

5 Raumorganisierende Strukturen zur Sekundärschlüsselsuche

Ebenso wie bei eindimensionalen Hashverfahren des vorigen Kapitels lassen sich unterscheiden:

- *multidimensionale Verfahren ohne Directory*
- *multidimensionale Verfahren mit Directory*

Außerdem gibt es raumorganisierende Strukturen, die eine Kombination von Baum- und Hashverfahren vornehmen:

- *multidimensionale Verfahren mit Hashbaum-Directory*

Anwendungsbereich Nichtstandard-Datenbanksysteme [Küs 86]

Standard-Datenbanksysteme werden im administrativ-betriebswirtschaftlichen Bereich angewandt, zum Beispiel:

- Personalwesen
- Buchungs- und Abrechnungswesen (Platzbuchung, Kontenführung)
- Produktionsplanung und -steuerung
- Finanz- und Investitionsplanung

Typischerweise treten dabei sehr *viele kleine, einfach strukturierte Objekte* (Punktobjekte) auf.

Nichtstandard-Datenbanksysteme (NDBS) haben Anwendungen in Bereichen wie z.B.:

- Bild- und Sprachverarbeitung
- Geographische Informationssysteme (GIS)
- Entwurf und Fertigung (CAD / CAM)
- Medizin und Biologie
- Anwendungen regelbasierter Expertensysteme

NDBS müssen *mehrdimensionale, oft komplex strukturierte Objekte* (komplexe Raumobjekte) verwalten.

Kriterien für Indexstrukturen von NDBS [HR 85]:

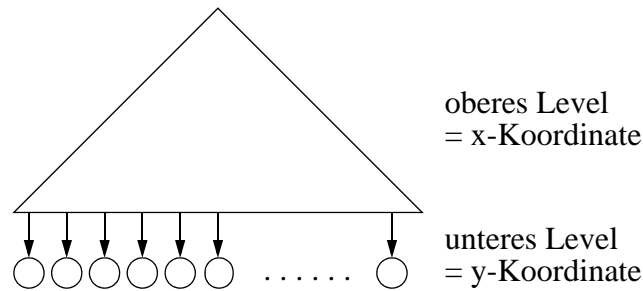
- Erhaltung der topologischen und geometrischen Eigenschaften der Objekte (*Clusterbildung*)
- Darstellung von Punkt- und Raumobjekten
- Dynamisches Verhalten
- Eignung für Sekundärspeicher

Beispiel: Speicherung geometrischer Daten

- Multidimensionale B-Bäume (und deren Varianten) sind für die Speicherung von mehrdimensionalen alphanumerischen Schlüsseln geeignet, wie sie in Standard-Datenbanksystemen auftreten.
- Für die Speicherung *mehrdimensionaler, geometrischer Daten* in NDBS wie z.B. Punktdaten eignen sie sich hingegen nicht.

Beispiel:

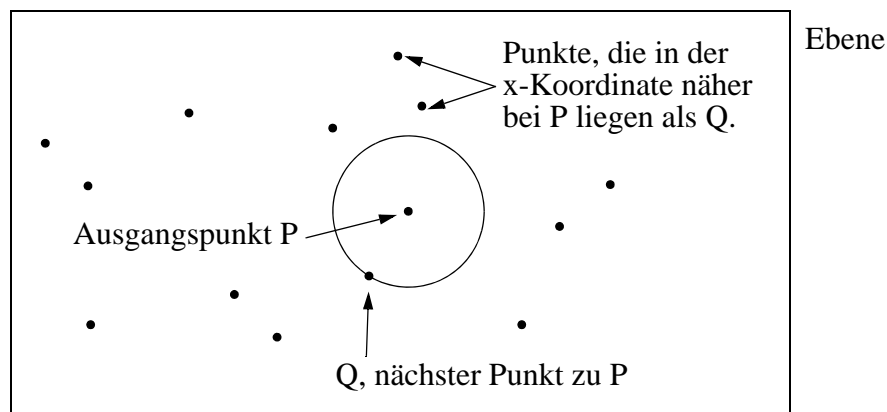
Die Speicherung von zweidimensionalen Punkten (x,y) der Ebene in einem 2B-Baum:



- Fast alle Punkte unterscheiden sich in der x-Koordinate
 ⇒ die Teilbäume des unteren Levels bestehen nur aus einer y-Koordinate
 ⇒ der zweidimensionale Baum degeneriert zu einer eindimensionalen Struktur
Ursache: Die Dimensionen werden nicht gleichberechtigt behandelt.
- Mit eindimensionalen Strukturen, die z.B. die Punkte nur nach einer Koordinate sortieren, lassen sich viele geometrische Anfragen nicht mehr effizient beantworten.

