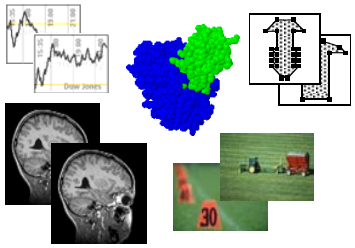
Kapitel 2
Prinzipien der Anfragebearbeitung

Skript zur Vorlesung: Neue Trends zur Suche in modernen Datenbanksystemen
Wintersemester 2013/14, LMU München

2.1 Prinzip der Ähnlichkeits-/Nachbarschaftssuche

2.1.1 Prinzip der feature-basierten Ähnlichkeit

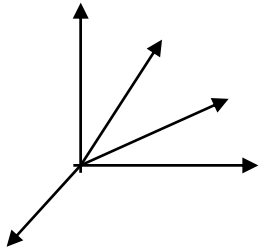
– Extrahiere charakteristische (nummerische) Eigenschaften („Features“) aus den Objekten



Anwendungsgebiete (Objektraum)

Feature-Transformation

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



Featurevektor-Raum

- Ähnlichkeitsanfrage wird auf Nachbarschaftsanfrage zurückgeführt
- Indexstrukturen für:
  – Multidimensionale Vektoren
  – Allgemein metrische Daten (beliebige Objekte, auf denen eine metrische Distanzfunktion definiert ist)

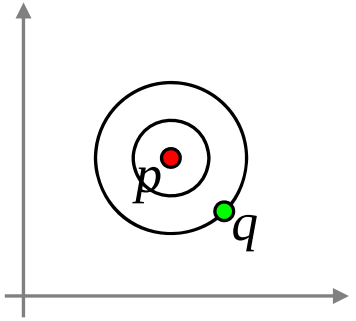
2.1.2 Ähnlichkeit und Nachbarschaft

– Distanzmaße in Vektorräumen (Ähnlichkeitssuche)

- Euklidische Norm (L_2):

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

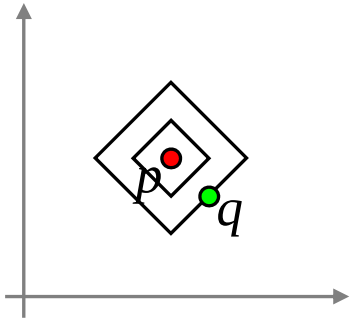
  Natürliches Distanzmaß



- Manhattan-Norm (L_1):

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

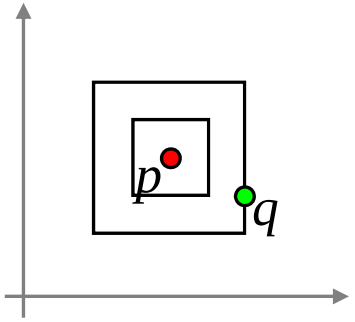
  Die Unähnlichkeiten der einzelnen Merkmale werden direkt addiert



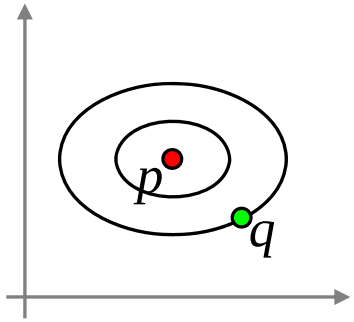
- Maximums-Norm (L_∞):

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

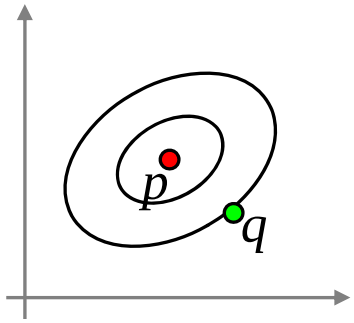
  Die Unähnlichkeit des am wenigsten ähnlichen Merkmals zählt



- Verallgemeinerung L_p-Abstand: $dist_p = (|p_1-q_1|^p + |p_2-q_2|^p + ...)^{1/p}$
- Gewichtete Euklidische Norm:
  - $dist = (w_1(p_1-q_1)^2+w_2(p_2-q_2)^2+...)^{1/2}$
  - Häufig sind die Wertebereiche der
  - Merkmale deutliche unterschiedlich
  - Beispiel: Merkmal $M_1 \in [0.01 \ldots 0.05]$
    Merkmal $M_2 \in [3.07 \ldots 22.2]$
  - Damit M1 überhaupt berücksichtigt
  - wird muss es höher gewichtet werden

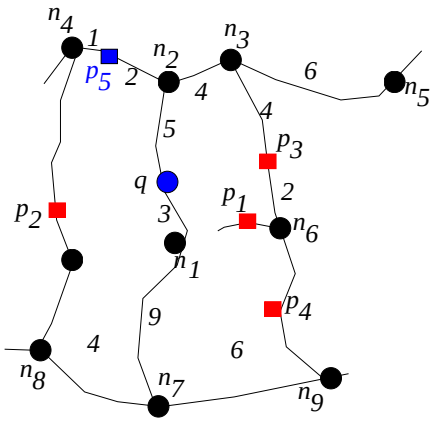


- Quadratische Form:
  - $dist = ((p - q)\ \mathbf{M}\ (p - q)^T)^{1/2}$
  - Bisherige Abstandsmasse gewichten
  - Merkmale nur getrennt
  - Besonders bei Farbhistogrammen müssen
  - verschiedene Merkmale gemeinsam
  - gewichtet werden



