

5. Zugriffsstrukturen für ausgedehnte räumliche Objekte (Forts.)

C) Transformationsmethoden

C1) Transformation in höherdimensionale Punkte

Einfaches geometrisches Objekt \mapsto Punkt in höherdim. Raum

- (i) z.B. Intervall $[x_1, x_2] \mapsto$ 2d-Punkt (x_1, x_2)
(Endpunkttransformation)
- (ii) z.B. Rechteck mit diagonal gegenüberliegenden Endpunkten (x_1, y_1)
und $(x_2, y_2) \mapsto$ 4d-Punkt (x_1, y_1, x_2, y_2) (Endpunkttransformation)
- (iii) z.B. Ellipse mit Mittelpunkt (x, y) und Radien r_x, r_y
 \mapsto 4d-Punkt (x, y, r_x, r_y) (Mittelpunkttransformation)

Komplexe Objekte wären zu approximieren.

Transformation in höherdimensionale Punkte (Forts.)

Vor-/Nachteile:

- + Zugriffsstrukturen für Punktmengen nutzbar
- + Übersetzung von vielen Anfragen einschl. Bereichsanfragen möglich
- Nachbarschaftsstruktur bleibt nicht erhalten
(benachbarte Objekte können zu weit entfernten Punkten werden)
- manche räumliche Prädikate nicht mehr direkt ausdrückbar,
z.B. “ist nächster Nachbar”
- Objektverteilung nach Transformation oft “schief”

Es gibt Vorschläge zur Adaption von Punkt-Zugriffstrukturen an solche Transformationen (z.B. k-d-B-Bäume → *LSD-Bäume*).

Transformation in höherdimensionale Punkte (Forts.)

zur Übersetzung von Anfragen bzgl. Transformation (i):

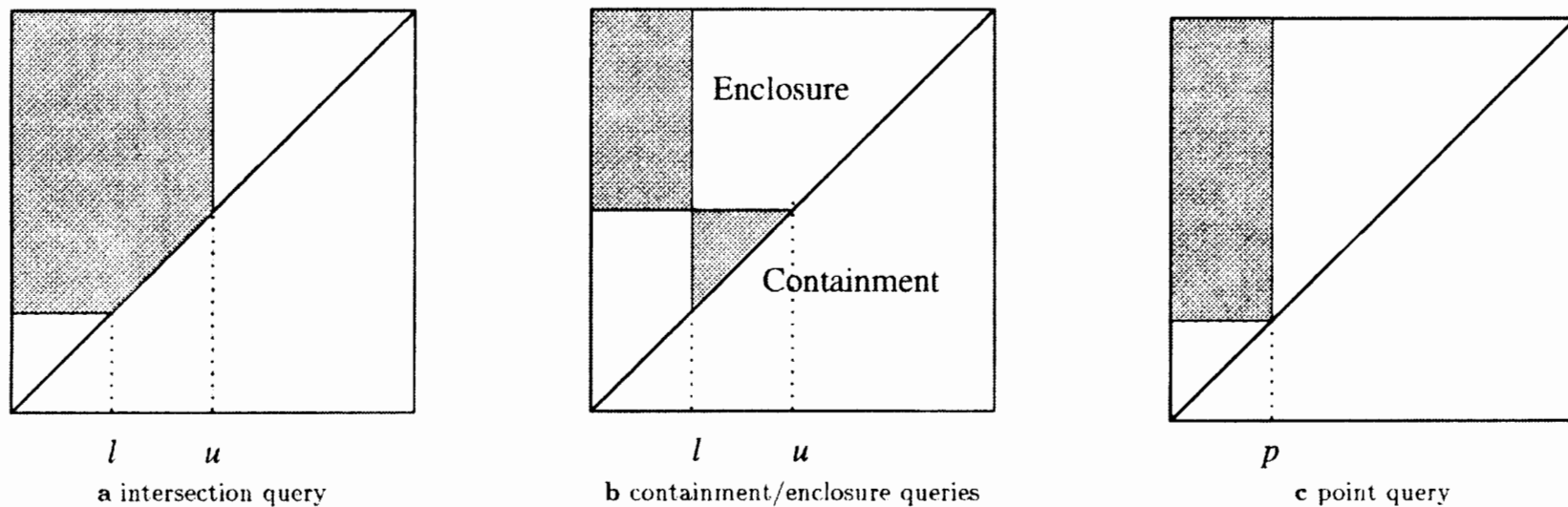


Figure 26. Search queries in dual space—endpoint transformation: (a) intersection query; (b) containment/enclosure queries; (c) point query.

- a) Welche Intervalle schneiden das Intervall $[l, u]$? $= \{ [x, y] \mid x \leq u \wedge y \geq l \}$
- b) Welche Intervalle enthalten das Intervall $[l, u]$ / sind darin enthalten ? (z.Üb.)
- c) Welche Intervalle enthalten den Punkt p ? $= \{ [x, y] \mid x \leq p \wedge y \geq p \}$