Beispiel: Nearest Neighbour Query

Die Suche nach dem zu einem Punkt P am nächsten gelegenen Punkt Q



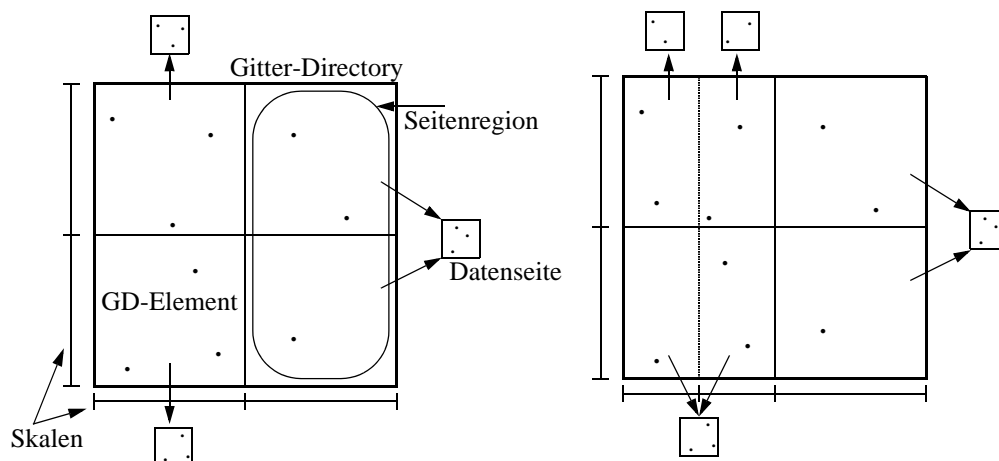
⇒ Bei geometrischen Daten müssen alle Koordinaten gleichberechtigt in der Indexstruktur behandelt werden.

5.1 Verfahren mit Directory

Gridfile [NHS 84]

Grundstruktur

- **Grid**
Der Datenraum wird durch ein k-dimensionales orthogonales Gitter (*Grid*) partitioniert.
- **Skalen**
definieren die Einteilung des Gitters für jeweils eine Dimension.
- **Grid-Directory (GD)**
k-dimensionales dynamisches Array, in dem das Gitter gespeichert wird.
- **GD-Element**
Zelle dieses Gitters, mit einem Zeiger zu der Datenseite (*Data Bucket*), die alle Datensätze enthält, die in der betreffenden Gitterzelle liegen.
- **Seitenregion (Bucket Region)**
Um eine niedrige Speicherplatzausnutzung der Datenseiten zu verhindern, dürfen mehrere GD-Elemente auf dieselbe Datenseite verweisen. Eine solche Menge von GD-Elementen heißt *Seitenregion (Bucket Region)*. Es gilt:
 - Seitenregionen dürfen nur die Gestalt k-dimensionaler Hyperrechtecke haben.
 - Seitenregionen sind paarweise disjunkt.
 - die Vereinigung der Seitenregionen ergibt den gesamten Datenraum.



Dynamisches Verhalten

- Das Gitter wächst (und schrumpft) dynamisch.
- Dabei ist das GD in der Regel aber so groß, daß es auf dem Sekundärspeicher abgespeichert werden muß.
- Entsprechend dem Gitter verändern sich auch die Skalen.
- Die Skalen sind klein genug, um im Hauptspeicher gelagert zu werden. Daher bieten sich zur ihrer Verwaltung dynamische Datenstrukturen wie AVL-Bäume oder 2-3-Bäume an.

Operationen

Suche

Das Vorgehen für eine *Exact Match Query*:

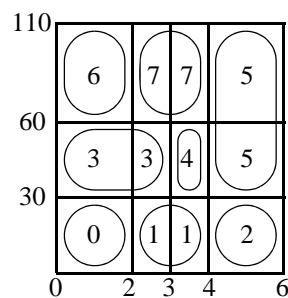
- (i) Beim Durchsuchen der Skalen werden die k Attributswerte eines Datensatzes in Intervallindizes umgewandelt (ohne Seitenzugriff).
- (ii) Diese Indexkombination erlaubt direkten Zugriff zu dem GD-Element, das den Zeiger zu der gewünschten Datenseite enthält (1. Seitenzugriff).
- (iii) Mit einem 2. Seitenzugriff wird die korrekte Datenseite in den Hauptspeicher übertragen, die man dann nach dem Datensatz durchsucht.

⇒ 2 Seitenzugriffe für erfolgreiche und erfolglose *Exact Match Queries*.

Für komplexere Anfragen (wie *Partial Match*, *Range* oder *Partial Range Queries*) sind weit mehr Seitenzugriffe durchzuführen.

Einfügen eines Datensatzes

Beispiel:
Ausgangssituation



- Suche der Datenseite.
- Einfügen des Datensatzes in der Datenseite.

Fall: Datenseite läuft über.

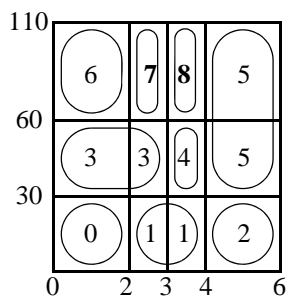
Als Hashverfahren mit Directory vermeidet das Gridfile Überlaufseiten.

⇒ Die überlaufende Datenseite wird in zwei aufgespalten.

Fall 1: Die Seitenregion der überlaufenden Datenseite umfaßt mehr als ein GD-Element.

- Man aktiviert eine Partitionierungslinie, die die Seitenregion schneidet und teilt die Datenseite nach dieser Partitionierungslinie in zwei auf.

