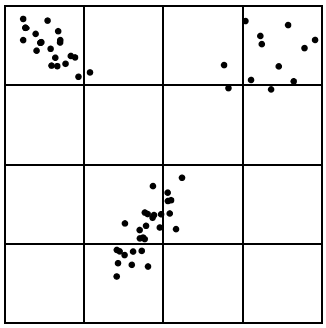
6 Suchstrukturen für multidimensionale Punktdaten

6.1 Motivation

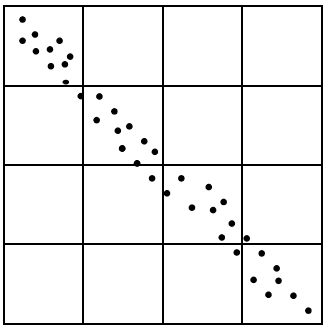
Die bisher vorgestellten Hashverfahren zeigten eine gute Leistungsfähigkeit bei

- Gleichverteilungen,
- Unabhängigkeit der Attribute.

Für ungleichverteilte Daten insb. *abhängige Ungleichverteilungen* sind diese Verfahren aber weniger geeignet:



Ungleichverteilung



abhängige Attribute

Probleme bei der Verwendung von Hashverfahren:

Verfahren ohne Directory:

- fehlende Adaptivität der Datenseiten

  d.h. solche Verfahren haben keine ausreichende Möglichkeit, die Regionen der Datenseiten an die Verteilung der Daten anzupassen.

  $\Rightarrow$ hohe Anzahl von Überlaufseiten

Verfahren mit ein- oder zweistufigem Directory:

- fehlende Adaptivität des Directory

  d.h. das Directory kann sich nicht oder nur eingeschränkt an die Datenverteilung anpassen.

  $\Rightarrow$ starkes Wachstum des Directory

Probleme bei der Verwendung von B-Baum-basierten Indexstrukturen:

- Für die Speicherung mehrdimensionaler *geometrischer Daten*, wie z.B. Punktdaten in der Ebene, eignen sich MDB- und kB-Bäume (und deren Varianten) nicht.

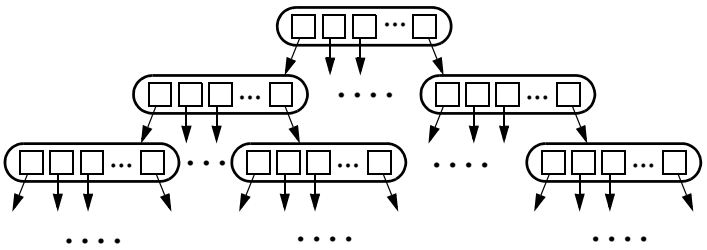
  Ursache:
  Die Dimensionen werden nicht gleichberechtigt behandelt.

Gesucht:

- Eine Partitionierungsstrategie, die
  - die Dimensionen gleichberechtigt behandelt,
  - die sich an die gegebene Datenverteilung entsprechend anpasst (gleichmäßige Speicheplatzausnutzung) und
  - die sich durch eine Baumstruktur angemessen repräsentieren lässt.

Idee *(Verfahren mit Hashbaum-Directory)*

- ***mehrstufiges Directory***:
  - Die Höhe des Directory passt sich der Zahl der gespeicherten Datensätze an.
  - Das Directory hat eine *Baumstruktur*.



- ***Hash-Organisation einer Directoryseite***
  - Die Einträge in der Directoryseite beschreiben die Region, die die zugehörige Sohnseite einnimmt *(Seitenregion)*.
  - Diese Seitenregionen werden durch einfache geometrische Körper (z.B. Rechtecke, Binärregionen) oder einfache räumliche Separatoren beschrieben.

Partitionierung des Datenraumes

Mehrdimensionale Indexstrukturen teilen den Datenraum D in m Seitenregionen S_1 bis S_m auf. Die Partitionierungen der bisher vorgestellten Hashverfahren waren:

- *rechteckig* (alle S_i bilden Hyperrechtecke)
- *vollständig* ($D = \bigcup_{i=1}^{m} S_i$)
- *disjunkt* ($S_i \cap S_j = \varnothing$ für alle $1 \le i,j \le m, i \ne j$)

Verfahren mit Partitionierungen, die diese drei Eigenschaften besitzen, können aber die Anforderung nach gleichbleibender Effizienz bei beliebiger Verteilung der Daten nicht erfüllen [See 89].

6.2 Z-Ordnung und Quadtrees

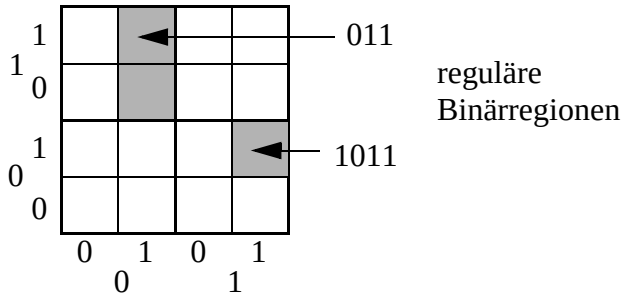
Grundidee:

- Die Seitenregionen von Datenknoten entsprechen *Binärregionen* (siehe unten) und werden über Binärfolgen beschrieben.
- Es wird eine *max. Auflösung (max. Level)* L_{max} bestimmt. Alle Binärfolgen, deren Länge kleiner als L ist, werden um *0*-Bits verlängert.
- Diese Binärfolgen werden als binäre Darstellung ganzer Zahlen interpretiert und dienen als Schlüssel in einem normalen eindimensionalen B^+-Baum.
- Da diese Binärfolgen Regionen unterschiedlicher Größe repräsentieren können, benötigen wir zusätzlich zur Unterscheidung die ursprüngliche Länge der Binärfolge *(Level)*.

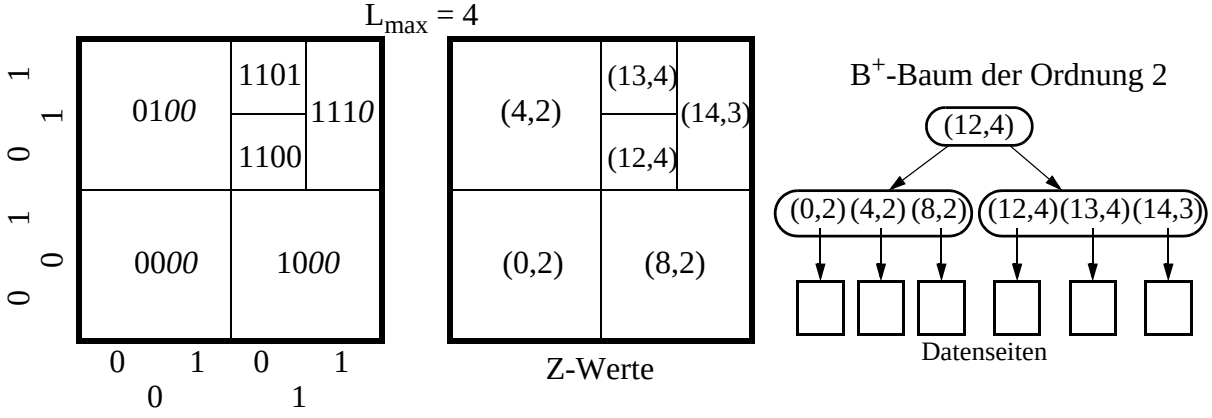
  Binärregionen

  - *Generierung*:
    durch fortgesetztes Halbieren des Gesamtdatenraums bzgl. jeder der Dimensionen.
  - *Repräsentation durch Bitfolge (Binärfolge):*
    die Binärfolge gibt an, welche Hälfte des aufzuteilenden Intervalls repräsentiert wird.
  - *Level (Auflösung):*
    entspricht der Anzahl der Bits in der Binärfolge;
    bestimmt die Größe der Binärregion.
  - *reguläre Binärregion*:
    Binärregion, die durch zyklischen Wechsel der Splitachse entstanden ist.

    Beispiel:

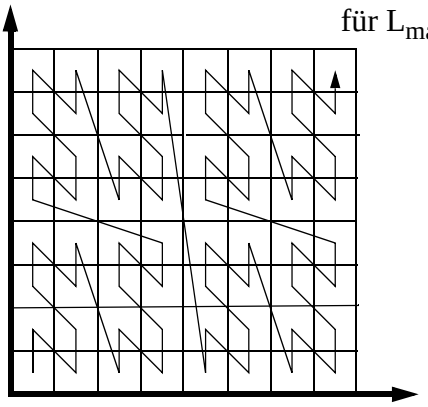


- ***Z-Wert***
  Das so entstandene Paar bestehend aus interpretierter Binärfolge und Level wird als *Z-Wert* bezeichnet.
- Die Z-Werte in einem Blatt des B^+-Baumes repräsentieren Seitenregionen und verweisen daher auf eine Datenseite.



- ***Z-Ordnung***
  ist die Ordnung, die aus der Interpretation der Z-Werte resultiert.