C2) Eindimensionale Einbettung

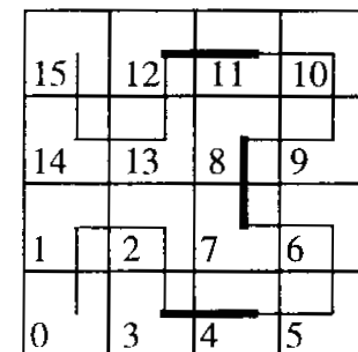
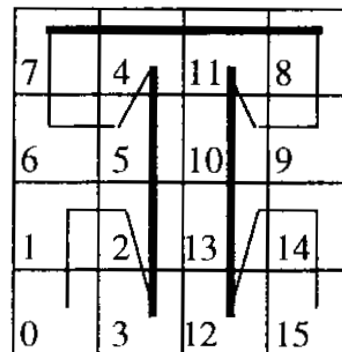
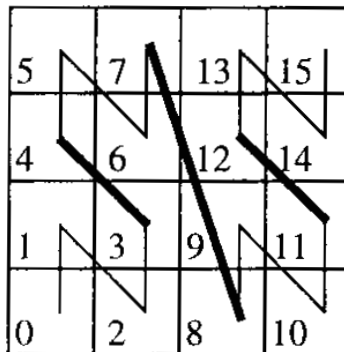
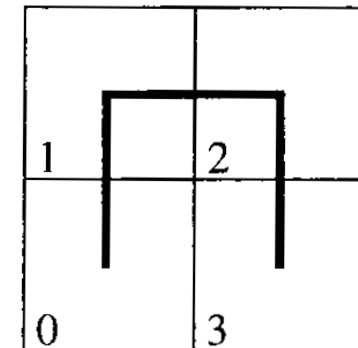
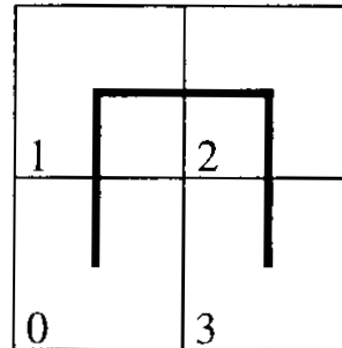
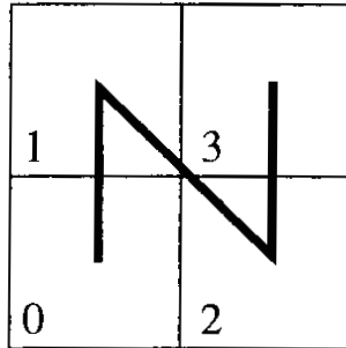
Zunächst Darstellung von Punkten.

Vorbemerkung: Es gibt keine totale Ordnung, die Nachbarschaft (räumliche Nähe) bewahrt.

Ansatz:

- Partitioniere Datenraum durch gleichförmiges Gitter, in mehreren Verfeinerungsstufen.
- Nummeriere Zellen fortlaufend so, dass räumliche Nähe möglichst bewahrt wird, mindestens innerhalb von Zellen einer gröberen Stufe.
- Nutze Zellennummern zur eindimensionalen Indexierung, etwa mittels B*-Baum.

Eindimensionale Einbettung (Forts.)



a) z-Ordnung

b) Gray-Code

c) Hilbert's Kurve

Abb. 9.8: Wichtige eindimensionale Einbettungen

Z-Ordnung = Z-Codes

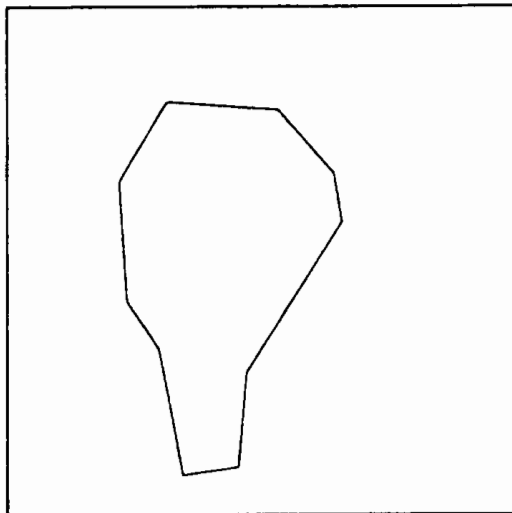
Nummerierung:

- Gegeben sei die Zelle mit Nummer p in Binärdarstellung auf Gitterstufe $i - 1$. Dann lauten die vier Zellennummern auf Stufe i in “Z-Reihenfolge”: $p00, p01, p10, p11$.
- Z-Code der einzigen Zelle auf Stufe 0 ist ε (das leere Wort).
- Also haben die Teilzellen den gleichen Präfix der Länge $2(i-1)$ wie die Oberzellen, nämlich p .
- Spezielle Eigenschaft: ungerades Bit 0/1 entspricht links/rechts, gerades Bit entspricht unten/oben.

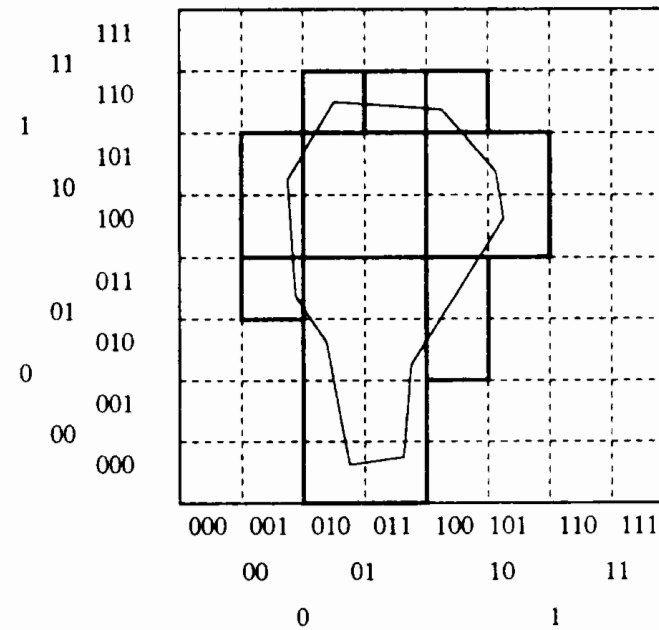
Umrechnungen: (Datenraum sei auf $[0, 1)^2$ normiert.)