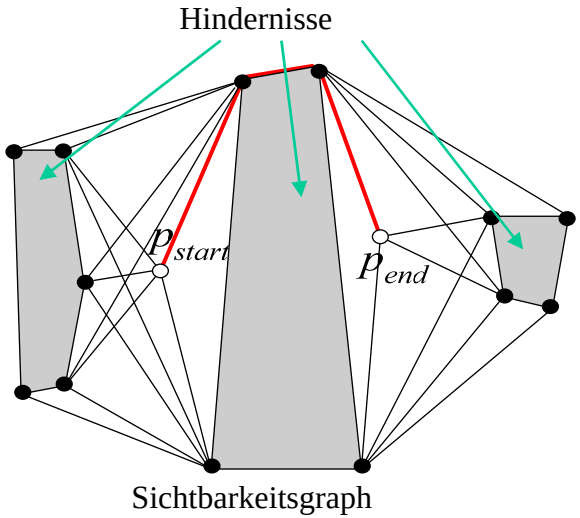
– Distanzmaße in Netzwerkgraphen

- Netzwerkdistanz
  - Netzwerkgraph G = (V,E)
    - V := Menge von Knoten
    - E := Menge von Kanten
    - Kantengewicht c: E → $\Re^+$
  - Pfad $P_{s,t} = <v_1,\ldots,v_n>, s=v_1, t=v_n$
    - $v_1,\ldots,v_n \in V$
    - $\forall 1 \le i \le n-1: \exists e \in E: e=(v_i,v_{i+1})$
    - Pfadkosten $c(P_{s,t})=\Sigma_{1 \le i \le n-1} c(v_i,v_{i+1})$



  - Distanz (Netzwerkdistanz)
    - » $d_{net}(v_s,v_t)$:= Pfad $P_{s,t}$ von Knoten *s* zu Knoten *t* mit minimalen Kosten $c(P_{s,t})$
- Anwendung:
  - Nachbarschaftsanfragen:
    - » Suche alle Objekte in einem Umkreis von 1 KM.
  - Routing in Verkehrsnetzen:
    - » Suche den schnellsten Weg von A nach B.
  - Verkehrsplanung

– Distanz in Räumen mit Hindernissen
  - Kombination aus Euklidischer Distanz und Netzwerkdistanz
  - Distanz zwischen zwei Punkten p_{start} und p_{end}
    = Netzwerkdistanz $d_{net}(p_{start},p_{end})$ im Sichtbarkeitsgraph



  - Anwendungen:
    – Nachbarschaftsanfragen und Pfadsuche in Gebäuden

2.1.3 Typen von Nachbarschafts-/Ähnlichkeitsanfragen

Für alle nachfolgenden Anfragen sei gegeben:

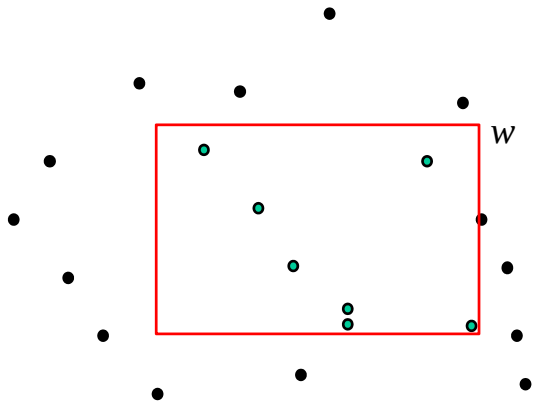
- Mehrdimensionaler Vektorraum $\Re^d$
- Menge von Punktobjekten DB = $\{o_1,\ldots,o_n\} \in \Re^d$

– Fensteranfrage

- Benutzer gibt Anfragefenster $w \in \Re^d \times \Re^d$ vor.
- Ergebnis enthält alle Objekte, die innerhalb von Anfragefenster *w* liegen.

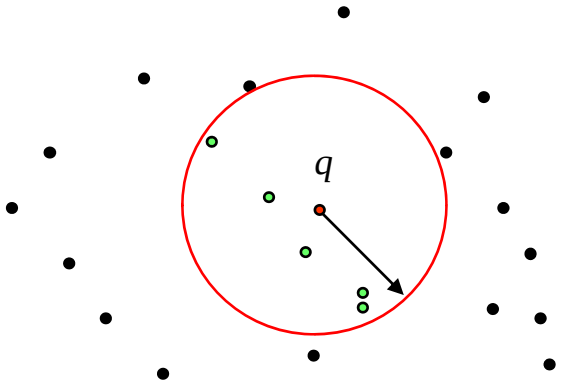
- Basisalgorithmus (sequentieller scan)

```
WQ-SeqScan(DB, q, ε)
    result = ∅;
    FOR i=1 TO n DO
        o = DB.getObject(i);
        IF o within w THEN
            result := result ∪ o;
    RETURN result;
```



– Bereichsanfrage

- Benutzer gibt Anfrageobjekt $q \in \Re^d$ 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 (sequentieller scan)

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

– k-Nächste Nachbar (*k*-NN) Anfrage

- – Benutzer gibt Anfrageobjekt $q \in \Re^d$ und Anfrageparameter $k \in \aleph^+$ vor.
- – Ergebnis enthält (mindestens/die) *k* Objekte, die die geringste Distanz zu *q* haben.
- – Mehrdeutigkeiten müssen wiederum sinnvoll behandelt werden

- Formal

Deterministisch:

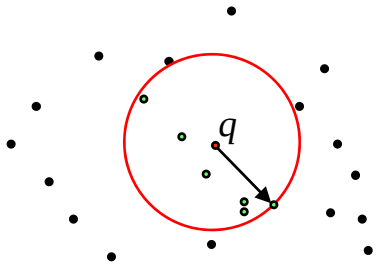
kleinste Menge *NN*(*q*,*k*) ⊆ DB mit mindestens *k* Objekten, sodass

$$\forall o \in NN(q,k), \forall o' \in DB - NN(q,k) : dist(q,o) < dist(q,o')$$

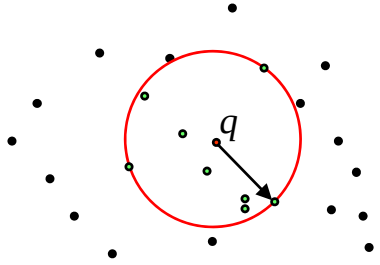
Nicht-deterministisch:

Menge *NN*(*q*,*k*) ⊆ DB mit exakt *k* Objekten, sodass

$$\forall o \in NN(q,k), \forall o' \in DB - NN(q,k) : dist(q,o) \leq dist(q,o')$$



Eindeutiges Ergebnis



Mehrdeutiges Ergebnis

- Basisalgorithmus (sequential scan): nichtdeterministisch

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

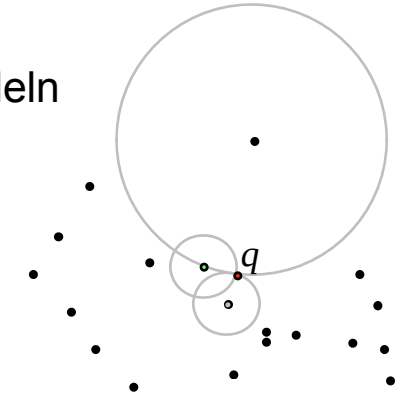
  - Bemerkung:

    Liste result wird als Heap anstelle einer sortierten Liste implementiert.

    Grund: Die Liste result braucht nicht vollständig sortiert sein – es reicht sicherzustellen, dass das erste Element in result immer den größten Distanzwert hat. Ermöglicht Einfügen in O(log(k)) .

– Reverse *k*-Nächste Nachbar (R*k*-NN) Anfrage

- Benutzer gibt Anfrageobjekt $q \in \Re^d$ und Anfrageparameter $k \in \aleph^+$ vor.
- Ergebnis enthält alle Objekte, die *q* in ihrer *k*-nächsten Nachbarschaft enthalten.
- Mehrdeutigkeiten (wie bei *k*-NN) entsprechend behandeln



R1-NN Anfrage

- Formal:

$$RNN(q,k) = \{o \in DB \mid q \in NN(o,k)\}$$

- Basisalgorithmus (sequential scan): nichtdeterministisch

```
RNN-SeqScan(DB, q, k)
    result = ∅;
    FOR i=1 TO n DO
        o = getObject(i);
        neighbors = k-NN-SeqScan(DB, o, k);
        IF q ∈ neighbors THEN
          result := result ∪ o;
    RETURN result;
```

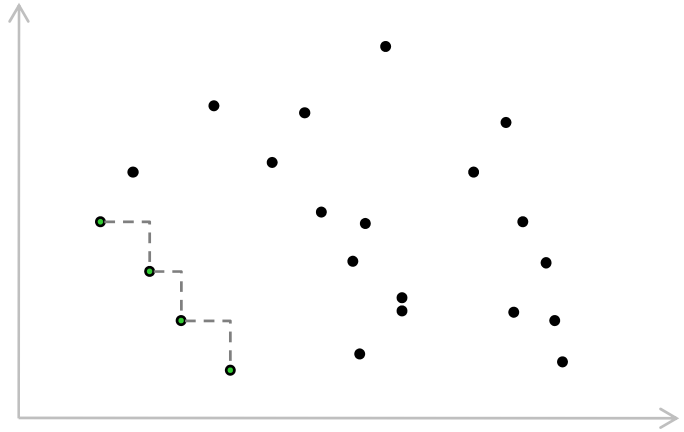
– Skylineanfrage
  – Gegeben: Menge von Objekten mit positiven Attributwerten $a_i \in \Re^+$ ($1 \le i \le d$).
  – Ergebnis enthält alle Objekte die von keinem anderen Objekt *dominiert* werden, d.h. kein anderes Objekt ist „besser“ bzgl. aller Attribute.

• Formal:

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

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

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



Skyline Anfrage

- Basisalgorithmus (sequential scan):

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

zusätzlicher sequentieller scan über DB

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

  zum Beispiel:

  » *Constrained Skyline*:

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

  » *Ranked Skyline*:

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

  » weitere Varianten:

  Group-by Skyline, *Dynamic Skyline*, *K-dominating Queries*, etc.

2.1.4 Algorithmische Paradigmen zur effizienten Nachbarschaftssuche

- Naive (sequentielle) Suche
  - Für alle *n* Objekte der Datenbank wird das Anfrage-Prädikat (d.h. meist eine Distanzberechnung zum Anfrageobjekt) ausgewertet
  - Kosten: O(*n* · Kosten für das Anfrage-Prädikat)

  zu teuer für
  - große Datenbanken
  - Anwendungen mit teurer Auswertung des Anfrageprädikats

- **Indexbasierte Suche**

  Organisation der Objekte sodass bei der Suche bereits ein Großteil der Datenbank effizient ausgeschlossen werden kann. (Pruning)

- **Mehrstufige Anfragebearbeitung**
  - (Räumliche) Approximation der Objekte
  - Effiziente Filterung von Kandidaten in einem Filterschritt