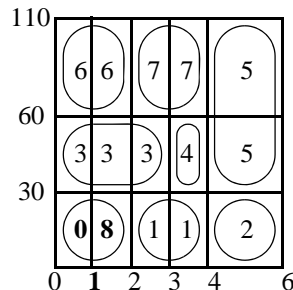
Beispiel:
Seite 7 läuft über



Fall 2: Die Seitenregion der überlaufenden Datenseite umfaßt nur 1 GD-Element.

- Eine neue Partitionierungslinie muß eingeführt werden.
 - 1.) Einfügen der Partitionierungslinie in die entsprechende Skala.
 - 2.) GD wird um eine (k-1)-dimensionale Scheibe erweitert. Auf diese “Scheibe” werden die alten Adressen bis auf das “überlaufende” GD-Element kopiert.

Beispiel:
Seite 0 läuft über



Wahl der Partitionierungslinien

Die Wahl einer Partitionierungslinie besteht aus:

1. Wahl der *Splitachse*
2. Wahl der *Splitposition*

Wahl der *Splitachse* (Möglichkeiten)

- Achsen, die in Partial Match und Partial Range Queries häufiger spezifiziert werden, sollten auch häufiger durch Partitionierungslinien gesplittet werden.
- Hat man kein Wissen über die Form der Bereichsanfragen, sollte man möglichst quadratische Seitenregionen anstreben, d.h. man nimmt die Achse, bei der die Seitenregion die größte Seitenlänge besitzt.

Wahl der *Splitposition* (Möglichkeiten)

- **Mittensplit:**
Halbieren der Datenseite.
Motivation: der *Datenraum* wird möglichst gleichmäßig aufgeteilt.
- **Mediansplit:**
Aufteilen der Datenseite, derart daß in beide neuen Seiten gleich viele Datensätze kommen.
Motivation: die *Daten* werden möglichst gleichmäßig aufgeteilt.
Bei Gleichverteilung (und nichtsortierter Eingabe) sind beide Methoden äquivalent.

Verschmelzen von Datenseiten

Werden Datensätze gelöscht, können Datenseiten auftreten, die nur gering gefüllt sind.

⇒ Solche Seiten sind mit anderen zu verschmelzen.

Frage:

Mit welcher Datenseite bzw. mit welchen Datenseiten (*Kandidaten*) verschmilzt man die unterfüllte Seite ?

Basisvoraussetzung:

Die Seitenregion der resultierenden Datenseite muß ein Hyperrechteck sein.

- **Twin-System**

Eine Seite darf nur mit dem Bruder im zugehörigen binären Splitbaum verschmolzen werden. Der Splitbaum gibt die Geschichte aller Splits wieder.

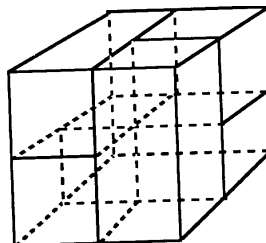
Einfache Kandidatenbestimmung und einfacher Test der Basisvoraussetzung.

- Es gibt nur einen Kandidaten.
- Ein Verschmelzen ist oft nicht möglich ⇒ schlechte Speicherplatzausnutzung.

- **Buddy-System**

Eine Seite darf nur mit den Brüdern verschmolzen werden, die bezüglich einer Dimension durch einen Split getrennt wurden. Diese Information kann in den Skalen gespeichert werden.

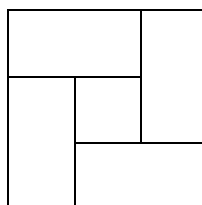
- Es gibt k Kandidaten.
- Es können (bei $k > 2$) *Verklümmungen (Deadlocks)* auftreten, so daß kein Verschmelzen zweier Datenseiten mehr möglich ist.



- **Nachbar-System**

Eine Seite darf mit einer beliebigen benachbarten Seite verschmolzen werden.

- $2k$ Kandidaten.
- Beste Speicherplatzausnutzung.
- Es können (bei $k > 1$) Deadlocks auftreten.



Leistungsverhalten

Das Zeitverhalten bei komplexen Anfragen, z. B. Range Queries, hängt insbesondere von der Größe des Grid-Directory ab.

Analyse [Reg 85]:

- Der Erwartungswert $|GD(N)|$ für die Größe des Grid-Directory bei *Gleichverteilung und Unabhängigkeit* aller k Attribute ist:

$$|GD(N)| = O\left(N^{1 + \frac{k-1}{bk+1}}\right)$$

Dabei ist N die Anzahl der Datensätze und b die Kapazität einer Datenseite.

$O\left(N^{1 + \frac{k-1}{bk+1}}\right)$ terminiert für $k \rightarrow \infty$ gegen $O\left(N^{1 + \frac{1}{b}}\right)$

- Bei *Gleichverteilung und Abhängigkeiten* gilt: $|GD(N)| = O(N^k)$
- Bei *Nichtgleichverteilung* gilt: $|GD(N)| = O(2^N)$
- Neben den Speicherplatzproblemen haben wir insbesondere ein extrem schlechtes Leistungsverhalten bei komplexen Anfragen.

Varianten des Gridfiles

Aufgrund des schlechten Leistungsverhaltens des Gridfiles liefen eine Reihe von Anstrengungen, das superlineare Wachstum des Grid-Directory zu mildern:

- *2-Level-Gridfile* [Hin 85]
- *Multilevel-Gridfile* [KW 85]
- *Interpolation-based Gridfile* [Ouk 85]

Keines dieser Konzepte kann das superlineare Wachstum verhindern !