- Zellennummer q der Länge $2i \mapsto$ linker unterer Endpunkt in $[0, 1)^2$
unshuffle: $q = (q_{1,x}q_{1,y} \dots q_{i,x}q_{i,y}) \mapsto (\sum_{j=1}^i q_{j,x}2^{-j}, \sum_{j=1}^i q_{j,y}2^{-j})$
- Punkt P in $[0, 1)^2$ (in Nachkommadarstellung, mit $m \geq i$ Bits)
 \mapsto Zellennummer bzgl. Gitterstufe $i =: \mathbf{Z-Code}(P, i)$
shuffle: $(.x_1 \dots x_m, .y_1 \dots y_m) \mapsto (x_1y_1x_2y_2 \dots x_iy_i)$

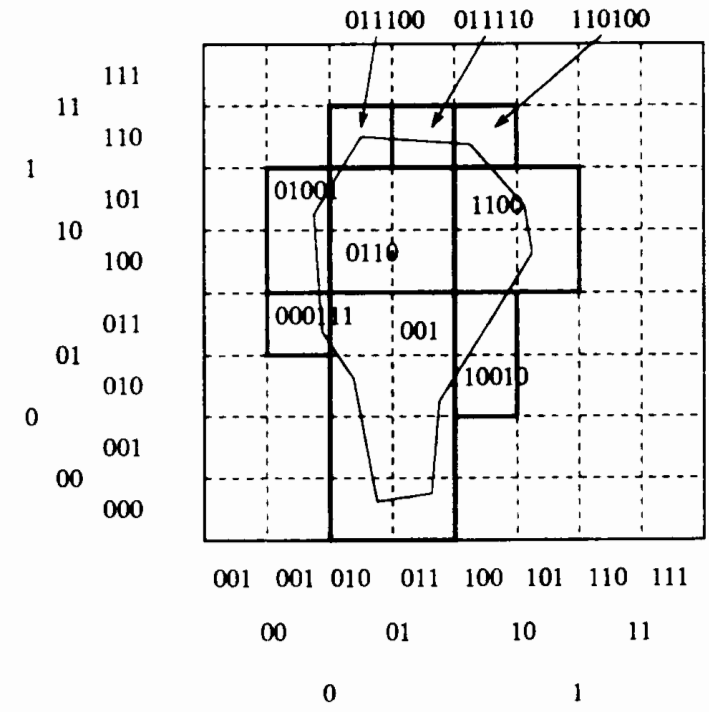
Z-Ordnung = Z-Codes (Forts.)



a



b



c

Figure 28. Z-ordering of a polygon.

Z-Ordnung = Z-Codes (Forts.)

Algorithmus: Rekursive Ermittlung der “maximalen Z-Code-Teilzellen” zu einem Objekt Q mit Ausdehnung $\mathrel{=}: \text{Z-Codes}(Q)$:

maxTeilzelle (Z-Code p einer Gitterzelle, Objekt Q , Gitterstufe j):

boolean

(true falls p Zelle,

die Q schneidet und deren direkte/indirekte Teilzellen alle Q schneiden;
mit Ausgabe der Z-Codes solcher Zellen auf kleinstmöglicher Gitterstufe)

if $j = \text{maximale Gitterstufe}$ **then return** $p \cap Q \neq \emptyset$

else if $p \subseteq Q$ **then return** true

else for $k, l = 0, 1$ **do** $b_{kl} := \text{maxTeilzelle}(p_{kl}, Q, j + 1)$;

if $b_{00} \wedge b_{01} \wedge b_{10} \wedge b_{11}$ **then**

if $j = 0$ **then** **Ausgabe**(p); **return** true

else

for $k, l = 0, 1$ **do if** b_{kl} **then** **Ausgabe**(p_{kl});

return false.

Aufruf: **maxTeilzelle**($\varepsilon, Q, 0$)

Z-Ordnung = Z-Codes (Forts.)

Verbesserung: betrachte auch Halbzellen (wie in Abbildung!),
zähle Bitfolgenlänge $2j-1$, $2j$ statt Gitterstufe j