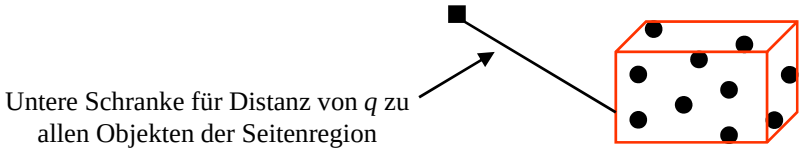
2.1.5 Indexbasierte Suche

Prinzip: [Böhm, Berchtold, Keim. ACM Computing Surveys, 2001]

- Indexierung über einen räumlichen Schlüssel (Struktur)
  - Räumlich nah beieinanderliegende Objekte werden häufig zusammen angefragt.
  - Gruppierung und Speicherung von räumlich nah-beieinanderliegenden Objekte.
  - Jeder Gruppe wird ein räumlicher Schlüssel zugeordnet.
- Bei der Suche
  - Zugriff nur auf relevante Objekte (über Auswertung des räumlichen Schlüssels).
  - Bei (räumlich) selektiven Anfragen kann ein Großteil der Objekte von der Ergebnismenge ausgeschlossen werden.

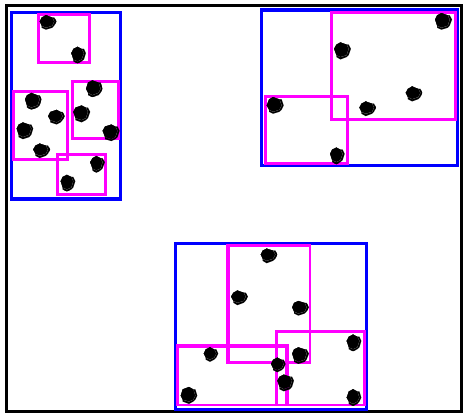
- Hierarchische Indexstrukturen (oft verwendet)
  - Objekte sind Baumartig organisiert
  - Jedem Knoten des Baumes ist zugeordnet
    - Seite des Hintergrundspeichers
    - Region des Datenraums
  - Typen von Knoten (= Seiten)
    - Blattknoten sind Datenseiten, speichern Objekte
    - Innere Knoten sind Directoryseiten, speichern Directory-Einträge
      - » Verweis zur Kindseite (Adresse auf dem Hintergrundspeicher)
      - » Beschreibung der Region der Kindseite
  - Jedes Objekt wird in genau einer Datenseite gespeichert
  - Regionen gewährleisten, dass ähnliche Objekte möglichst auf den selben Datenseiten (oder Teilbäumen) gespeichert werden

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



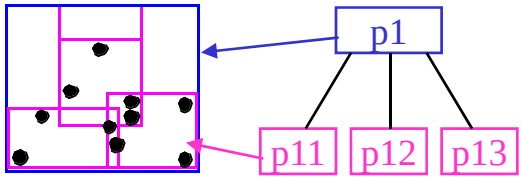
  - Damit kann man Vollständigkeit der Anfrageergebnisse gewährleisten
- Bäume sind meist balanciert (alle Blätter auf dem selben Level)
- Indexstrukturen sind dynamisch, d.h. Insert/Delete sind effizient
  - Tiefensuche nach entsprechender Datenseite (auf einen Pfad beschränkt)
  - Einfügen/Löschen des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

– Beispiel: R-Baum
  - Verallgemeinert die Idee des B+-Baums auf den mehr-dimensionalen Raum
  - Ursprünglich entworfen zur Organisation von (2D) Rechtecksdaten.
  - Oft verwendet für mehr-dimensionale Vektordaten (Punkte sind spezielle Rechtecke).
  - Basiert auf der Technik überlappender Seitenregionen

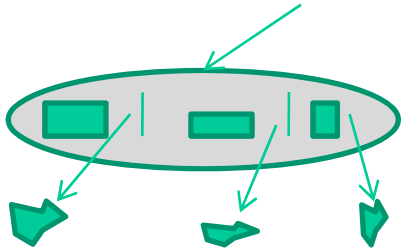


- Struktur des R-Baums (R*-Baum):
  - Hierarchische Indexstruktur
  - Minimal-umgebende Hyperrechtecke

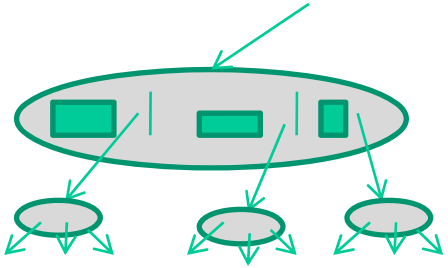
  als räumliche Schlüssel



  - Parameter:
    - m : minimaler Füllgrad der Knoten, M : maximaler Füllgrad der Knoten
    - Es gilt: $2 \leq m \leq \lceil M/2 \rceil$
  - Datenseite:
    - Einträge der Form (Rectangle, Verweis auf die exakte Beschreibung, weitere Attribute)
    - Rectangle : minimal-umgebendes Rechteck (MUR), zur Approximation der exakten Objektbeschreibung