  Beobachtung:
  Z-Werte die in ihrer Ordnung direkt aufeinanderfolgen, sind auch oft räumlich benachbart.
  Damit wird indirekt eine *räumliche Ordnungserhaltung* erzielt.



für $L_{max} = 6$

21	23	29	31	53	55	61	63
20	22	28	30	52	54	60	62
17	19	25	27	49	51	57	59
16	18	24	26	48	50	56	58
5	7	13	15	37	39	45	47
4	6	12	14	36	38	44	46
1	3	9	11	33	35	41	43
0	2	8	10	32	34	40	42

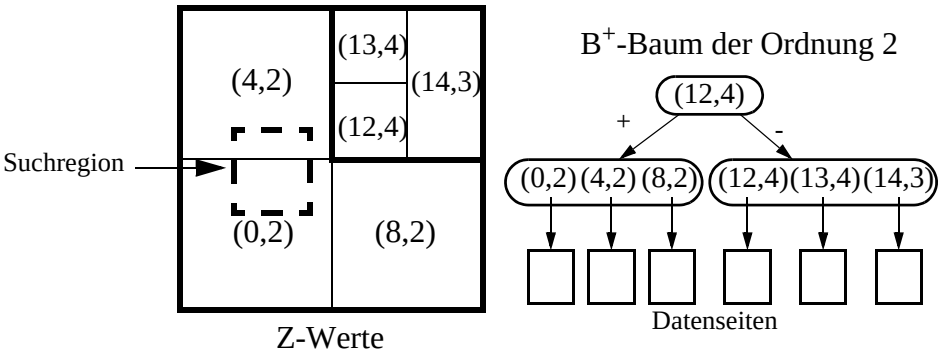
Anfragen

Exact Match Query:

- Bestimme durch Mischen der Binärdarstellung der einzelnen Koordinaten des Anfragepunktes dessen Z-Wert.
- Bestimme wie im herkömmlichen B^+-Baum durch Vergleich dieses Z-Wertes mit den Separatoren die Zelle, die den Anfragepunkt enthält.
- Durchsuche die zugehörige Datenseite nach dem Anfragepunkt.

Range Queries:

- Die Wurzel des B^+-Baumes repräsentiert den gesamten Datenraum.
- In jedem Knoten des B^+-Baumes zerlegen die Separatoren den von dem Knoten repräsentierten Datenraum in disjunkte Teilbereiche.
- Diese Teilbereiche sind aufgrund der Z-Ordnung weitgehend räumlich zusammenhängend; daher werden in der Regel nur einige dieser Bereiche von der Suchregion geschnitten.
- Die Range Query läuft damit nur Teilbäume hinab, deren Bereich von der Suchregion geschnitten wird.



Partitionierungsstrategien:

1.) Partitionierung gemäß einer Splitachse

2.) Partitionierung gemäß Quadtree

PR-Quadtree [Sam 90]

- Quadtrees partitionieren den Datenraum, indem sie ihn rekursiv in vier ***Quadranten (Zellen)*** aufteilen.

  Die relative Lage der Quadranten kann über zwei Bits beschrieben werden.

  Übliche Bezeichnungsweisen: NW, NE, SW und SE.

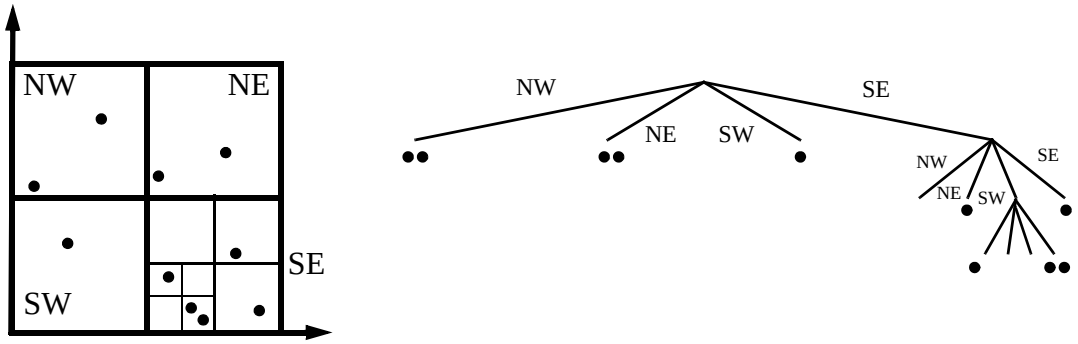
- Die Partitionierung terminiert, falls ein spezifisches ***Abbruchkriterium*** erfüllt ist.

  Beispiele für Abbruchkriterien:

  - die max. Auflösung ist erreicht.
  - eine max. Zahl von Punkten pro Zelle ist unterschritten.

- Die Partitionierung kann durch einen Baum repräsentiert werden, dessen innere Knoten einen Grad von 4 haben.

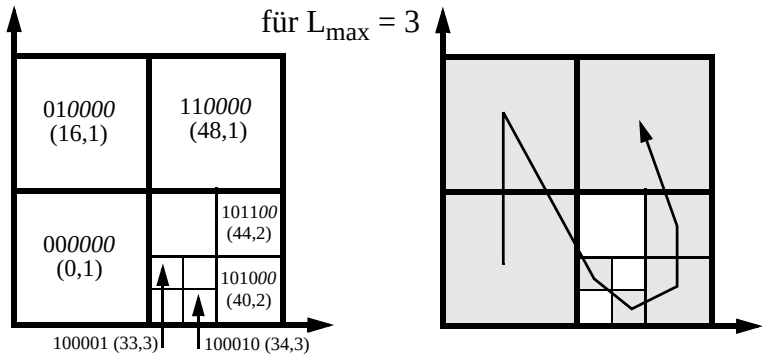
Beispiel:



Partitionierung des PR-Quadtrees
Abbruchkriterium: max. 2 Punkte pro Zelle

Die nicht-leeren Zellen des Quadtrees können durch Z-Werte beschrieben und in einem B^+-Baum gespeichert werden. Da nur Binärregionen mit gerader Länge auftreten können, kann das Level der Z-Werte und das max. Level halbiert werden.

Beispiel:



Literatur:

Eine ausführliche Darstellung über Quadtrees findet man in

[Sam 90] Samet H.: *'The Design and Analysis of Spatial Data Structures'*, Addison Wesley, 1990.

Quellenhinweis:

[See 91] Seeger B.: *'Multidimensional Access Methods and their Applications'*, Tutorial, 1991.

6.3 R-Bäume

R-Baum

Der *R-Baum* [Gut 84] ist ein balancierter Baum, der ursprünglich zur Speicherung von Rechteckdaten entworfen wurde.

Da ein multidimensionaler Punkt ein Spezialfall eines Rechteckes ist (Rechteck mit der Fläche Null), eignet sich der R-Baum auch für die Verwaltung von multidimensionalen Punktdaten.

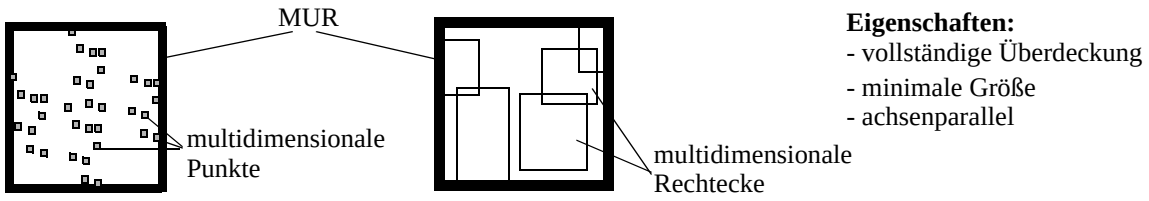
Idee

- basiert auf der Technik überlappender Seitenregionen.
- verallgemeinert die Idee des B+-Baums auf den 2-dimensionalen Raum

Aufbau einer Seite

- Eine Seite besteht aus einer Menge von Einträgen.
- Jeder Eintrag in einer Directory-Seite besteht aus einem *Minimal Umgebenden Rechteck* (MUR) und einem Verweis auf eine Seite.

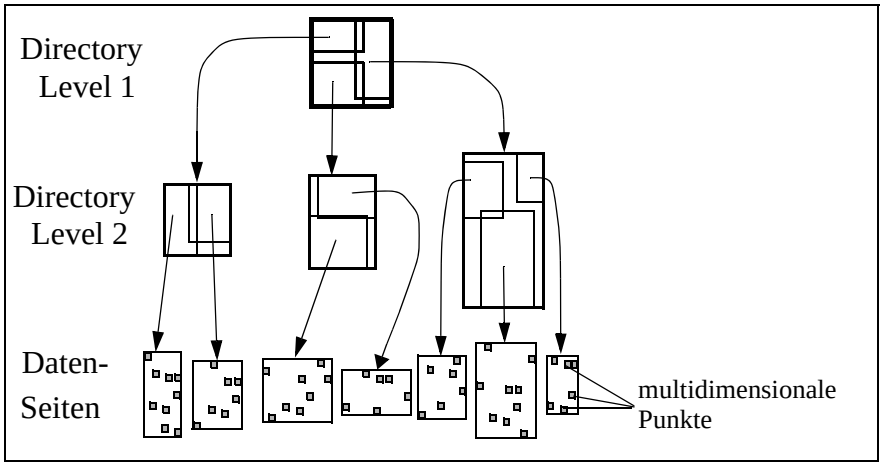
  Minimal Umgebendes Rechteck (*MUR*) = kleinstes achsenparalleles Rechteck, das eine Menge von Punkten bzw. Rechtecken vollständig umfasst (konservative Approximation)



- Ein Eintrag in einer Datenseite besteht aus einem Punkt (bzw. MUR) und evtl. einem Verweis auf die vollständige Objektbeschreibung (exakte Objekt-Repräsentation).

Partitionierung des Datenraums

- Jedes Rechteck in einer Directoryseite umfasst als MUR alle Rechtecke (bzw. Punkte) in allen Directory- oder Datenseiten, die im zugehörigen Teilbaum liegen.
- Die Rechtecke einer Seite können sich überlappen.
- Die Partitionierung des Datenraumes der Directoryseite muss nicht vollständig sein.



Eigenschaften

Sei M die maximale Zahl von Einträgen pro Seite und m ein Parameter mit $2 \leq m \leq M$. Es gilt:

- Jeder Knoten des R-Baumes außer der Wurzel hat zwischen m und M Einträge.
- Die Wurzel hat mindestens zwei Einträge, außer sie ist ein Blatt.
- Ein innerer Knoten mit k Einträgen hat genau k Söhne.
- Der R-Baum ist balanciert.