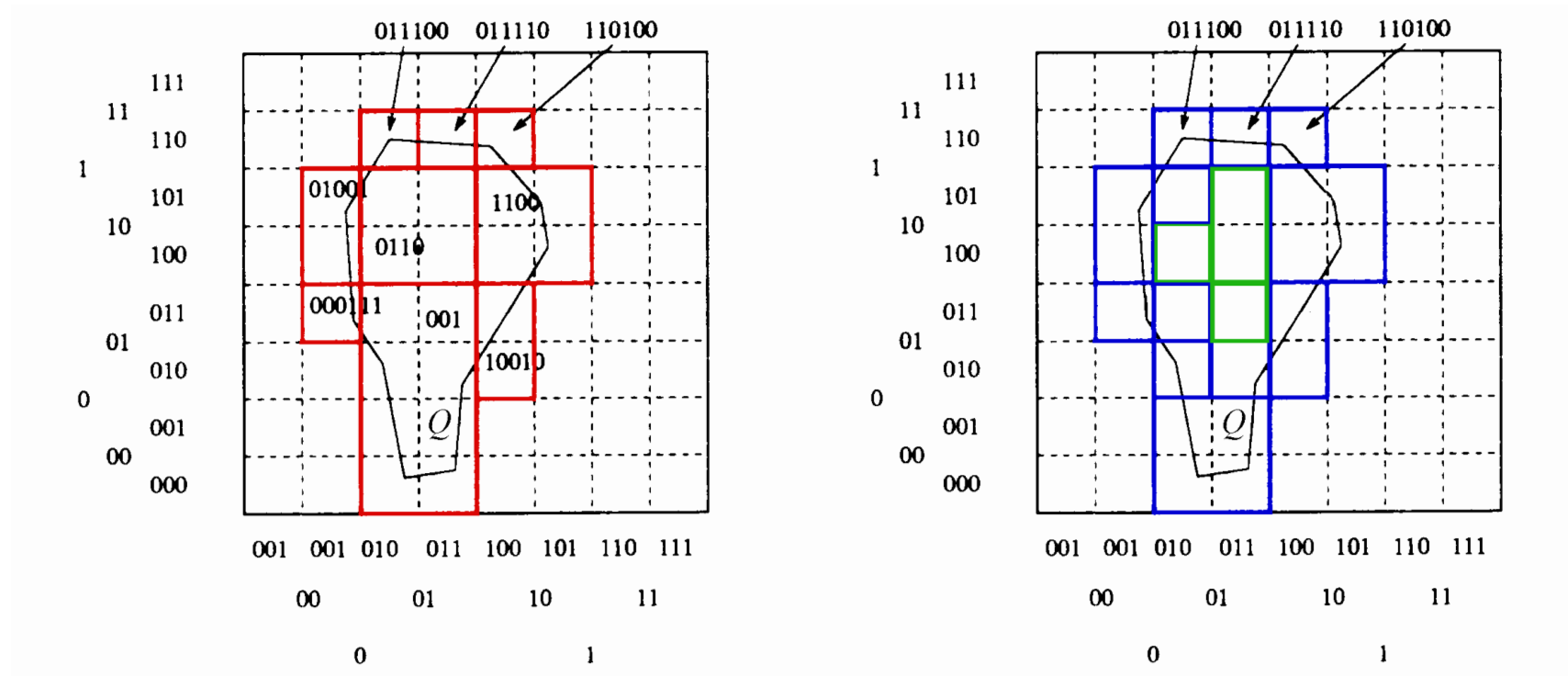
Variante: Ermittle maximale Teilzellen zu Q , aber mit Merkmal,
ob Teilzelle $p \subseteq Q$ ('innen') oder $p \not\subseteq Q \wedge p \cap Q \neq \emptyset$ (am 'Rand')

maxRandTeilzelle (p , Q , j): boolean

```
if  $p \subseteq Q$  then Ausgabe( $p$ , 'innen'); return false
else if  $j =$  maximale Gitterstufe then return  $p \cap Q \neq \emptyset$ 
else for  $k, l = 0, 1$  do  $b_{kl} :=$  maxRandTeilzelle( $p_{kl}$ ,  $Q$ ,  $j+1$ );
    if  $b_{00} \wedge b_{01} \wedge b_{10} \wedge b_{11}$  then
        if  $j = 0$  then Ausgabe( $p$ , 'Rand'); return true
    else
        for  $k, l = 0, 1$  do if  $b_{kl}$  then Ausgabe( $p_{kl}$ , 'Rand');
    return false
```

Die Differenzierung erleichtert die Bearbeitung von Anfragen; für Kandidaten P in Innenzellen braucht $P \in Q$ nicht mehr nachgeprüft werden.

Beispiel



links: maximale Teil(halb)zellen für Objekt Q =

000111, 001, 01001, 0110, 011100, 011110, 10010, 1100, 110100

rechts: maximale Innen-/Rand(halb)teilzellen ...

Z-Ordnung = Z-Codes (Forts.)

Indexierung und Anfragen:

- Indexierung eines Punkts P unter seinem Z-Code bzgl. max. Gitterstufe in einem klassischen B*-Baum
- Indexierung eines Objekts Q unter allen seinen Z-Codes (Approximations-Teilzellen) wie oben in einem klassischen B*-Baum
- Suche nach einem gespeicherten Punkt P = Suche nach seinem Z-Code bzgl. max. Gitterstufe im Index
- Suche nach einem Objekt Q als Suchbereich = Suchen [Mehrzahl] nach allen seinen Z-Codes (Such-Teilzellen) gegen Index
- *Suche gegen Index:* Such-Teilzelle p schneidet Approximations-Teilzelle q gdw. p Präfix von q oder q Präfix von p
also zu jeder Such-Teilzelle: Suche danach (als Präfix) und nach allen ihren Präfixen (exakt) im Index

Z-Ordnung = Z-Codes (Forts.)

- anschließend exakter Vergleich der Koordinatenwerte bzw. OIDs der Kandidaten-Objekte in den gefundenen Approximations-Teilzellen

- d.h. z.B. für *Bereichsanfrage*:

Welche Punkte liegen in Anfragebereich Q ?:

Jede Suchzelle p von Q der Stufe j liefert aus dem Index nacheinander alle gespeicherten Punkte P mit

$$\text{substring}(\text{shuffle}(P), 1, 2j) = p$$

(eindimensionaler Index-Teilscan); für diese “Kandidaten” ist dann noch $P \in Q$ exakt nachzuprüfen.

Z-Ordnung = Z-Codes (Forts.)