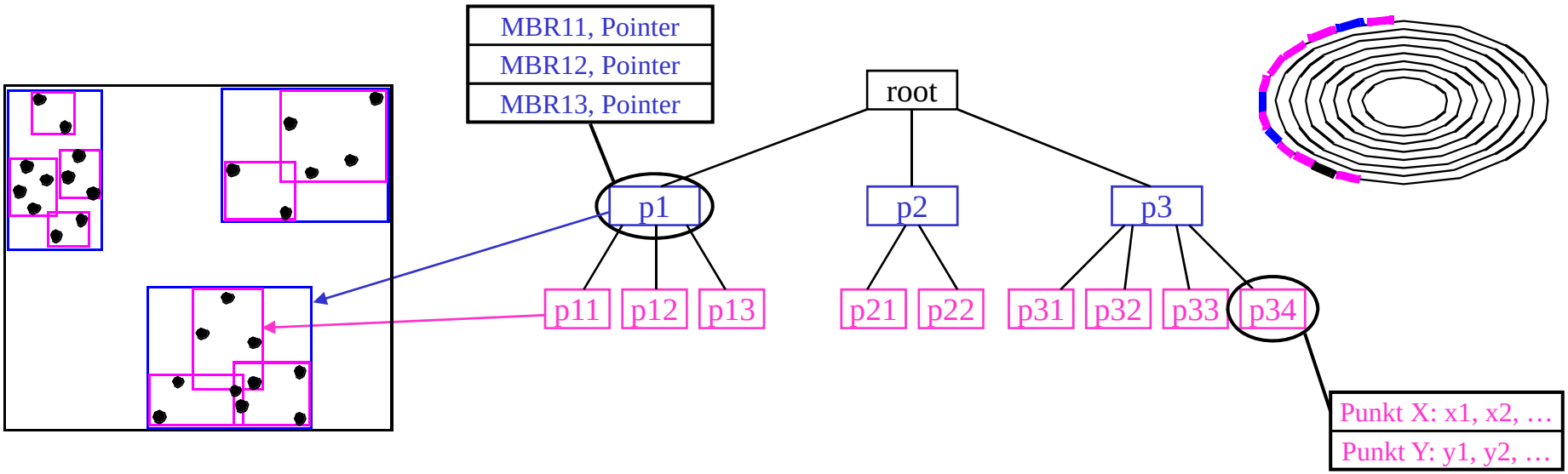


  - Directoryseite:
    - Indexeinträge der Form (Rectangle, Subtree^)
    - Rectangle: MUR um alle Einträge des darunterliegenden Teilbaums.
    - Subtree^: Verweis auf Teilbaum (Seitenadresse)



- Alle Datenseiten sind Blätter des Baums.
  → Der Baum ist vollständig balanciert
  d.h. alle Pfadlängen von der Wurzel zu einem Blatt sind gleich.
- Höhe des Baumes:
  Ist N die Anzahl der gespeicherten Datensätze,
  so gilt für die Höhe h des R-Baumes: $h \leq \lceil \log_m N \rceil + 1$

  Die Höhe eines R-Baums ist also O(log N).



– Aufbau des Baumes:
  - Optimierungsziele:
    – geringe Überlappung der Seitenregionen
    – Seitenregionen mit geringem Flächeninhalt
      => geringe Überdeckung von totem Raum
    – Seitenregionen mit geringem Umfang
  - Einfügen von einem Punkt/Rechteck R

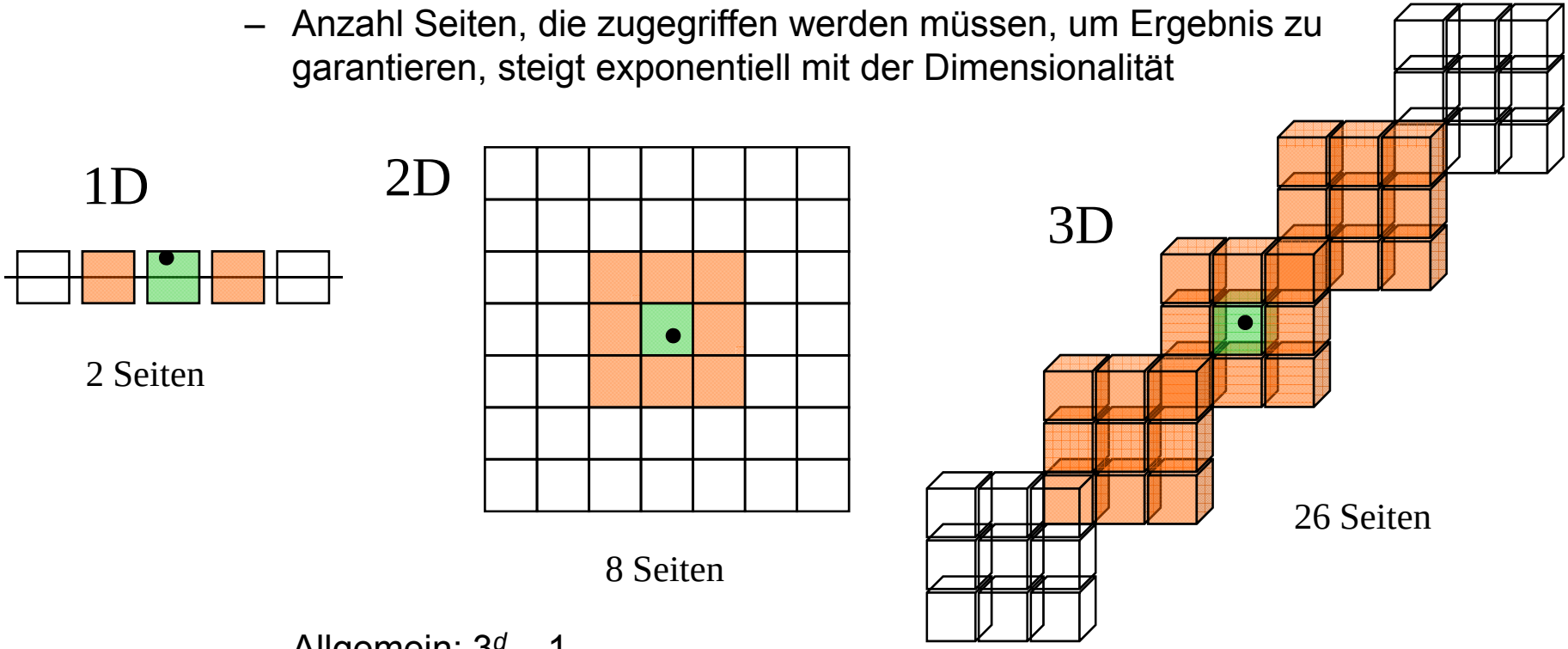
    (1) Suche nach (Blatt-) Knoten in dem R eingefügt werden soll:
    – **Fall 1:** R fällt vollständig in genau ein Directory-Rechteck D
      → Einfügen in Teilbaum von D
    – **Fall 2:** R fällt vollständig in mehrere Directory-Rechtecke D1, ... , Dn
      → Einfügen in Teilbaum von D_i, das die geringste Fläche aufweist
    – **Fall 3:** R fällt vollständig in kein Directory-Rechteck
      - *D verweist auf Directoryseite:*
      → Einfügen in Teilbaum von D, das den geringsten Flächenzuwachs erfährt (in Zweifelsfällen: .., das die geringste Fläche hat)
      → D muß entsprechend vergrößert werden.
      - *D verweist auf Datenseite:* → Einfügen sodass Überlappung minimal