Ist N die Anzahl der gespeicherten Datensätze, so gilt für die Höhe h des R-Baumes:

$$h \leq \lceil \log_m N \rceil + 1$$

Point Query

Wir rufen folgenden Algorithmus mit der Wurzel des R-Baumes und dem Anfragepunkt P auf:

```
PointQuery (Page, Point);
    FOR ALL Entry ∈ Page DO
        IF Point IN Entry.Rectangle THEN
            IF Page = DataPage THEN
                Write (Entry)
            ELSE
                PointQuery (Entry.Subtree^, Point);
```

Window Query

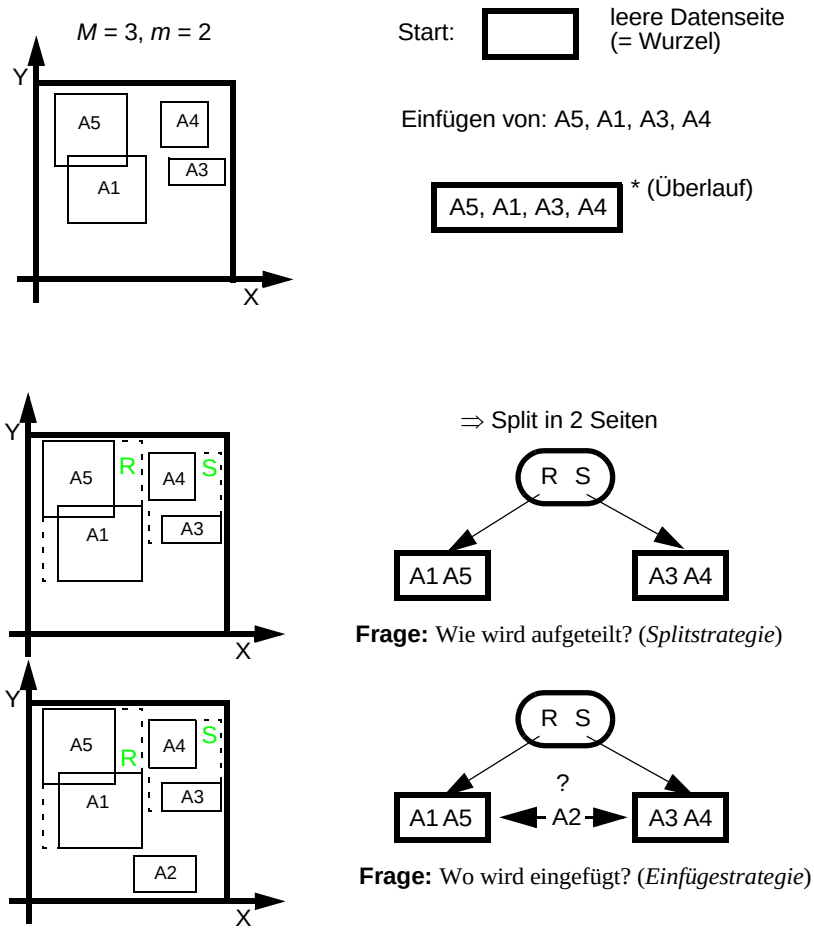
Wir rufen folgenden Algorithmus mit der Wurzel des R-Baumes und dem Window W auf:

```
Window Query (Page, Window);
    FOR ALL Entry ∈ Page DO
        IF Window INTERSECTS Entry.Rectangle THEN
            IF Page = DataPage THEN
                Write (Entry)
            ELSE
                WindowQuery (Entry.Subtree^, Window);
```

- Gibt es eine Überlappung der Directory-Rechtecke im Bereich der Anfrage, verzweigt die Suche in mehrere Pfade. Dies gilt sowohl für die Point-Query als auch für die Window-Query.

Optimierungsziele

- geringe Überlappung der Seitenregionen
- Seitenregionen mit geringem Flächeninhalt
  $\Rightarrow$ geringe Überdeckung von totem Raum
- Seitenregionen mit geringem Umfang



Einfügen eines Punktes P (bzw. Rechteckes R)

Beim Durchlauf durch den Baum können drei Fälle eintreten:

1. *P (bzw. R) fällt vollständig in genau ein Directory-Rechteck D*

   Wir folgen dem Verweis von D.

2. *P (bzw. R) fällt vollständig in mehrere Directory-Rechtecke* D_1 , ... , D_n

   Wir folgen dem Verweis des D_i mit der geringsten Fläche.

3. *P (bzw. R) fällt in kein Directory-Rechteck vollständig*

   Wir vergrößern das Directory-Rechteck D, welches dadurch den geringsten Flächenzuwachs erfährt (falls mehrere solche Rechtecke existieren, wähle davon das mit der kleinsten Fläche), und folgen dem Verweis von D.

Split von Seiten

- Durch das Einfügen von Datensätzen in Datenseiten bzw. durch Split von Sohnseiten kann eine Seite überlaufen.
- *Frage:*

  Wie teilen wir eine Menge von Punkten bzw. Rechtecken in zwei Mengen auf ?

  - Eine optimale Aufteilung ist zu aufwendig zu berechnen, da es 2^n verschiedene Arten gibt, n Punkte oder n Rechtecke in zwei Mengen aufzuteilen.
  - Was ist ein geeignetes Kriterium, um eine Aufteilung zu bewerten ?

  $\Rightarrow$ Wir benötigen *Heuristiken*, die die Aufteilung vornehmen, $\rightarrow$ R^*-Baum.

R*-Baum

Der R*-Baum [BKSS 90] ist eine Variante des R-Baumes. Bei seinem Entwurf wurden folgende Entwurfskriterien zugrunde gelegt:

- Die *Fläche* von Directory-Rechtecken, die nicht von den enthaltenen Rechtecken überdeckt wird ("toter Raum"), soll minimiert werden. So schneiden Query-Windows möglichst wenige Directory-Rechtecke, und möglichst große Teilbäume können früh von der weiteren Suche ausgeschlossen werden.
- Die *Überlappung* der Directory-Rechtecke soll minimiert werden. Dadurch wird die Zahl der zu verfolgenden Pfade (speziell bei Point-Queries) minimiert.
- Der *Umfang* eines Directory-Rechteckes soll minimiert werden. Dieses Kriterium ist gut bei der Annahme quadratischer Query-Windows, d. h. solcher mit minimalem Umfang.

Offensichtlich konkurrieren diese Kriterien miteinander: z. B. benötigt man, um die Überlappung zu minimieren, mehr Freiheit in der Form der Directory-Rechtecke, wodurch der Umfang der Directoy-Rechtecke wachsen kann.

Einfügen

Das Einfügen läuft wie beim normalen R-Baum ab, nur dass jetzt im 3. Fall wie folgt unterschieden wird:

3. P (bzw. R) fällt in kein Directory-Rechteck vollständig

3. a) Die Directoryseite verweist auf Datenseiten

Wir wählen das Rechteck, dessen Vergrößerung den kleinsten Zuwachs an Überlappung bringt. Weitere Kriterien in Zweifelsfällen: Flächenzuwachs und Größe der Fläche.

3. b) Die Directoryseite verweist auf Directoryseiten

Wir wählen das Rechteck, dessen Vergrößerung den kleinsten Flächenzuwachs bringt. Weiteres Kriterium in Zweifelsfällen: Größe der Fläche.

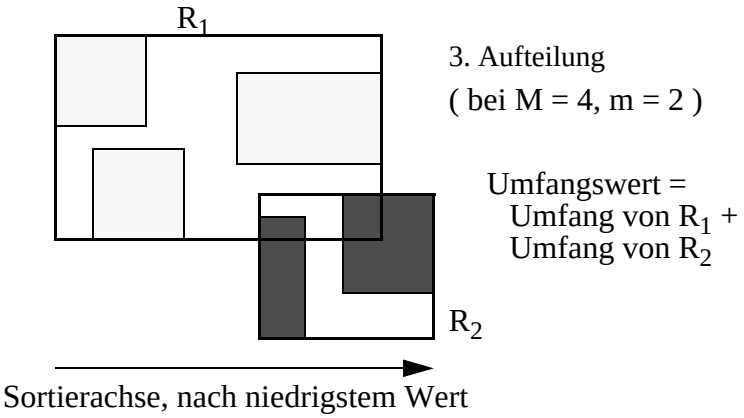
Split von Seiten

Die *Splitheuristik* sieht wie folgt aus:

1. Bestimmung der Splitachse

- Entlang jeder Achse werden die Rechtecke (bzw. Punkte) gemäß ihrem niedrigsten und ihrem höchsten Wert sortiert.
- Für jede der beiden Sortierungen werden M-2m+2 Aufteilungen der M+1 Rechtecke bestimmt, so dass die erste Gruppe der j-ten Aufteilung m-1+j Rechtecke und die zweite die übrigen Rechtecke enthält.
- Der *Umfangswert* sei die Summe aus dem Umfang der beiden Rechtecke R_1 und R_2, die die Rechtecke beider Gruppen umfassen.
- Es wird für jede Achse die Summe der Umfangswerte aller zugehörigen Aufteilungen bestimmt.

- Es wird die Achse gewählt, die die geringste Umfangssumme besitzt.



2. Wahl der Aufteilung

- Es wird die Aufteilung der Splitachse genommen, bei der R_1 und R_2 die geringste Überlappung haben.
- In Zweifelsfällen wird die Aufteilung genommen, bei der R_1 und R_2 die geringste Überdekkung von "totem Raum" besitzen.

Die besten Resultate hat bei Experimenten m = 40 % von M ergeben.