Anmerkungen:

- Es werden nur klassische Speicherstrukturen benötigt. Allerdings müssen Operationen auf Bitstrings effizient unterstützt sein.
- *Tradeoff*: geringere Anzahl von Approximationszellen, also weniger Indexeinträge vs. genauere Approximation, weniger falsche Kandidaten
- Zudem sollten Objekte im externen Speicher nach Z-Codes (Nachbarschaft!) geclustert gespeichert werden, um Blockzugriffe nach Auffinden der OIDs im Index zu minimieren.
- Bei ungleichmäßigen Objektverteilungen sollte mit variabler maximaler Gittertiefe (für die kleinsten Suchzellen) gearbeitet werden.
- Z-Codes sind auf beliebige Dimensionsanzahlen d verallgemeinerbar. Der shuffle-Operator muss dann pro Stufe d statt 2 Bits mischen.

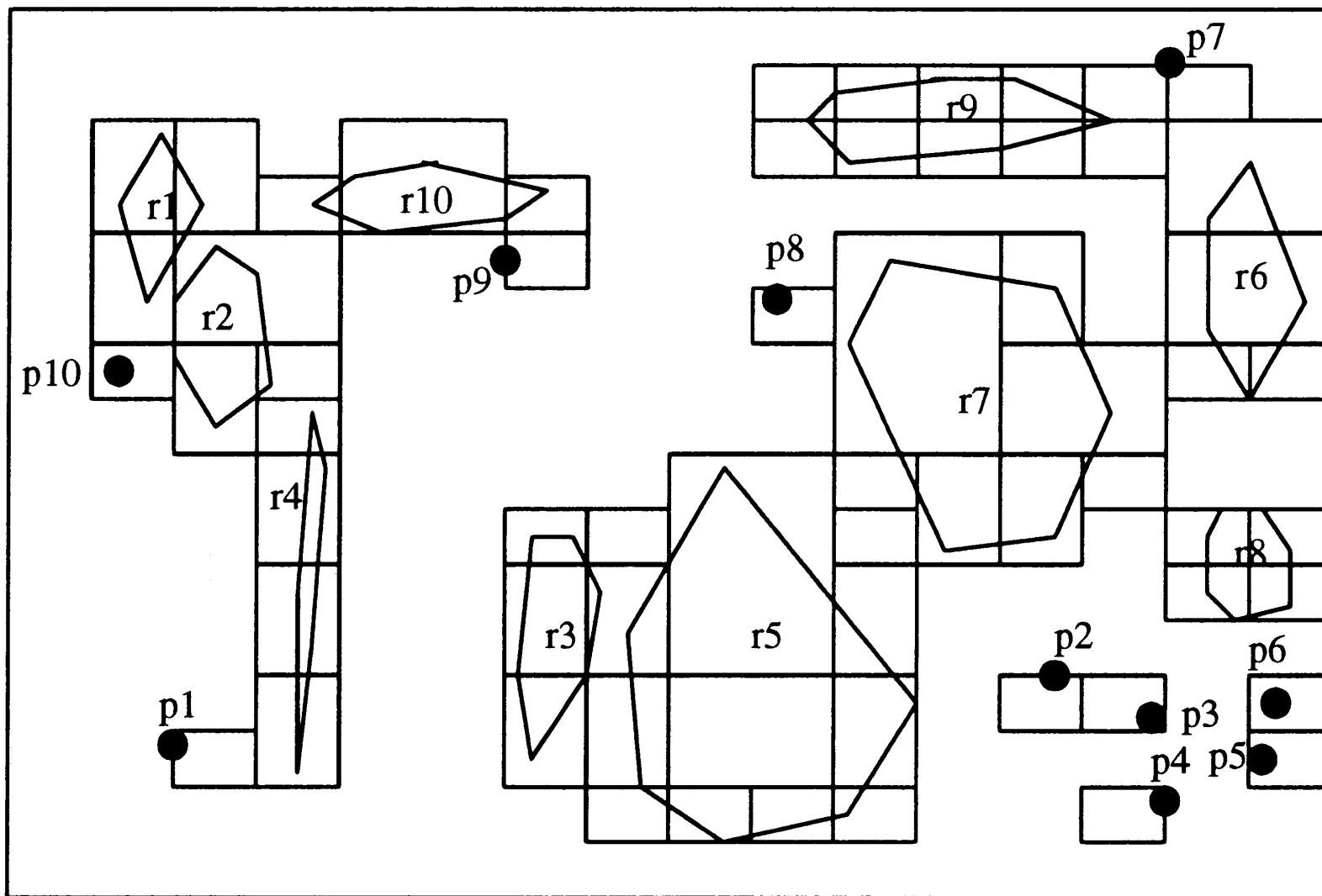


Figure 29. Z-ordering.

6. Räumliche Verbunde (Spatial Joins)

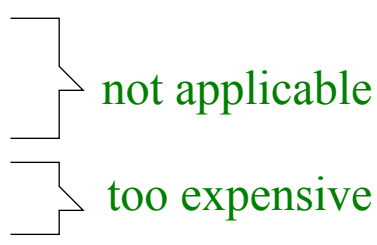
Definition: Gegeben seien zwei (große) Mengen O_1, O_2 räumlicher Objekte, mindestens mit Objekt-Identifizier oid , Geometrie geo und minimalem umgebenden Rechteck mbr^1 .

Der **räumliche Verbund** $O_1 \bowtie_{\cap} O_2$ von O_1 und O_2 ist die Menge von (Objekt- oder OID-) Paaren $\{ o_1.oid, o_2.oid \mid o_1.geo \cap o_2.geo \neq \emptyset \}$.

Variante: $\dots \subseteq$ statt $\cap \dots$

¹*minimum bounding rectangle*

Active area in the last few years.