- Der Knoten K läuft mit |K| = M+1 über:
  => Aufteilung auf zwei Knoten K1 und K2,
  sodaß |K1| ≥ m und |K2| ≥ m
- R-baum-varianten unterscheiden sich in der Splitstrategie (erschöpfende Suche (optimal), quadratischer Algorithmus, linearer Algorithmus (R-Baum))
- Splitstrategie des R*-Baum:
  (1) Bestimmung der Split-Dimension:
  Für jede Dimension:
  - » Sortiere Rechtecke gemäß beider Extremwerte.
  - » Für jede der beiden Sortierungen werden M-2m+2 Aufteilungen der M+1 Rechtecke bestimmt, so daß die 1. Gruppe der j-ten Aufteilung die ersten m-1+jRechtecke und die 2. Gruppe die übrigen Rechtecke enthält.
  - » Summiere die Umfänge (MURs) beider Gruppen für alle betrachteten Aufteilungen.

  – Dimension mit dem geringster Umfangsumme wird als Split-dimension gewählt.

  (2) Bestimmung der Aufteilung: Aufteilung mit geringster Überlappung, im Zweifelsfall geringster Überdeckung von totem Raum.

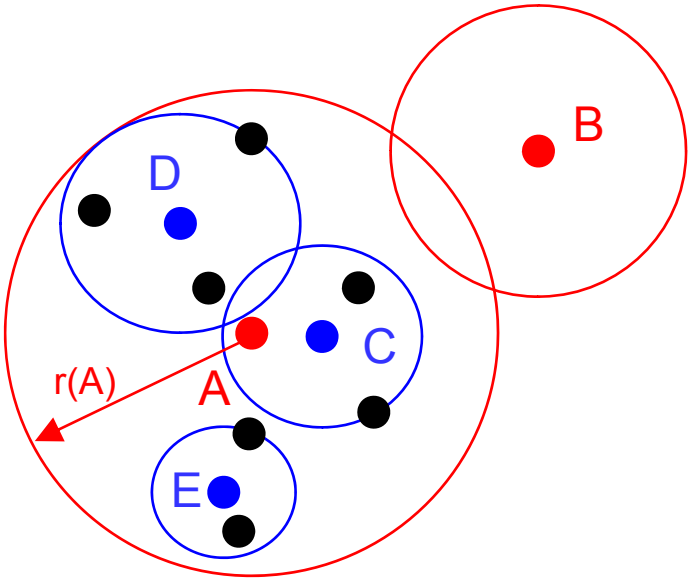
- Curse of Dimensionality (Vektordaten)
  - Anfrageleistung von Indexstrukturen verschlechtern sich mit zunehmender Dimension
    - Anzahl Seiten, die zugegriffen werden müssen, um Ergebnis zu garantieren, steigt exponentiell mit der Dimensionalität