2-Level Grid File

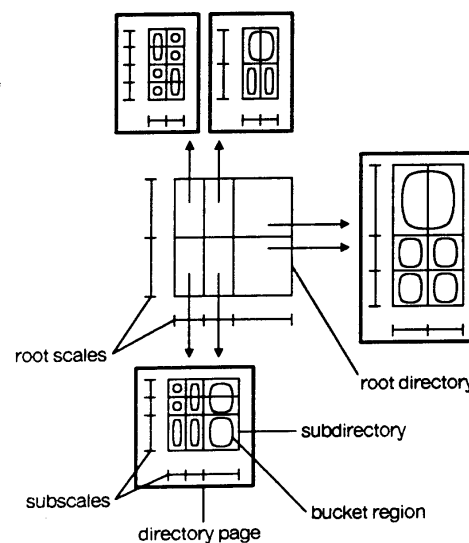
Beobachtung:

- Häufungspunkte in den Daten führen wegen der durch den ganzen Datenraum durchgezogenen Partitionierungslinien zum starken Wachstum des Directory.

Idee:

- Es wird eine zweite Hierarchie-Stufe eingeführt, d.h. das Grid-Directory wird über ein (zweites) Grid-Directory verwaltet.

Aufbau:



1. Level:

Wurzel-Gridfile mit *Wurzel-Skalen* (*root scales*) und *Wurzel-GD* (*root directory*).

- ist klein genug um im Hauptspeicher gehalten zu werden.

2. Level:

Sub-Gridfiles mit *Subskalen* (*sub-scales*) und *Sub-GDs* (*subdirectories*).

- Ein Element des Wurzel-GDs verweist zu einem Sub-Gridfile.
- Ein Sub-GD ist auf ein oder mehreren Directoryseiten (*directory pages*) gespeichert.

Eigenschaften:

- (i) Verschiedene Elemente des Wurzel-GDs können zu einer Directory-Region zusammengefaßt werden, falls die Directory-Region einem Hyperrechteck des Datenraums entspricht.
- (ii) Wurzel-Gridfile und Sub-Gridfiles werden dynamisch organisiert (wie das normale Gridfile).
- (iii) Seitenregionen der Sub-Gridfiles müssen in genau einer Directory-Region des Wurzel-Gridfiles liegen.

(iii) impliziert:

Schneidet eine Seitenregion eine Directory-Region, so muß die zugehörige Datenseite gesplittet werden.

(iii) ist notwendig,

um die 2-Seiten-Zugriffs-Garantie für Exact Match Queries zu gewährleisten.

- (iv) Jedes Sub-Gridfile besitzt eigene Partitionierungslinien, unabhängig von denen anderer Sub-Gridfiles, d.h. es gibt *lokale Partitionierungslinien*.

(iv) ermöglicht:

bessere Anpassung an Häufungspunkte

aber:

Das Wurzel-GD wächst noch immer superlinear.

Hauptnachteil:

- (v) Das 2-Level Grid File ist eine *keine beliebig dynamische Struktur*, da die Anzahl der Level fest ist.
Insbesondere bei Nicht-Gleichverteilungen der Daten ist dieser Ansatz in seiner Leistungsfähigkeit beschränkt.

Lösung: Einführen weiterer Level.

Dieser Ansatz führt zur Entwicklung von *Hashbaum-Directories*, auf die später eingegangen wird.

Leistungsuntersuchungen

Frage: Welche Indexstruktur ist am effizientesten ?

- *Analytische Untersuchungen* sind meist nur für den schlechtesten Fall (*worst case*) möglich, nicht jedoch für beliebige Verteilungen von Daten und Operationen.
⇒
- *Experimenteller Leistungsvergleich* von implementierten Indexstrukturen.

Testumfeld

- **getestete Operationen**
 - *Suchoperationen*
(z.B. Exact Match, Range oder Partial Match Queries)
 - *Update-Operationen*
(z.B. Einfügen und Löschen)
- **gemessene Parameter** (pro Operation), z.B.:
 - CPU-Zeit
 - Anzahl der Seitenzugriffe
 - Anzahl der Antworten bei Anfragen
- **Daten**, mit denen die Untersuchung durchgeführt wird:
 - *synthetische Daten*
(gemäß statistischer Verteilungen)
 - *reale Daten*
(z.B. Personaldaten, geographische Karten)

Untersuchungen

- **Verteilung der Daten in realen Anwendungen**

In Rahmen seiner Dissertation an der University of Toronto im Jahre 1981 hat Christodoulakis reale Datenfiles untersucht:

- Annahmen der Gleichverteilung und der stochastischen Unabhängigkeit sind in realen Files i.allg. nicht erfüllt.
⇒ Gridfile und 2-Level-Gridfile sind für die Speicherung von Real-Daten nicht gut geeignet.

- **Vergleich von Indexstrukturen**

- Das Gridfile ist sehr bekannt.
- Es ist mit seiner Implementierung in Universitätskreisen weit verbreitet

⇒ Das Gridfile ist oft Maßstab für experimentelle Untersuchungen:

- In [Kri 84] wurde das Gridfile gegen bisher vorgestellte Verfahren (invertierte Listen, MDB-Bäume und KB-Bäume) getestet.