Reorganisation

Die Partitionierung des R-Baumes wird stark von der Einfügereihenfolge geprägt, da das Directory bei Splits nur sehr lokal verändert wird. Insbesondere die als erstes eingefügten Punkte (bzw. Rechtecke) prägen die Partitionierung des R-Baumes.

Idee: Reorganisation des Baumes durch Löschen und Wiedereinfügen von Punkten.

Umsetzung:

Der R^*-Baum ruft dazu vor der Durchführung eines Splits den *ReInsert-Algorithmus* auf:

- Die Distanz der Punkte (im Fall von Rechteck-Daten die Mittelpunkte der Rechtecke) zum Mittelpunkt des umgebenden Rechteckes wird bestimmt.
- Die p Punkte (Rechtecke) mit dem größten Abstand werden gelöscht und das umgebende Rechteck entsprechend angepasst.
- Die gelöschten Punkte (Rechtecke) werden wieder eingefügt.

- Durch das ReInsert kann unter Umständen ein Split vermieden werden.
- Das ReInsert kann sowohl für Daten- als auch für Directory-Rechtecke eingesetzt werden.
- Die besten Resultate hat bei Experimenten p = 30 % von M ergeben.

Experimentelles Leistungsverhalten [BKSS 90]

- Der R^*-Baum zeigt ein wesentlich besseres Leistungsverhalten als der normale R-Baum:
  (für Rechteckdaten)
  - Anfragen haben 10 bis 75 % Prozent weniger Seitenzugriffe.
  - Die Speicherplatzausnutzung ist erheblich besser, sie liegt bei etwa 71 bis 76 %.
  - Selbst die Kosten für das Einfügen sind trotz des ReInsert fast immer unter denen des R-Baumes.
- Der R^*-Baum zeigt auch als Punktzugriffsstruktur ein ausgezeichnetes Leistungsverhalten.

The R-tree is based on the heuristic optimization of the area of directory rectangles.

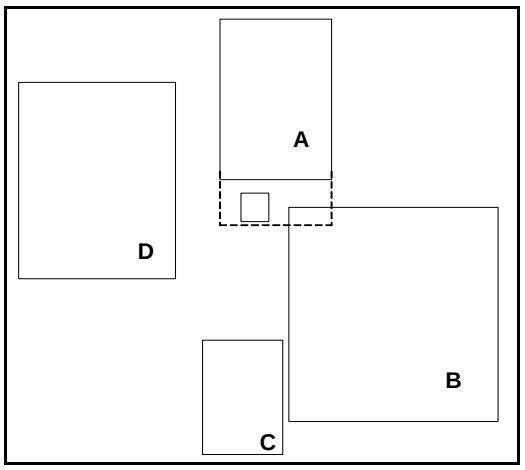
Our Engineering-Type Approach

using a standardized testbed for performance comparison of access structures
find a best possible combination of optimization criteria such as area, overlap, margin, storage utilization, shape etc.

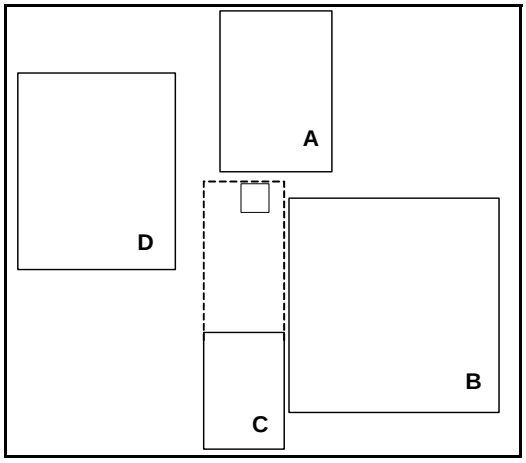
ChooseSubtree Algorithm:

R-tree

For accommodating a new data rectangle, the R-tree minimizes the area increase.



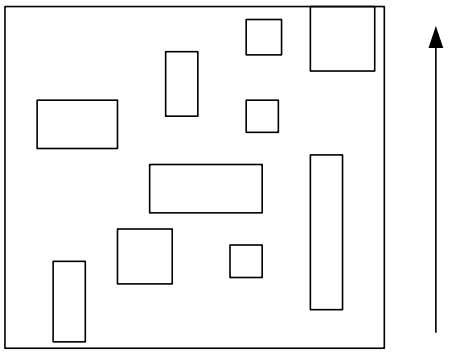
The R*-tree minimizes the overlap increase of the directory rectangles pointing to data rectangles
(on the lowest level of the directory).



R*-tree Split:

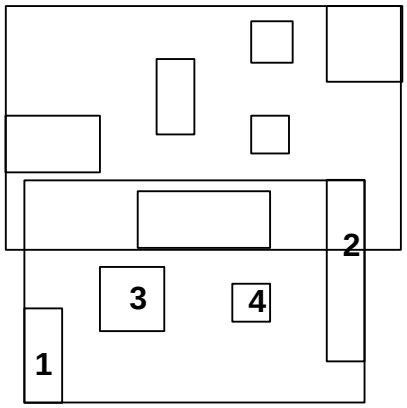
Determine split axis:

- For each axis, the entries are first sorted by the low values, then sorted by the high values of their rectangles.
- For each sort, all possible distributions are generated:

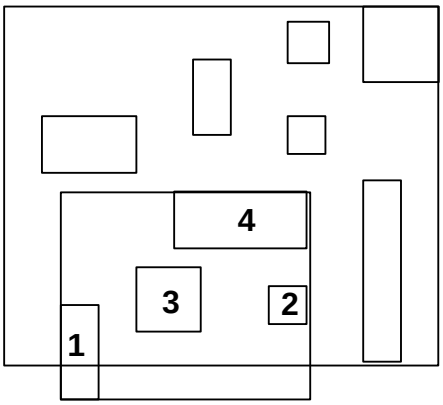


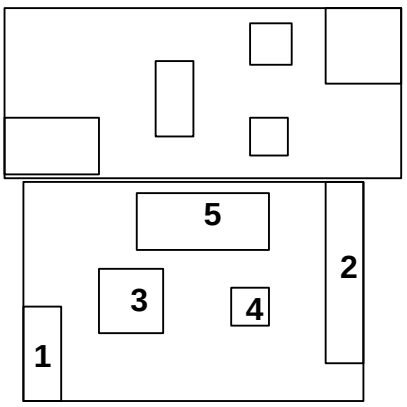
sorted by low values

sorted by high values

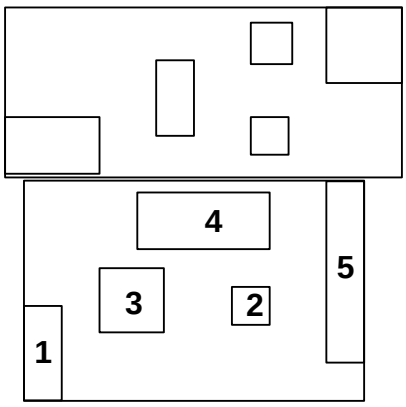


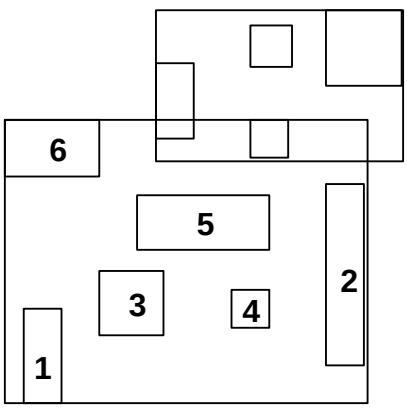
all possible distributions for the vertical axis

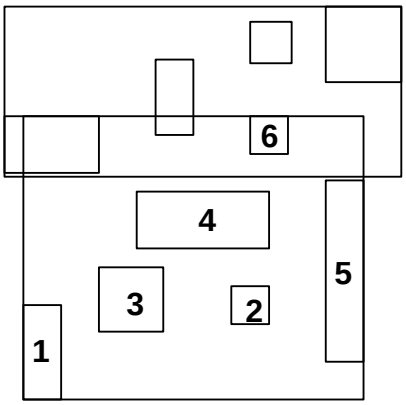




minimal fill degree = 40%







- For each axis, the sum of all margins of the different distributions is computed.

  Margin of a rectangle = sum of the lengths of its edges
- The axis with the minimum sum of its margins is the split axis

Performing Split:

- The split is performed according to the distribution which yields the minimum overlap (minimum area of dead space, if overlap = 0)
- Best minimum fill degree m = 40%
- CPU cost of the split is O(n log n), where n = number of entries of a node (page)

Fact:

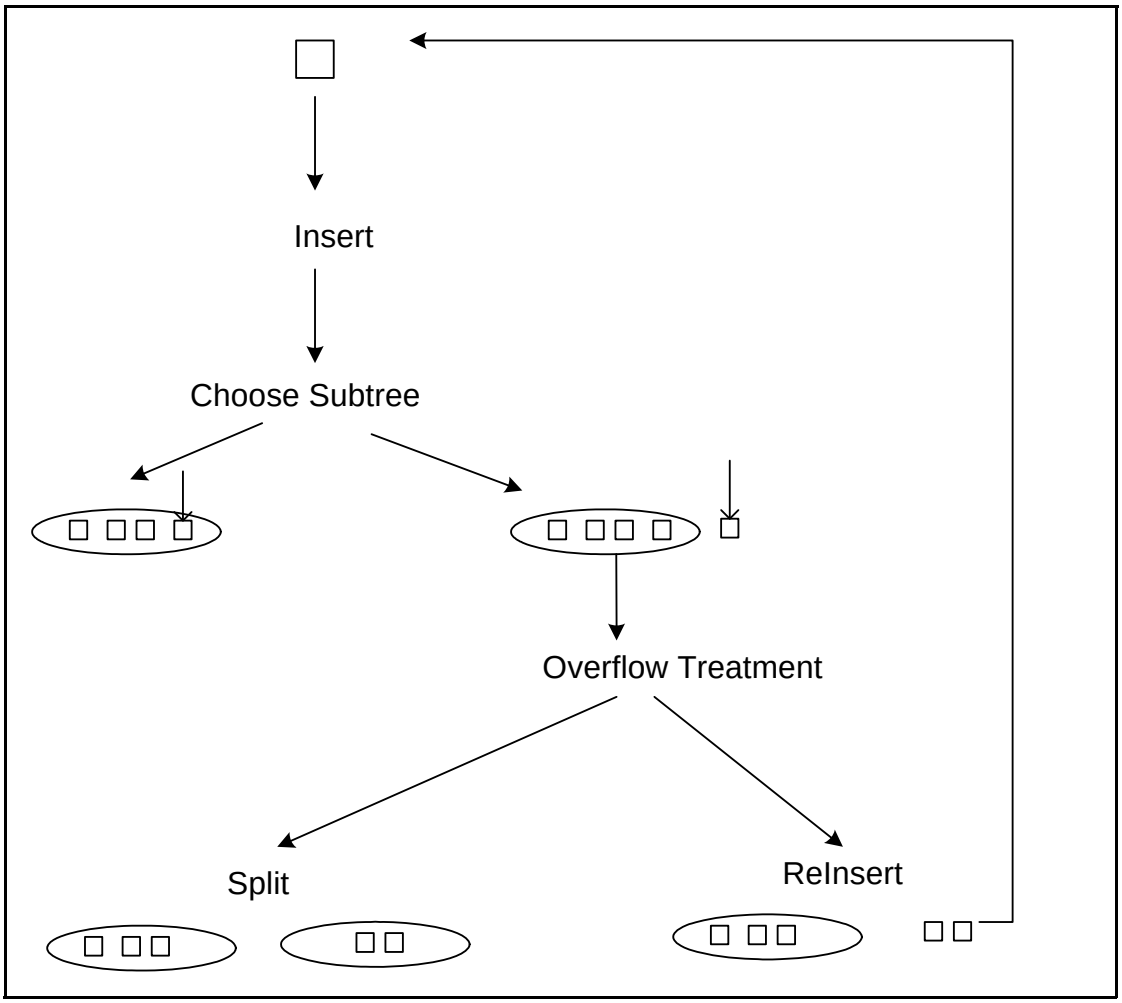
The retrieval performance of the R-tree may suffer from data rectangles inserted during the early growth of the tree.

Improvement:

The retrieval performance of the R-tree can be considerably improved by simply deleting early inserted data and reinserting it again.

Insertion Algorithm for the R*-tree:

For dynamic reorganization, the R*-tree forces entries to be reinserted during the insertion routine.

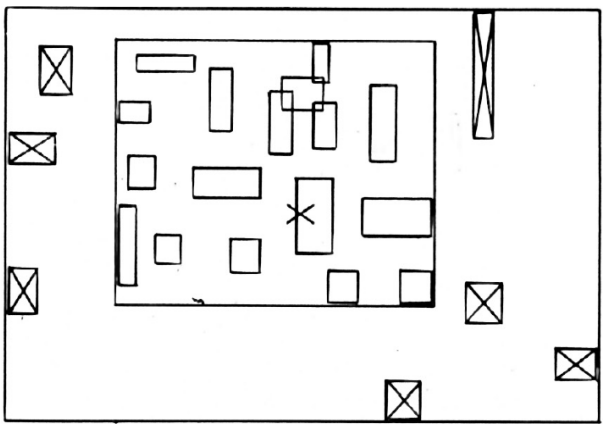


Algorithm ReInsert:

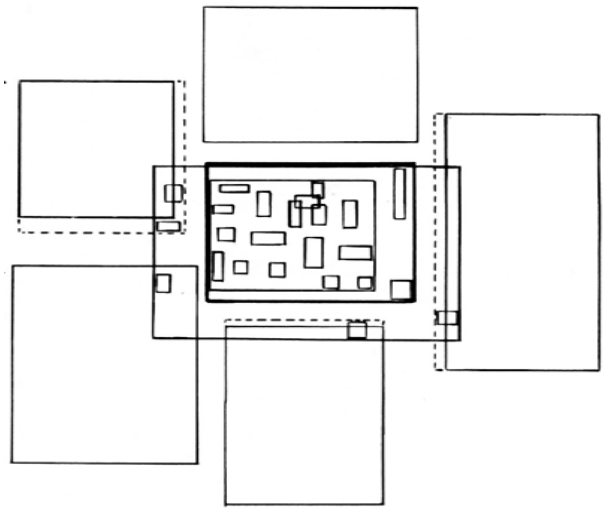
(i) For all M+1 entries of the overfilled node N compute the distances between the centers of their rectangles and the center of the bounding box of N.

(ii) Sort the entries in decreasing order of their distances.

(iii) Remove the first p entries from N and adjust the bounding box of N. (p = 30% yields best performance in experiments)



(iv) Reinsert the p removed entries, beginning with the entry closest to the adjusted node, invoking the insertion algorithm.



Due to ReInsert, splits are often prevented:

- Forced ReInsert moves entries to neighboring nodes ⇒ decreases overlap
- Storage utilization is improved
- The outer rectangles of a node are reinserted
  ⇒ the shape of directory rectangles will be more quadratic
  ⇒ improves packing in the next higher level
  ⇒ improves storage utilization

With forced ReInsert in the R*-tree, the average number of disc accesses for insertions increases only 4% and is the lowest of all R-tree variants.

Experimental Evaluation:

Unweighted average over all 7 distributions (data files)

	query average	spatial join	storage	insert
lin. Gut	227.5	261.2	62.7	12.63
qua. Gut	130.0	147.3	68.1	7.76
Greene	142.3	171.3	69.7	7.67
R*-tree	100.0	100.0	73.0	6.13

Highlights:

- R*-tree is the most robust method, i.e. there is no experiment where the R*-tree does not have the best performance
- The average performance gain of the R*-tree for spatial join is higher than for other queries
- The R*-tree has the best storage utilization
- Even with *Forced ReInsert* the average insertion cost of the R*-tree is lower than for the other R-tree variants

Good spatial access methods should handle both spatial objects and point objects efficiently (geometry and non-geometry information)

We ran the R*-tree with our benchmark for point access methods (Santa Barbara 89).
GRID = 2-level grid file (Hinrichs 85)

	query average	storage utiliz.	insert
lin. Gut	233.1	64.1	7.34
qua. Gut	175.9	67.8	4.51
Greene	237.8	69.0	5.20
GRID	127.6	58.3	2.56
R*-tree	100.0	70.9	3.36

- The performance gain of the R*-tree over the R-tree variants is even higher for points than for rectangles.
- The 2-level grid file is better than the R*-tree in average insertion cost

Summary:

- The R*-tree outperforms the other R-tree variants for rectangle and point data in all experiments
- R*-tree is robust against ugly data distributions
- Best storage utilization
- Dynamic reorganization using *Forced ReInsert*
- Cost of the implementation of R*-trees is not much higher than for other R-trees

6.4 Distanz-Basierte Indexstrukturen: M-Tree

Probleme bei herkömlichen Indexstrukturen:

- Objekte nicht immer als multidimensionaler Punkt darstellbar

  z.B. komplexstrukturierte Objekte wie Graphen, Bäume etc.
- Die Ähnlichkeit der Objekte entspricht nicht der Nähe der Objekte im Objektraum

  z.B. Objekte in Netzwerkgraphen (Ähnlichkeit zweier Objekte entspricht dem kürzesten Netzwerkpfad zwischen den Objekten)

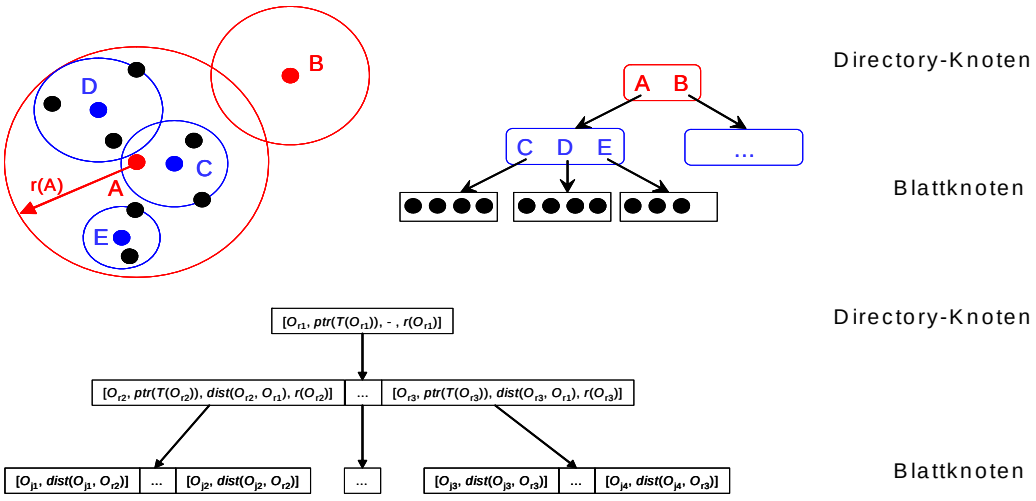
Lösung:

- Indexierung der Objekte über Referenzobjekte

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-/Directory-Seiten
  - 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



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.

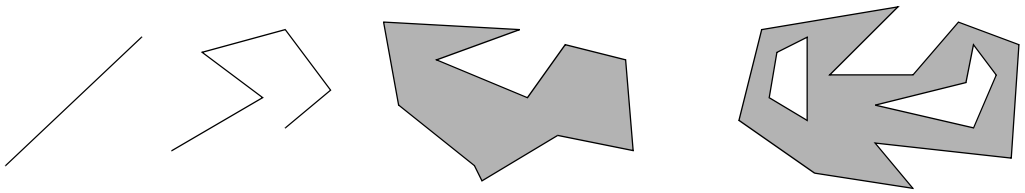
7.1 Grundlagen

- ***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.



