweisbar, daß eine *minimale* Anzahl von Kandidaten verfeinert wird
  (analog zu Optimalitätsbeweis in 3.5.6 für Prioritätssuche nach NN ohne Multistep)
- Einfach, schlanke Schnittstelle zum Datenbanksystem über Ranking-Query

Nachteile

- Komplexität des Ranking-Algorithmus (Speicher und/oder Zeit)

3.7.4 *k*-Nearest-Neighbor-Queries und Ranking

Alle drei Alternativen leicht auch zu *k*-nearest-neighbor-Algorithmen erweiterbar

- Bereichsanfrage: Ersetze die NN-Query im Filter durch *k*-NN-Query und passe die Auswertung auf der Ergebnismenge der Bereichsanfrage an
- Unmittelbare Verfeinerung:
  Erweitere statt NN-Algorithmus einen *k*-NN-Algorithmus um die Refinement-Aufrufe
- Auswertung nach Priorität:
  Verwalte statt *result* eine Liste von Punkten und benutze den *k*-ten NN für das Abbruchkriterium

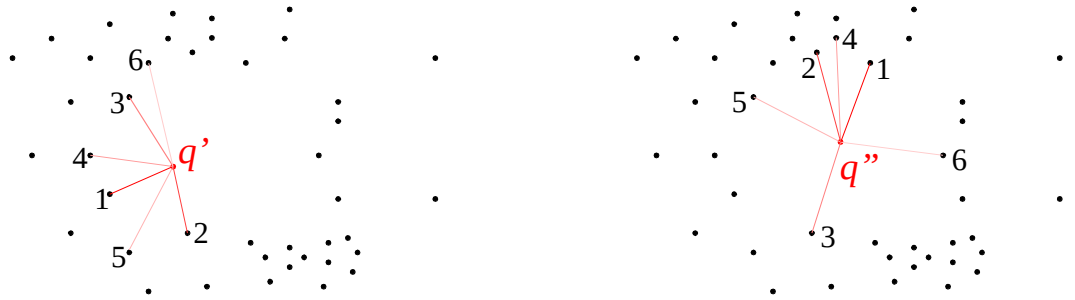
Ranking-Algorithmen sind sehr wichtig, da sie für den 2. (usw.) Filterschritt benötigt werden. Nicht alle Alternativen lassen sich zu Ranking-Algorithmen erweitern

- Bereichsanfrage: Nicht möglich, da sich kein Bereich ε ermitteln läßt
- Unmittelbare Verfeinerung:
  Erweitere einen Ranking-Algorithmus um das Konzept
- Auswertung nach Priorität:
  — Verwalte unbegrenzte Liste von Punkten
  — Abbruchkriterium ist die Validierung des ersten Punktes

3.8 Ranking-Anfragen

3.8.1 Allgemeines

- Der Benutzer gibt ein Anfrageobjekt *q* vor und initialisiert damit die Anfragebearbeitung
- Der Benutzer kann dann mehrfach eine Funktion get_next aufrufen, die ihm jeweils den 1., 2., usw. Nachbarn zurückgibt.



3.8.2 Basisalgorithmus ohne Index

(siehe Übung)

3.8.3 Algorithmus mit Indexunterstützung

[HS 95] Hjaltason G. R., Samet H.: *Ranking in Spatial Databases*, Int. Symp. on Large Spatial Databases (SSD), 1995.

Prinzipiell können sowohl die Tiefensuche-Algorithmen als auch der Prioritätsalgorithmus zur *k*-NN-Suche zu einem Ranking-Algorithmus erweitert werden. Da die Algorithmen jedoch nachdem der *i*-te Nachbar gefunden wurde, das Ergebnis an den Aufrufer übergeben, und beim nächsten Aufruf an derselben Stelle der Anfragebearbeitung wiederaufsetzen, eignen sich die rekursiven Versionen nicht sehr gut. Sie müssten entrekursiviert werden, und der Zustand des Algorithmus müsste vollständig in globalen Variablen gespeichert werden. Der Prioritätsalgorithmus eignet sich besser, weil der gesamte algorithmische Zustand in den beiden Listen *apl* und *result* gespeichert ist.

Im Gegensatz zum *k*-NN-Algorithmus hält der Ranking-Algorithmus eine unbeschränkte Ergebnisliste. Jeder Punkt, der auf einer betrachteten Datenseite liegt, wird eingefügt. Entsprechend ergibt sich auch keine Pruning-Distanz, d.h. jede Kindseite einer geladenen Seite wird in die *APL* eingefügt.

Der Algorithmus stoppt (vorläufig), sobald das Element der Ergebnisliste mit der geringsten Distanz zum Anfragepunkt validiert ist, d.h. sobald die erste Seite der APL eine größere MINDIST als dieses Element aufweist. In diesem Fall wird das Element aus der Ergebnisliste gelöscht und an den Aufrufer übergeben. Beim nächsten Aufruf von get_next() wird wie vorher mit den beiden Listen *apl* und *result* weitergearbeitet.

Im Gegensatz zu *k*-NN benötigt der Ranking-Algorithmus *result* in aufsteigender Sortieung.

- Algorithmus (Initialisierung):
  apl: **list of** (*dist*: Real, *pa*: DiskAdr) **ordered by** *dist* **ascending** := ⟨(0.0, *root*)⟩ ;
  result: **list of** (*dist*: Real, *P*: Point) **ordered by** *dist* **ascending** := ⟨⟩ ;

- Algorithmus:
  while not empty (*apl*) and *apl* [0].*dist* ≤ *result* [0].*dist* **do**
  p := LoadPage (*apl* [0].*pa*) ;
  delete_first (*apl*) ;
  if (IsDataPage (*p*)) **then**
  for *i* := 0 **to** *p.num_objects* **do**
  (* Jeden Punkt einfügen *)
  insert ((distance (*q*, *p.object* [*i*]), *p.object* [*i*]), *result*) ;
  else
  for *i* := 0 **to** *p.num_objects* **do**
  (* Jede Seite einfügen *)
  insert ((MINDIST (*q*, *p.region* [*i*]), *p.childpage* [*i*]), *apl*) ;
  resultpoint := *result* [0].*P* ;
  delete_first (*result*) ;

- Speicherbedarf sehr hoch. Im worst case liegt nahezu die gesamte Datenbank in *result*.