- sort / merge join
 - hash join
 - filtering the cartesian product
- 
- not applicable
- too expensive

Central ideas:

- **filter** step (join bounding boxes = rectangles)
+ **refine** step (check exact geometries)
- use of spatial index structures for the filter step

Classification of strategies:

- grid approximation / bounding box
- none / one / both operand sets represented in a spatial index

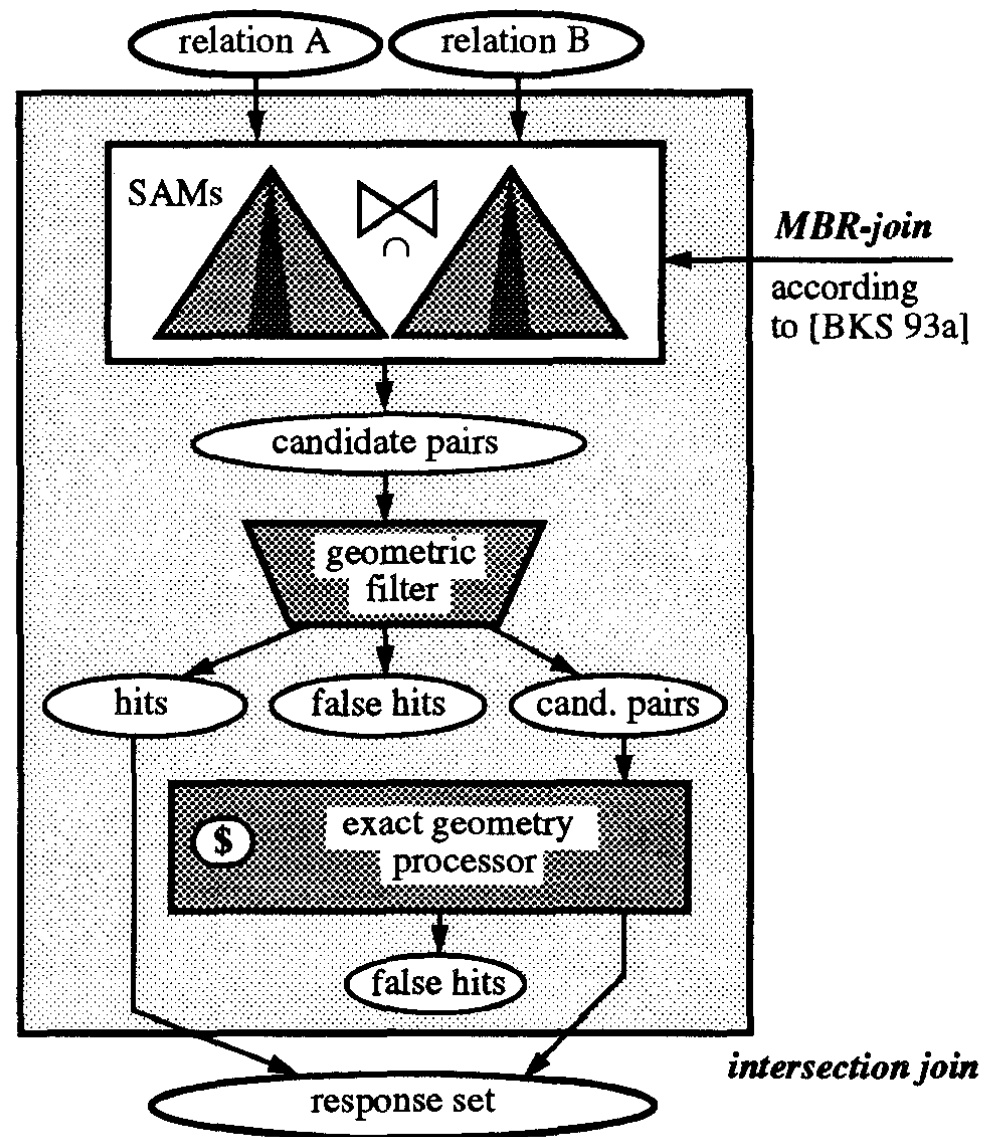
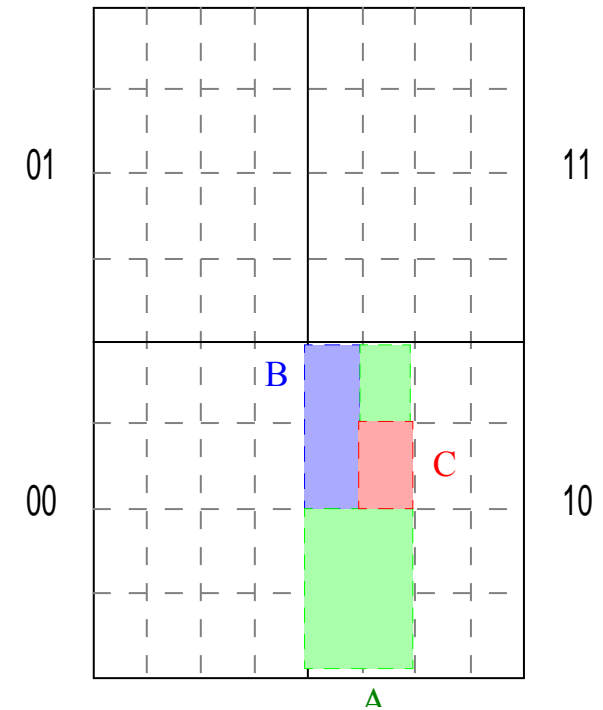
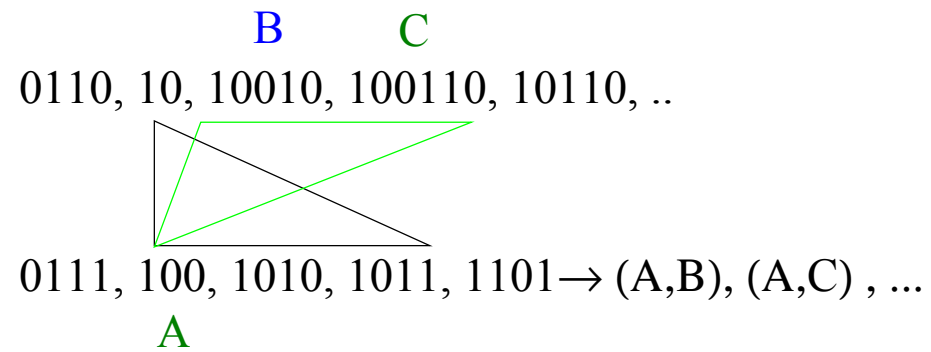