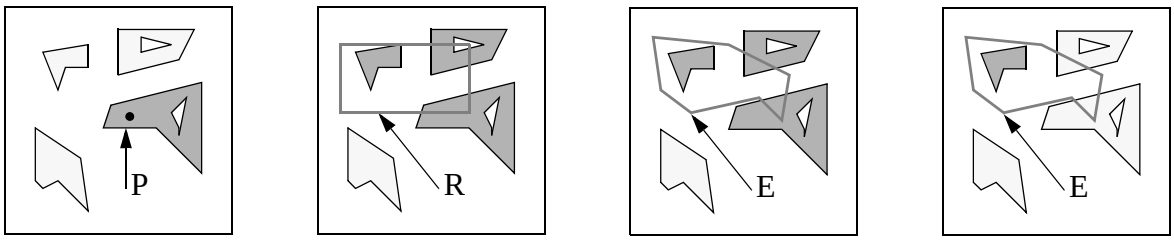
Linie Linienzug Fläche (einfaches Polygon) Fläche mit Löchern (EPL)

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:

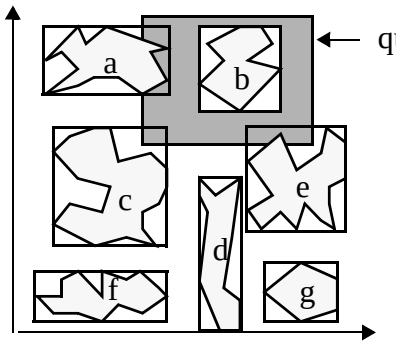
1. Filterschritt

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.

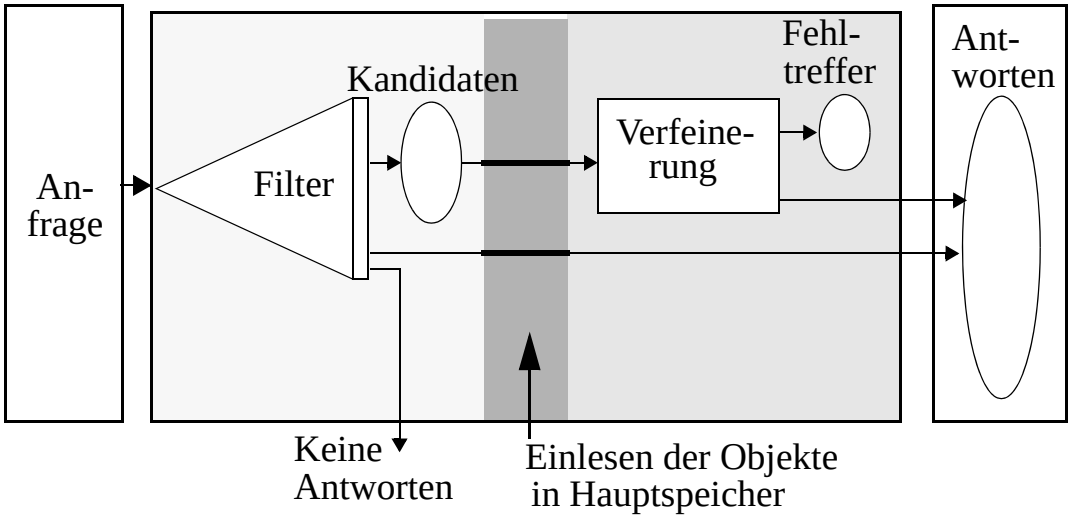
2. Verfeinerungsschritt

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.

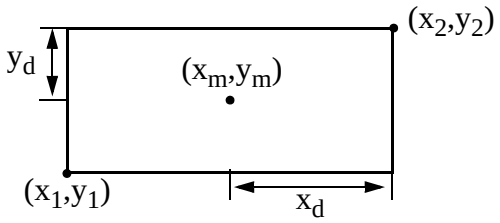
Punkttransformation

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*

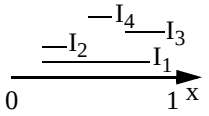
  Das Rechteck wird durch den Mittelpunkt (x_m,y_m) und die jeweilige halbe Ausdehnung x_d und y_d beschrieben.
- *Eckentransformation*

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

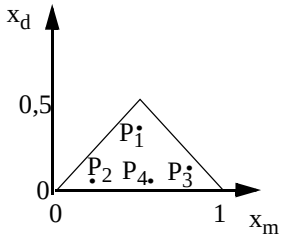


Transformation des Datenraumes und der Anfragebereiche (für 1-dimensionale Intervalle):

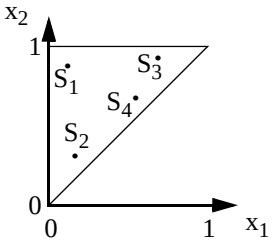
Transformation von Intervallen I_j



in 2-dim. Punkte:

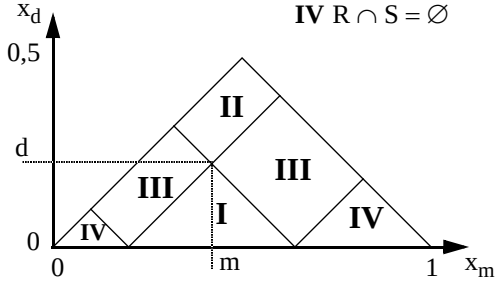


$P_j = (x_m,x_d)$
(Mittentransformation)



$S_j = (x_1,x_2)$
(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.

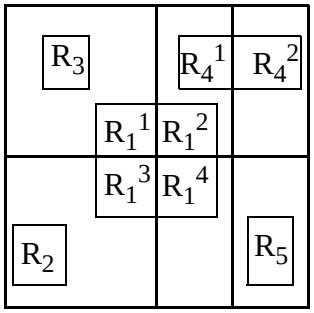
  ⇒Raumbezogene Anfragen verlieren dadurch deutlich an Effizienz.
- Die Punkttransformation ist auf jede Punktzugriffsstruktur anwendbar. So müssen z.B. die Einfüge- oder Lösch-Operationen nicht geändert werden.

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

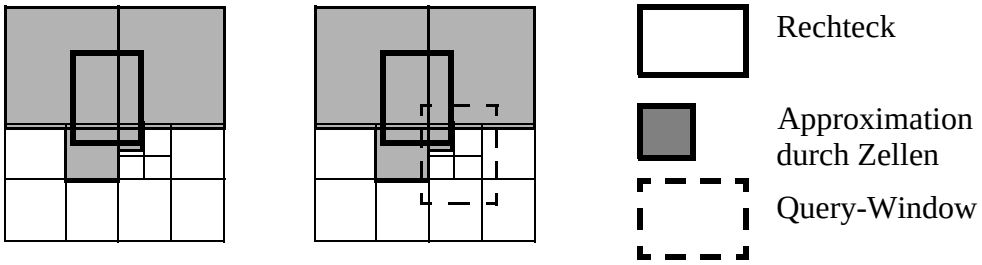
Eigenschaften:

- 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.

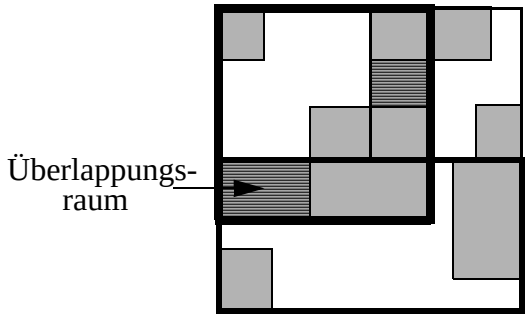
Alle diese Zellen erhalten einen Eintrag für das Rechteck.



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.

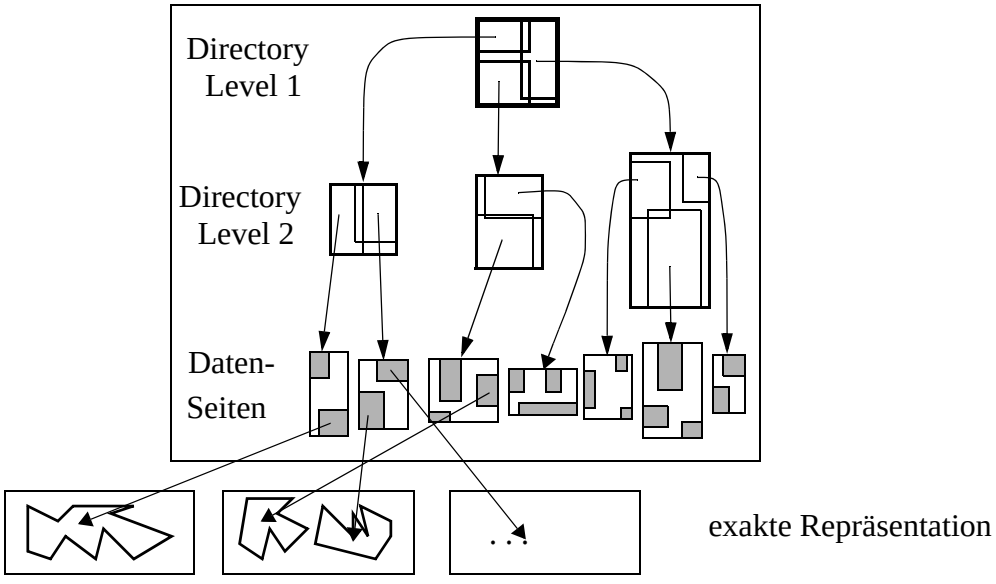
- Fällt eine Anfrage in einen Überlappungsraum, müssen mehrere Pfade im Suchbaum untersucht werden.