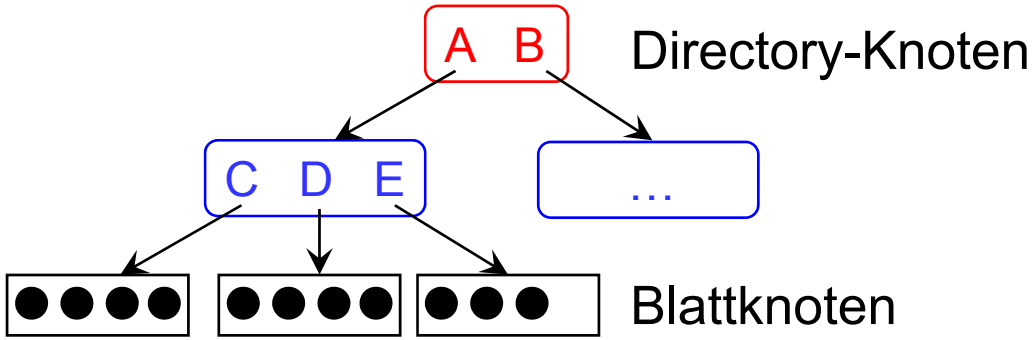


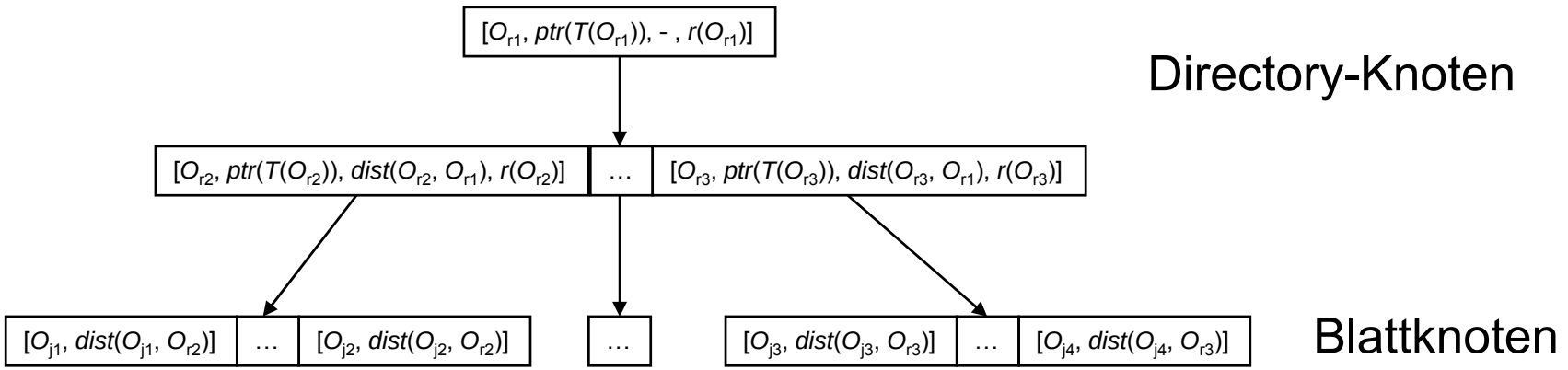
    - Allgemein: $3^d - 1$
    - Seitenregionen überlappen sich stärker
    - Folge: oft muss komplettes Directory gelesen werden

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

– M-tree Struktur







– 2.1.6 Mehrstufige Anfragebearbeitung

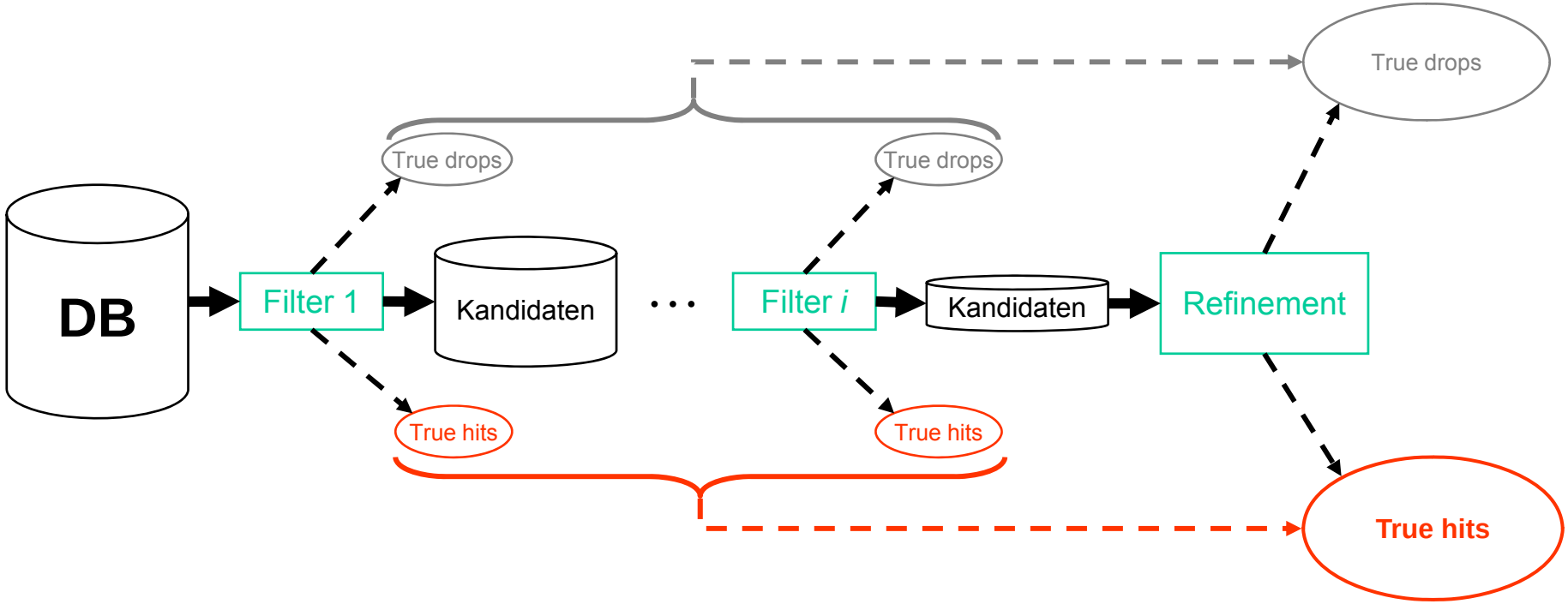
Idee:

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

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

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

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



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

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

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

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

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

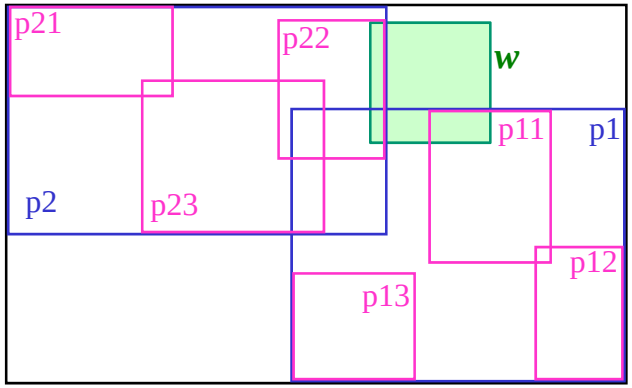
2.2 Indexbasiere Nachbarschaftssuche

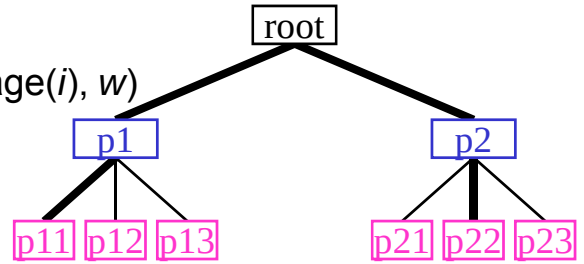
2.2.1 Indexbasierte Fensteranfrage

- Algorithmus

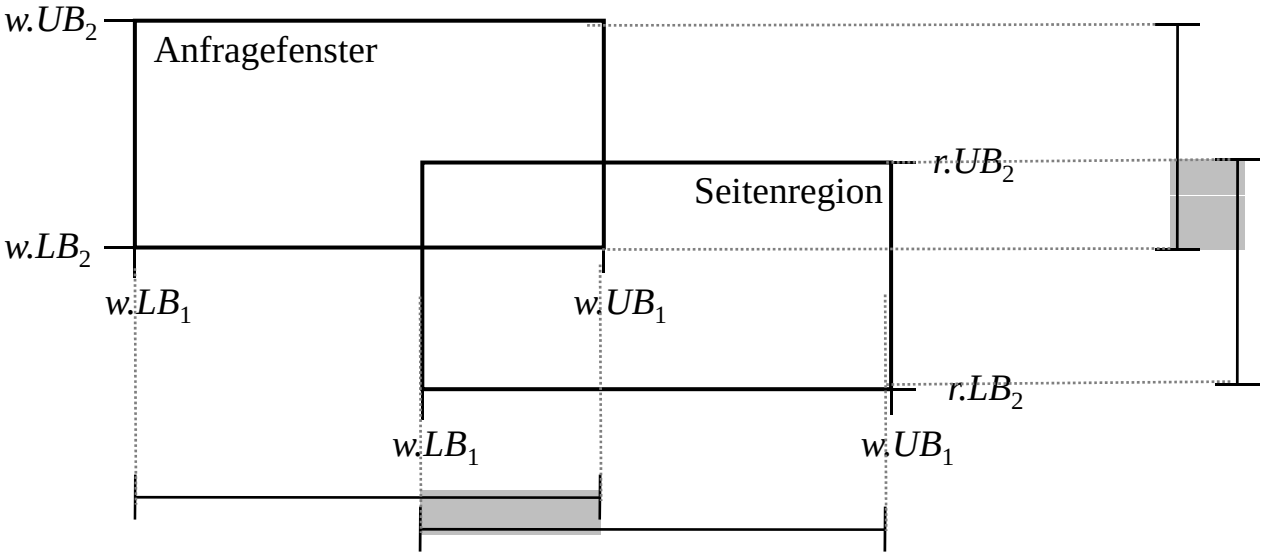
```
// w := Anfragefenster
WQ-Index(pa,w)          // pa initialisiert mit Diskadress der Wurzel des R*-Baums
    result = ∅;
    p := pa.loadPage();
    IF p.isDataPage() THEN
        FOR i=0 TO p.size() DO
            o = p.getObject(i);
            IF o ∈ w THEN
                result := result ∪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

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





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

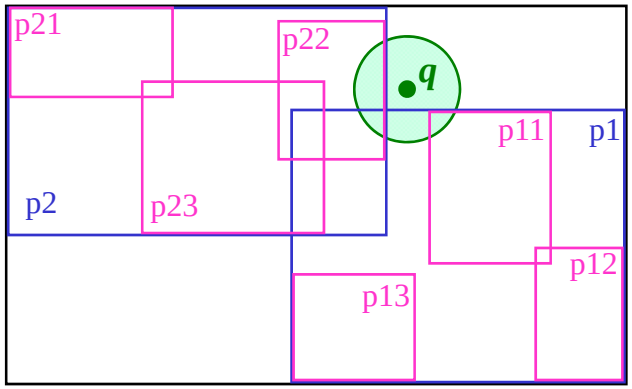


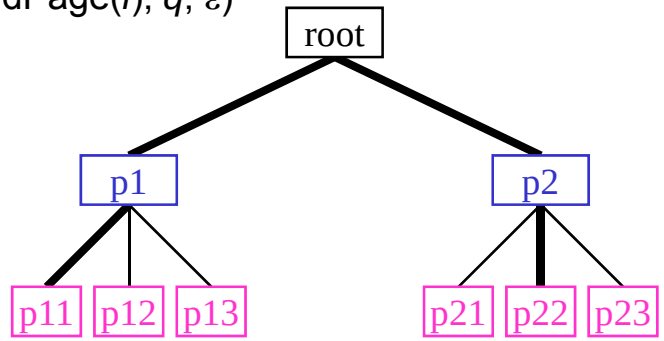
2.2.2 Indexbasierte Bereichsanfrage

- Algorithmus:

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

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





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

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