Figure 1: Spatial join processor

The Filter Step

Grid approximations

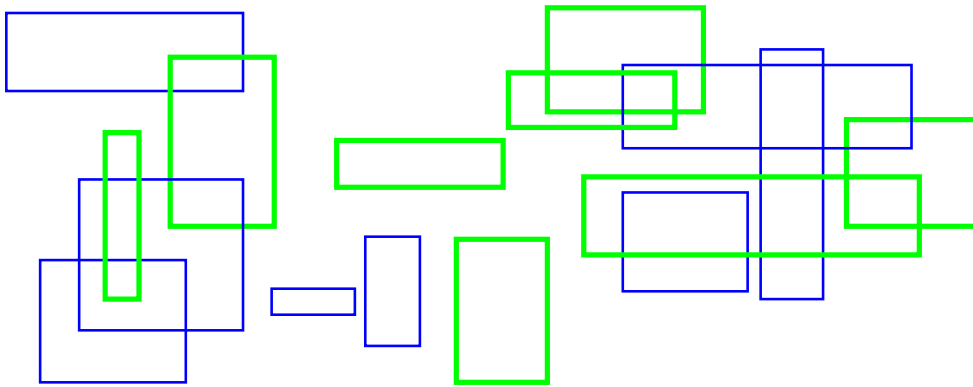
E.g., overlap \rightarrow parallel scan through the sets of Z-codes



Bounding box approximations

(1) None of the operands represented in a spatial index

→ rectangle intersection problem



determine all pairs (p, q) , p intersects q

→ bb-join operation, general basis for spatial join

Needed as a query processing method in any case:

```
cities select[pop > 500 000]
states select[language = "french"]
join[center inside area]
```

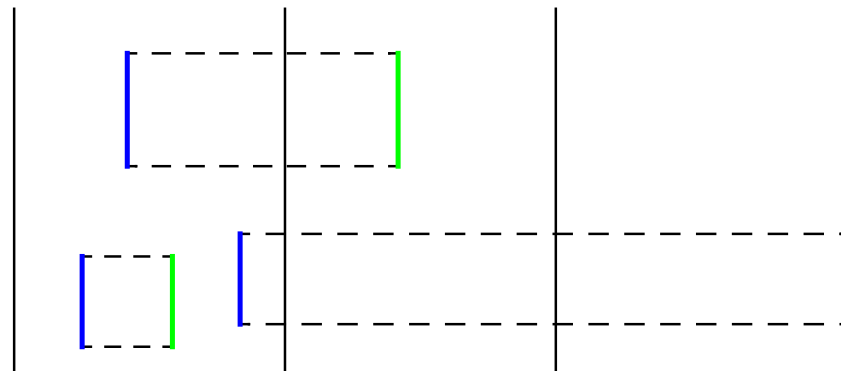
} no index
any more

Proposed solutions:

- External *divide-and-conquer algorithm* (Güting & Schilling 87), adapted from internal computational geometry algorithm :

Finds all intersecting pairs in *one* set of rectangles.

Simple modification to treat two sets (Becker&Güting 92).



Divide plane into vertical stripes such that each stripe contains about *c* vertical edges of rectangles. Compute intersections between rectangles represented in the stripe by internal DAC algorithm. *External*: Merge adjacent stripes bottom-up, *as in external merge sort*, writing intermediate structures into files again.

- “*Spatial hash join*” (Lo & Ravishankar 96, Patel & De Witt 96). Assign the two sets of rectangles to two sets of buckets; process pairs of buckets internally.
Many design choices:

Spatial Hash Joins

- Vergleiche klassischen Hash-Join von zwei Datenbanktabellen O_1 und O_2 über die Gleichheit von zwei Attributen A_1 bzw. A_2 ($O_1 \bowtie_{A_1=A_2} O_2$):

1. durchlaufe O_1 und ordne Objekte in Hash-Buckets gemäß Hashfunktion

$$h_1 : A_1\text{-Werte} \rightarrow \text{Nummern von } O_1\text{-Buckets}$$

ein (Einordnung₁, Buckets₁)

h_1 kann sogar die Identität sein (Einordnung nach Attributwerten).