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

Anwendung

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

Beispiel:



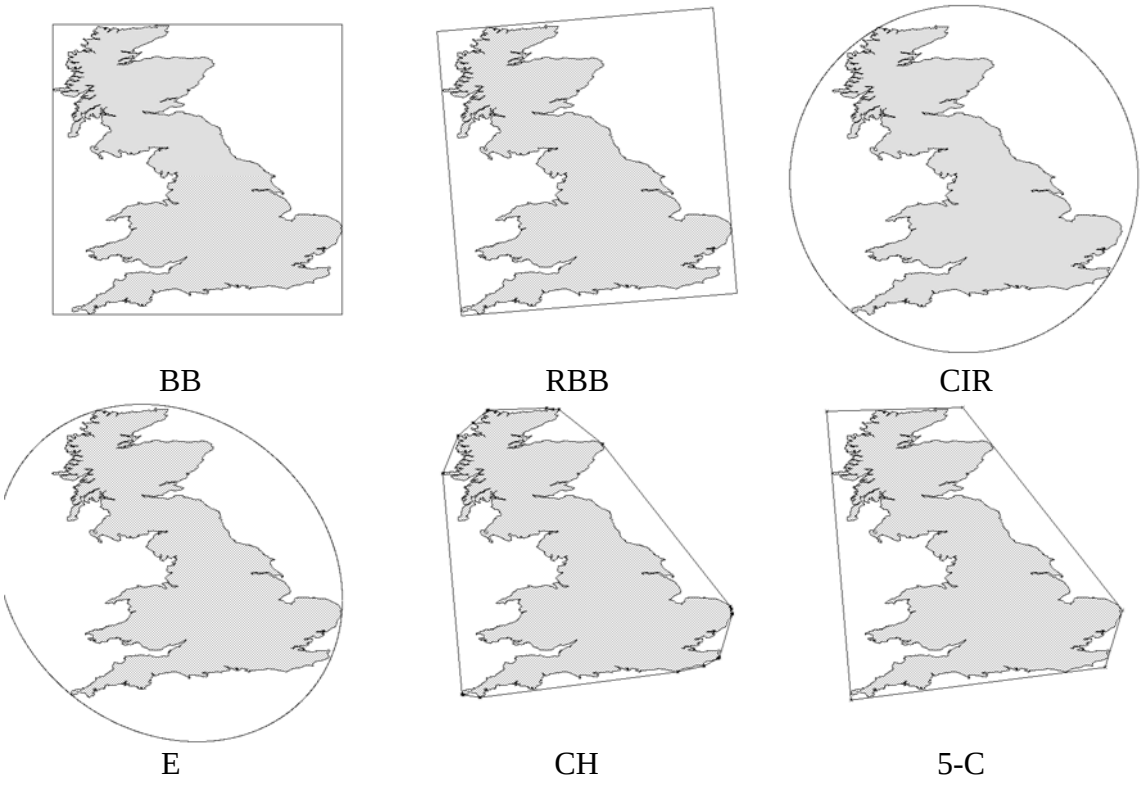
7.3 Approximation

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)*



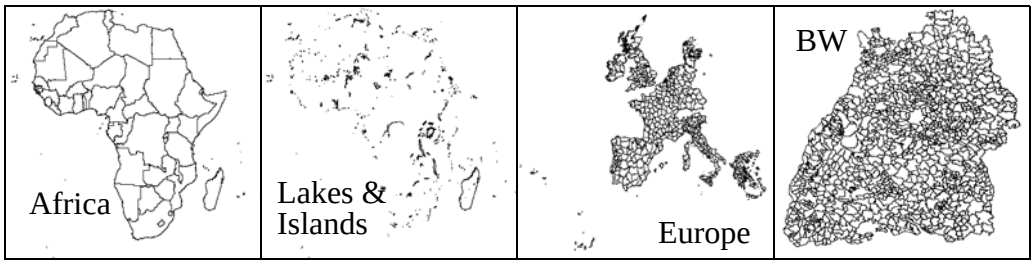
Approximationsgüte

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

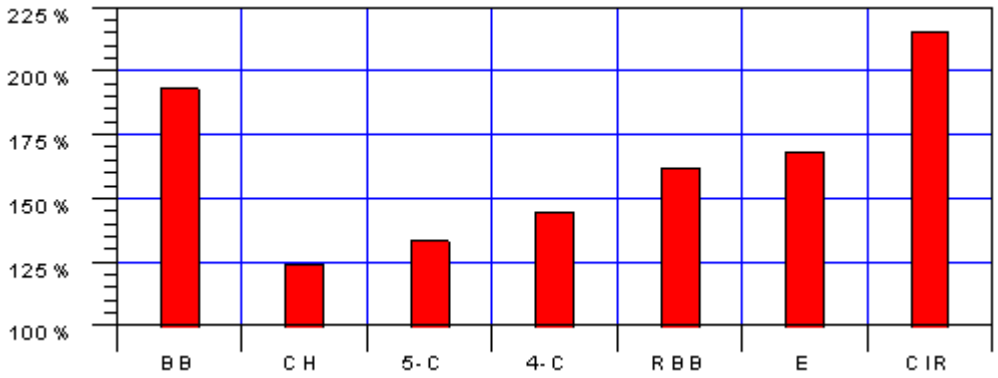
- A: Fläche, O: Objekt, Appr[O]: Approximation von O
- Q_{Appr} = 100% wenn Approximation optimal, d. h. Appr[O] = O
- je kleiner Q_{Appr}, umso besser die Approximation

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

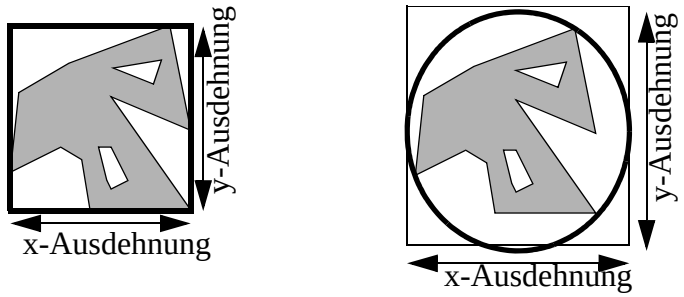
- höherer Speicherplatzbedarf auf Datenseiten, z. B. bei Annahme von 4 Bytes pro Parameter:

	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

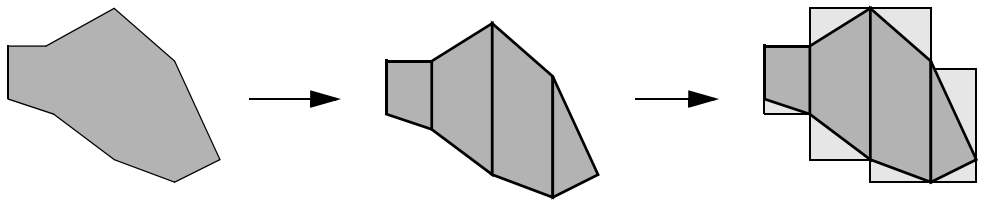
Zusammenfassende Bewertung

- 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

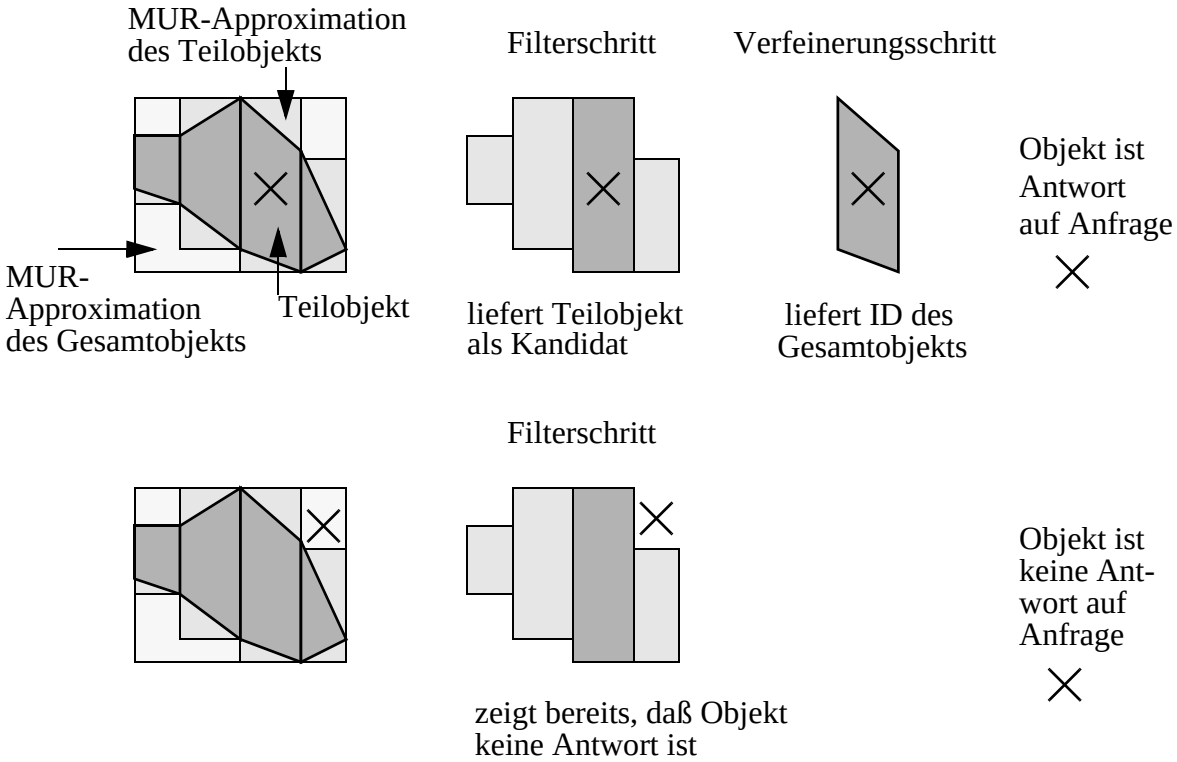
- Ziel: Tests auf der exakten Repräsentation der Objekte (im Verfeinerungsschritt) vereinfachen.
- Beobachtung: für das Ergebnis eines Tests ist i. A. nur ein kleiner Teil des Objekts relevant.
- Vorgehen: Dekomposition der EPL in einfache Teilobjekte und Approximation der Teilobjekte durch minimal umgebende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