Für nichtgleichverteilte, abhängige Files gilt:

- Der KB-Baum ist für komplexe Anfragen (Partial Match, Range, Partial Range) dem Gridfile überlegen.
- Das Gridfile ist für Exact Match Queries überlegen.
- In [KSSS 89] wurde das Gridfile gegen Verfahren getestet, die in den letzten Jahren entwickelt worden sind und im weiteren Verlauf dieses Skriptes vorgestellt werden.

5.2 Verfahren ohne Directory

Hauptproblem des Gridfiles:

- Wachstum des Directory

⇒ Entwicklung von **multidimensionalen Hashverfahren ohne Directory**

Bei der Entwicklung des nächsten Verfahrens ging man in zwei Schritten vor:

1. Schritt:

Entwicklung eines ordnungserhaltenden Hashverfahrens ohne Directory, das eine sehr gute Leistungsfähigkeit bei *Gleichverteilung* besitzt:

multidimensionales ordnungserhaltendes lineares Hashing mit partiellen Erweiterungen [KS 86].

2. Schritt:

Entwicklung einer Methode, die erreicht, daß das multidimensionale ordnungserhaltende lineare Hashing mit partiellen Erweiterungen bei *Nichtgleichverteilung* praktisch dieselbe Performanz besitzt wie bei Gleichverteilung:

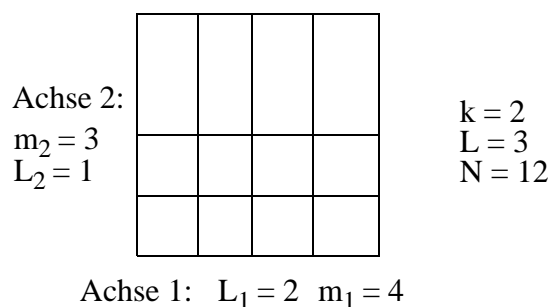
Quantilverfahren [KS 87].

Multidimensionales ordnungserhaltendes lineares Hashing mit partiellen Erweiterungen

Multidimensionales ordnungserhaltendes lineares Hashing

- *Dimension*
Ein Schlüssel x ist ein k -Tupel (x_1, \dots, x_k) der Dimension k .
- *Achsen*
Jede Achse wird mit $j \in \{1, \dots, k\}$ adressiert.
- *Level L*
gibt an (wie beim eindimensionalen linearen Hashing), wie oft sich die Datei verdoppelt hat.
- m_j : Anzahl der Partitionierungen der j -ten Achse
 L_j : Level der j -ten Achse
gibt an, wie oft sich die Zahl der Partitionierungen dieser Achse vollständig verdoppelt hat.
 - $2^{L_j} \leq m_j < 2^{L_j+1}$, $L_j = 0$ am Anfang
 - $L = \sum_{j=1}^k L_j$
- N : Anzahl der Primärseiten, die mit $0, \dots, N-1$ adressiert werden. Es gilt: $N \leq \prod_{j=1}^k m_j$

Beispiel:



Idee:

- Für jede Achse j wende man eine eindimensionale ordnungserhaltende lineare Hashfunktion (OLH) an
- Die Ergebnisse kombiniere man über einer weitere Funktion.

Vorgehen:*1. Berechnung des Pseudoschlüssels*

- Im weiteren müssen wir L_j+1 Bits betrachten, die vom j -ten Attributwertes x_j des Schlüssels x abgeleitet werden.

Dafür definieren wir für jedes Attribut A_j , $1 \leq j \leq k$, eine *Kodierungsfunktion* Ψ_j :

$$\Psi_j : \text{Domain}(A_j) \rightarrow \{0, 1\}^w$$

mit $\Psi_j(x_j) = (b_1, \dots, b_w)$ und $w \geq L_j + 1$.

$\Psi_j(x_j)$ heißt *Pseudoschlüssel* von x_j .

- Damit das Verfahren ordnungserhaltend ist, muß die Kodierungsfunktion $\Psi_j(x_j)$ die Ordnung erhalten:

$$x_j < \bar{x}_j \Rightarrow \Psi_j(x_j) \stackrel{L}{\leq} \Psi_j(\bar{x}_j) \quad \text{für } x_j, \bar{x}_j \in \text{Domain}(A_j)$$

wobei $\stackrel{L}{\leq}$ die lexikographische Ordnung auf Bitstrings ist.

- *Konkrete Spezifikation von Ψ_j* (Bitabschneidefunktion):

1. Transformation von $\text{Domain}(A_j)$ nach $[0, 1)$.

2. Betrachtung der w Präfix-Bits der binären Darstellung: $x_j = \sum B_{j_r} 2^{-r}$ für $1 \leq j \leq k$.

$$\Rightarrow \Psi_j(x_j) = (b_1, b_2, \dots, b_w) = (B_{j_1}, B_{j_2}, \dots, B_{j_w}) \text{ mit } w \geq L_j + 1.$$

- *Beispiel:*

$\text{Domain}(A_j) = [0 \dots 1)$.