2. ebenso O_2 (Einordnung₂, Buckets₂)
 3. Die Buckets₁ und Buckets₂ sind einander 1:1 zugeordnet, typischerweise über gleiche Nummern. Bilde den Gesamt-Join durch Vereinigung der Joins zwischen einander zugeordneten Buckets (Bucketjoin).
- ⇒ Einordnung_i eindeutig, Buckets_i fest, Bucketmengen disjunkt, Bucketjoin 1:1

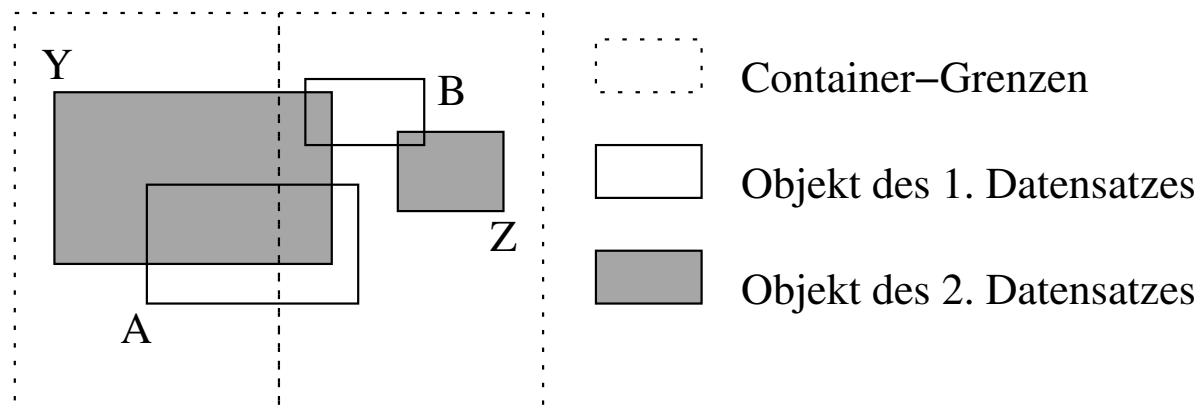
- Wie läßt sich dieses Vorgehen auf einen räumlichen Verbund übertragen ?

Die Einordnungen der Objekte in Buckets muss natürlich in Abhängigkeit von ihren *mbr*-Werten erfolgen.

Spatial Hash Joins (Forts.)

1. Alternative:

- * **Buckets_i** : Zellen eines festen Gitters
- * **Einordnung_i** : Objekt $o \in O_i \mapsto$ alle Buckets_i, die o überlappen, also mehrdeutig
- * **Bucketjoin**: 1:1
- * **Nachteile**: Buckets evtl. ungleichmäßig gefüllt, Mehrfachdarstellung von Objekten, Bucketjoin erfordert Duplikateliminierung



Spatial Hash Joins (Forts.)

2. *Alternative:*

- * **Buckets_i** : anfangs Zellen eines festen Gitters; dann wachsen Buckets unabhängig voneinander so, dass Objekte eingeschlossen werden
- * **Einordnung_i** : Objekt $o \in O_i \mapsto \text{Bucket}_i$ mit minimaler Flächenvergrößerung, um o einzuschließen (eindeutig)
- * **Bucketjoin**: m:n, da Buckets beliebig überlappen können
- * *Nachteil*: nested-loop-Algorithmus für Bucketjoin

3., *empfohlene Alternative:* [=Lo&Rav.96]

- * **Buckets₁** : anfangs nur “Saat”-Punkte (etwa Mittelpunkte von aus Stichprobe erwarteten Häufungen); dann wachsen Buckets so wie in Alternative 2; Bucketeinteilung basiert also auf Objektverteilung.
- * **Einordnung₁** : wie bei Alternative 2, eindeutig
- * **Buckets₂** := (immer) Buckets₁ !!!
- * **Einordnung₂** : wie bei Alternative 1, mehrdeutig, aber Join-vorbereitend (keine Duplikateliminierungen nötig, keine unnötigen Flächenüberlappungen)
- * **Bucketjoin**: 1:1

(2) One of the operands represented in a spatial index

→ index join, repeated search join. Scan “outer” operand set; for each object perform a search with the bounding box on the index for the “inner” operand.

(3) Both operands are represented in a spatial index

Basic idea: Synchronized, parallel traversal of the two data structures

- grid files
(Rotem 91, Becker, Hinrichs & Finke 93)
- R-trees ("*R-tree join*")
(Brinkhoff, Kriegel & Seeger 93; refined version with breadth-first traversal: Huang, Jing & Rundensteiner 97)
- generalization trees(Günther 93)

Further idea: Use of join indices (Rotem 91, Lu & Han 92)

R-Tree Join

Basis-Algorithmus^{2,3} :

rtree-join (Knoten V_1 , Knoten V_2 , Suchfenster w):

if not (V_1 Blatt **or** V_2 Blatt) **then**

for alle Einträge $E_1 \in V_1$ mit $E_1.mbr \cap w \neq \emptyset$ **do**

for alle Einträge $E_2 \in V_2$ mit $E_2.mbr \cap w \neq \emptyset$ **do**

if $E_1.mbr \cap E_2.mbr \neq \emptyset$ **then**

rtree-join($E_1.succ$, $E_2.succ$, $E_1.mbr \cap E_2.mbr$)

else if (V_1 Blatt **xor** V_2 Blatt) **then**

 ... analoger Abstieg nur für den Nicht-Blattknoten ...

else /* zwei Blattknoten */

for alle Einträge $E_1 \in V_1$ mit $E_1.mbr \cap w \neq \emptyset$ **do**

for alle Einträge $E_2 \in V_2$ mit $E_2.mbr \cap w \neq \emptyset$ **do**

if $E_1.mbr \cap E_2.mbr \neq \emptyset$ **then** **output**($E_1.oid$, $E_2.oid$).

Aufruf: **rtree-join** (Wurzel 1.Baum, Wurzel 2.Baum, ges. Datenraum)

²Alle umgebenden Objekt- und Verzeichnisrechtecke seien hier einheitlich mit *.mbr* zugreifbar.

³Weitere Optimierungen im Detail möglich, z.B. Plane-Sweep-artiger Durchlauf durch gleichsortierte Einträge zweier Knoten.