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

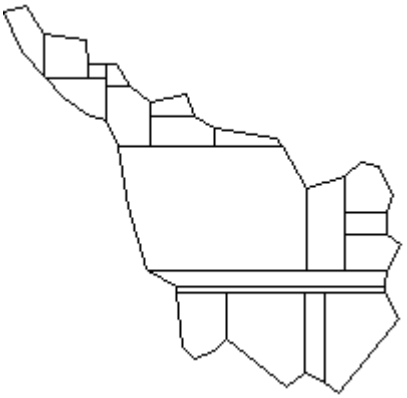
Anfragebearbeitung mit Dekomposition (Point Query)

- 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

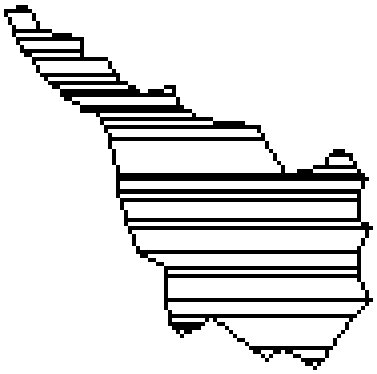
(n = Zahl der Eckpunkte des EPL)



konvexe Dekomposition

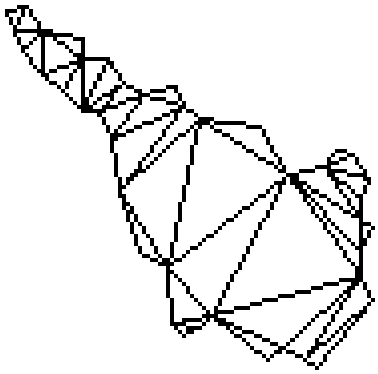
in konvexe Polygone

≈ *n*/2 Komponenten



Dekomposition in Trapeze

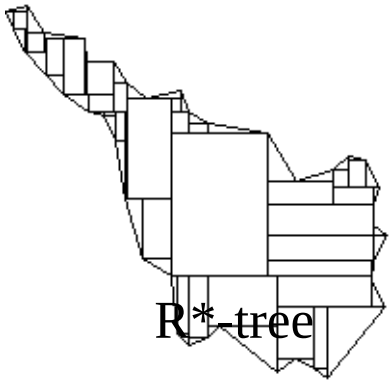
≈ *n* Komponenten



Triangulation

Dekomposition in Dreiecke

≈ *n* Komponenten



Heterogene Dekomposition

in Dreiecke und Rechtecke

≈ 1.8**n* Komponenten

Eigenschaften

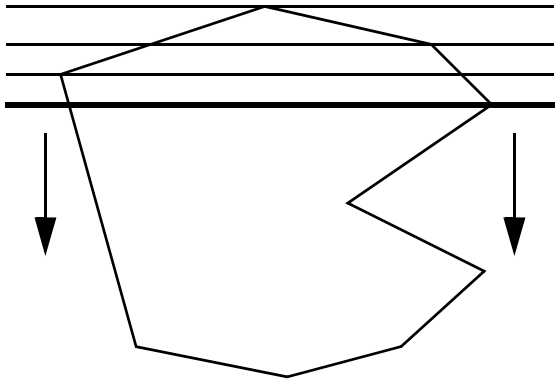
- lineare Anzahl einfacher Komponenten
- Die Typen der Komponenten sind so gewählt, daß sie gut durch MUR approximiert werden können.

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.

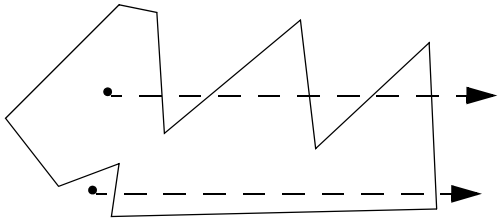
Für jedes Element der sortierten Liste definiere ein Trapez durch das aktuelle und das nächste Listenelement.

⇒ 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

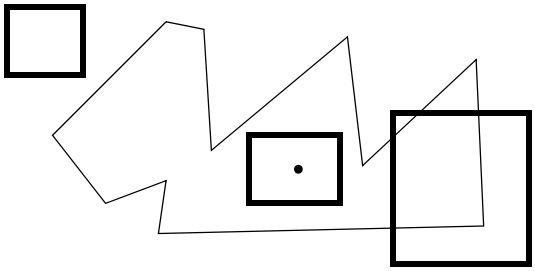


Zähle die Schnittpunkte.

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



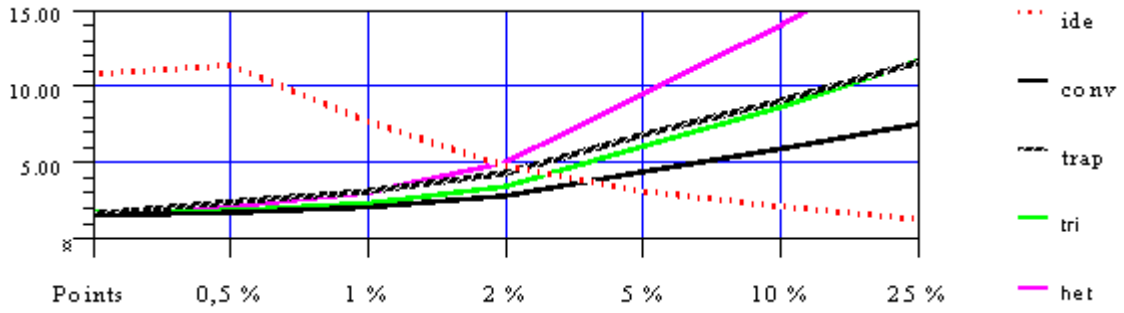
Schneidet eine der Kanten das Window?
Wenn ja: Antwort = ja
Wenn nein:

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.

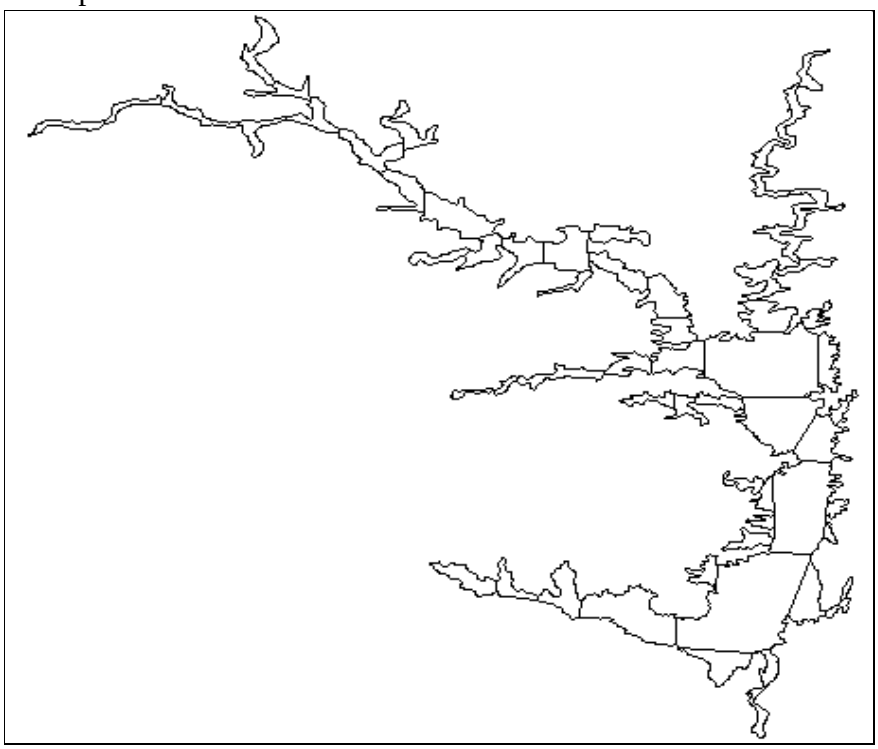
Balancierte Dekomposition [SK 93]

- 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.

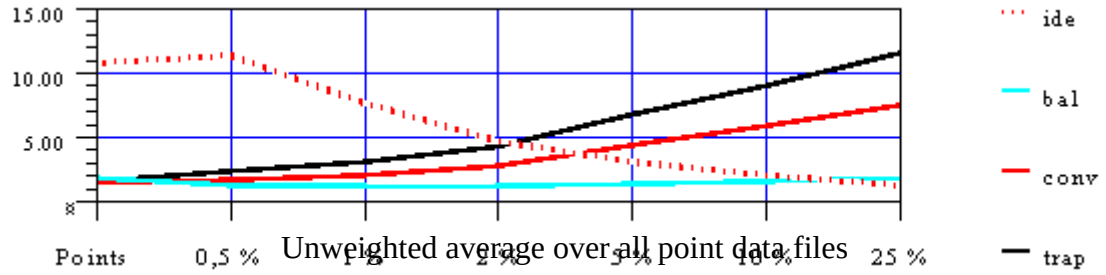
- Beispiel: Volta See



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.