Dezimaler Wert von x_j : 0.38

Binäre Darstellung von x_j : 0110 ...

$\Psi_j(x_j)$ mit $w = 3$: 011

2. Berechnung der Komponentenadressen

- Für jede Achse j , $1 \leq j \leq k$, bildet die *Hashfunktion* g_j die Attributswerte auf die Menge der Partitionierungen der j -ten Achse (*Komponentenadressen*) $\{0, \dots, m_j-1\}$ ab:

$$g_j(\Psi_j(x_j), L_j, m_j) := \begin{cases} \sum_{r=1}^{L_j+1} b_r \cdot 2^{r-1} & , \text{ falls } \sum_{r=1}^{L_j+1} b_r \cdot 2^{r-1} < m_j \\ L_j & , \text{ sonst} \\ \sum_{r=1}^{L_j} b_r \cdot 2^{r-1} & , \text{ sonst} \end{cases}$$

$$g_j(\Psi_j(x_j), 0, m_j) := 0$$

g_j nimmt die L_j bzw. $(L_j + 1)$ Präfix-Bits des Pseudoschlüssels $\Psi_j(x_j)$, dreht die Reihenfolge um und interpretiert das Resultat als ganze Zahl (Integer).

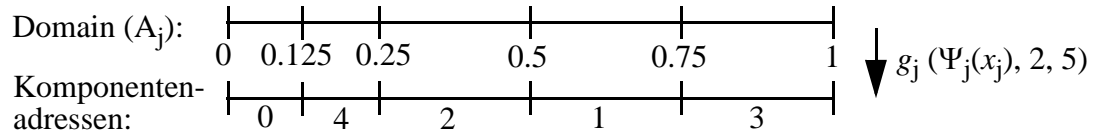
- g_j ist eine ordnungserhaltende Hashfunktion (OLH).

- *Beispiel:* (wie oben)

$$L_j = 2, m_j = 5.$$

$$g_j(\Psi_j(x_j), L_j, m_j) = g_j(011, 2, 5) = 10_2 = 2_{10}$$

Abbildung des Datenraumes in Komponentenadressen:



3. Berechnung der Primärseite über die Adreßfunktion G

- Für jedes $j, 1 \leq j \leq k$, sei $i_j := g_j(\Psi_j(x_j), L_j, m_j)$
- Wir wenden nun auf das Tupel (i_1, \dots, i_k) die Adreßfunktion G an, um die Adresse der Primärseite zu ermitteln:

$$G(i_1, \dots, i_k) \rightarrow \{0, \dots, N-1\}$$

- G ist die Adreßfunktion von multidimensionalem erweiterbarem Hashing [Oto 84]:

$$G(i_1, \dots, i_k) = \begin{cases} i_z \left(\prod_{j=1, j \neq z}^k J_j \right) + \left(\sum_{j=1, j \neq z}^k c_j \cdot i_j \right) & , \text{ falls } \max(i_1, \dots, i_k) \neq 0 \\ 0 & , \text{ sonst} \end{cases}$$

wobei:

$$z = \max\{j \in \{1, \dots, k\} \mid \lfloor \log_2 i_j \rfloor = \max_{1 \leq q \leq k} (\lfloor \log_2 i_q \rfloor)\}$$

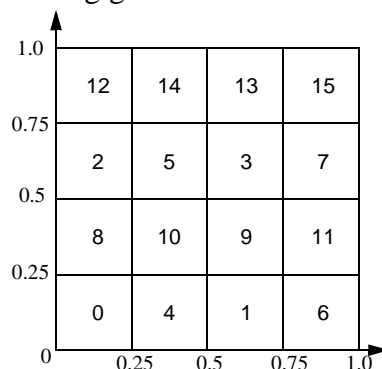
$$J_j = \begin{cases} 2^{s+1} & , \text{ falls } j < z \\ 2^s & , \text{ sonst} \end{cases} \quad \text{mit } s = \lfloor \log_2 i_z \rfloor$$

$$c_j = \prod_{r=j+1, r \neq z}^k J_r$$

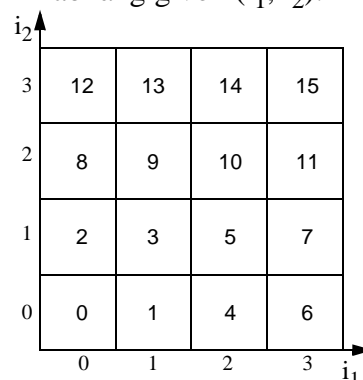
- *Beispiel:*

$k = 2, L = 4, L_1 = L_2 = 2, m_1 = m_2 = 4, \text{Domain}(A_1) = \text{Domain}(A_2) = [0 \dots 1)$.

Adressen abhängig vom Wertebereich:



Adressen abhängig von (i_1, i_2) :



Multidimensionales ordnungserhaltendes lineares Hashing mit partiellen Erweiterungen

Ziel:

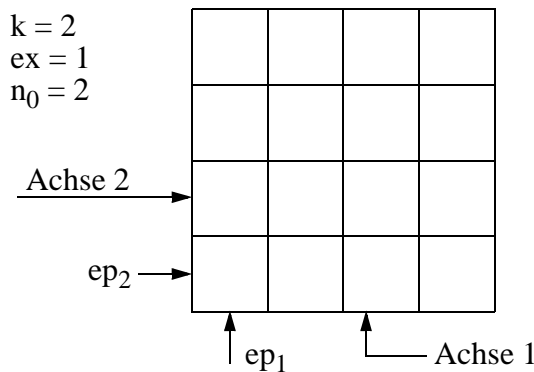
eine möglichst gleichmäßige Belegung aller Ketten

Ansatz:

das multidimensionale ordnungserhaltende lineare Hashing wird um das Konzept der partiellen Erweiterungen angereichert:

Multidimensional Order Preserving Linear Hashing with Partial Expansions (MOLHPE) [KS 86].

- Für jede Expansion gibt es eine Achse (Dimension) $ex \in \{1, \dots, k\}$, in der die Expansion durchgeführt wird.
- ex wird zyklisch weitergesetzt, sobald sich die Primärdatei verdoppelt hat.
- Für ex wählen wir Ψ_j , so daß g_j eine OLH mit partiellen Erweiterungen ist, für alle anderen Adressen $j \in \{1, \dots, k\} - \{ex\}$ wählen wir Ψ_j so, daß g_j eine OLH ist.
- Die Seite, die als nächstes expandiert wird, wird durch k *Expansionszeiger* ep_1, \dots, ep_k bestimmt.
- n_0 ist die Zahl der partiellen Expansionen bis zu einer Verdoppelung der Primärdatei. Im weiteren gehen wir von $n_0 = 2$ aus.



Organisation von partiellen Expansionen

1. partielle Expansion

Während der ersten partiellen Expansion betrachten wir Paare von Seiten:

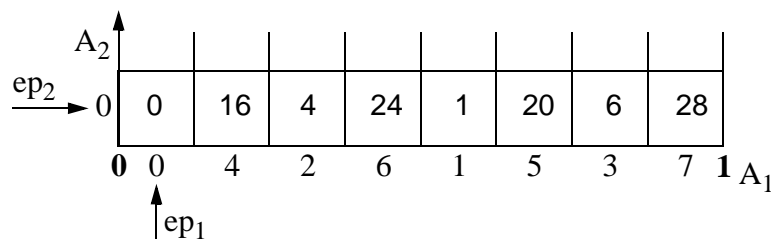
$$\left(G(ep_1, \dots, ep_{ex}, \dots, ep_k), \right. \\ \left. G(ep_1, \dots, ep_{ex} + 2^{L_{ex}-1}, \dots, ep_k) \right)$$

mit $0 \leq ep_j < 2^{L_j}$ für $j \neq ex$ und $0 \leq ep_{ex} < 2^{L_{ex}-1}$.

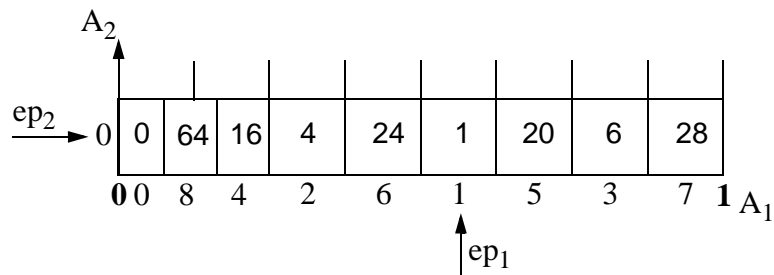
Diese Paare werden jeweils um eine Seite erweitert.

Beispiel:

$$k = 2, ex = 1, L_1 = L_2 = 3, m_1 = m_2 = 8; ep_1 = 0, ep_2 = 0.$$



Zunächst wird das Paar (0,4) mit $G(0,0) = 0$ und $G(4,0) = 16$ um die Seite $G(8,0) = 64$ erweitert und ep_1 um 1 erhöht:



Nachfolgend werden die Paare (1,5), (2,6) und (3,7) expandiert.

Allgemeines Vorgehen:

Verlangt die Kontrollfunktion eine Expansion um eine Seite, werden die Schlüssel vom Paar $(G(ep_1, \dots, ep_{ex}, \dots, ep_k), G(ep_1, \dots, ep_{ex} + 2^{L_{ex}-1}, \dots, ep_k))$ auf 3 Seiten wie folgt verteilt:

- Betrachte die binäre Darstellung von $ep_{ex} = \sum_{i>0} b_i \cdot 2^{i-1}$
- Berechne den Anfangspunkt a des Seitenpaares:

$$a = \frac{\sum_{i=1}^{L_{ex}-1} b_{L_{ex}-i} \cdot 2^{i-1}}{2^{L_{ex}-1}}$$

- Zerlege den Bereich des Seitenpaares in 3 halboffene Intervalle I_a^1, I_a^2 und I_a^3 wie folgt:

$$I_a^j = \left[a + \frac{j-1}{3 \cdot 2^{L_{ex}-1}}, a + \frac{j}{3 \cdot 2^{L_{ex}-1}} \right), \quad j = 1, 2, 3$$

- Nun wird der Schlüssel $x = (x_1, \dots, x_{ex}, \dots, x_k)$ in Seite p abgespeichert mit

$$p = \begin{cases} G(ep_1, \dots, ep_{ex}, \dots, ep_k), & \text{falls } x_{ex} \in I_a^1 \\ G(ep_1, \dots, ep_{ex} + 2^{L_{ex}}, \dots, ep_k), & \text{falls } x_{ex} \in I_a^2 \\ G(ep_1, \dots, ep_{ex} + 2^{L_{ex}-1}, \dots, ep_k), & \text{falls } x_{ex} \in I_a^3 \end{cases}$$

Somit werden die Schlüssel in der neu geschaffenen Seite abgespeichert, wenn die ex -Komponente im 2. Intervall liegt.

Danach werden die Expansionszeiger so weitersetzt, daß sie auf die als nächstes zu expandierende Gruppe von Seiten zeigen.

Beispiel: (wie oben)

$$ep_1 = ep_{ex} = 0, ep_2 = 0.$$

$$\Rightarrow a = 0$$

$$\Rightarrow I_a^1 = \left[0, \frac{1}{12} \right), \quad I_a^2 = \left[\frac{1}{12}, \frac{2}{12} \right), \quad I_a^3 = \left[\frac{2}{12}, \frac{3}{12} = \frac{1}{4} \right)$$

Weitersetzen von ep_{ex} auf $ep_{ex} = 1$ (nächstes betrachtetes Seitenpaar (1,5)).

2. partielle Expansion

Während der zweiten partiellen Expansion betrachten wir Tripel von Seiten:

$$\begin{aligned} & (G (ep_1 , \dots , ep_{ex} , \dots , ep_k), \\ & G (ep_1 , \dots , ep_{ex} + 2^{L_{ex}-1} , \dots , ep_k), \\ & G (ep_1 , \dots , ep_{ex} + 2^{L_{ex}} , \dots , ep_k)) \end{aligned}$$

mit $0 \leq ep_j < 2^{L_j}$ für $j \neq ex$ und $0 \leq ep_{ex} < 2^{L_{ex}-1}$.

Ansonsten verläuft die 2. partielle Expansion analog zur 1. partiellen Expansion ab.

Exact Match Query

Gegeben sei eine Exact Match Query (x_1, \dots, x_k) während der 1. partiellen Expansion der Datei.

Vorgehensweise:

- Berechne den *Pseudoschlüssel* $\Psi_j(x_j) = (b_1, \dots, b_w)$ mit $w \geq L_j + 1$ für jede Dimension j .
- Für jedes j , $1 \leq j \leq k$, berechne die *vorläufige Komponentenadresse* i_j :

$$i_j = \sum_{r=1}^{L_j} b_r \cdot 2^{r-1}$$

- Mit Hilfe der Expansionszeiger ep_1, \dots, ep_k kann nun entschieden werden, ob die Seite mit Adresse $G(i_1, \dots, i_k)$ schon expandiert wurde.

Fall 1: die Seite mit Adresse $G(i_1, \dots, i_k)$ wurde noch nicht expandiert

- greife auf diese Seite zu.

Fall 2: die Seite mit Adresse $G(i_1, \dots, i_k)$ wurde bereits expandiert

- i_{ex} muß neu bestimmt werden:

- Betrachte die binäre Darstellung von $i_{ex} = \sum_{r=1}^{L_{ex}-1} b_r \cdot 2^{r-1}$.

- Berechne den Anfangspunkt a des Seitenpaares, in dem i_{ex} liegt:

$$a = \frac{\sum_{i=1}^{L_{ex}-1} b_{L_{ex}-i} \cdot 2^{i-1}}{2^{L_{ex}-1}}$$

- Für dieses a berechne nun

$$i_{ex} = \begin{cases} i_{ex} & , \text{ falls } x_{ex} \in I_a^1 \\ i_{ex} + 2^{L_{ex}} & , \text{ falls } x_{ex} \in I_a^2 \\ i_{ex} + 2^{L_{ex}-1} & , \text{ falls } x_{ex} \in I_a^3 \end{cases}$$

- Greife nun zur Seite mit Adresse $G(i_1, \dots, i_{ex}, \dots, i_k)$ zu.

Leistungsverhalten

- Bei Gleichverteilung anderen Verfahren überlegen.
- Der relative Belegungsfaktor aller Ketten $bf(K_i)$ erfüllt bei Gleichverteilung die Bedingung:

$$\frac{1}{2} \leq bf(K_i) \leq 1$$

- Beispiel für eine Untersuchung:
 - 2 dimensionale gleichverteilte Schlüssel
 - Schwellwert $\alpha = 85\%$
 - Kapazität einer Primärseite = 31
 - Kapazität einer Sekundärseite = 7
 - Zahl der Datensätze zwischen 4200 und 16800

⇒ durchschnittliche Anzahl von Seitenzugriffen bei erfolgreichen Exact Match Queries: 1,095

Das Quantilverfahren [KS 87]

Anpassung an Ungleichverteilungen

Problem: Der Partitionierungsprozeß von multidimensionalem ordnungserhaltenden linearen Hashing (MOLHPE) paßt sich nicht gut genug an die zugrundeliegende Verteilung der Schlüssel an.

Grund: Der Abstand zwischen den Partitionierungslinien wird nicht durch die Datenverteilung beeinflusst.

Folge: MOLHPE verarbeitet zwar gleichverteilte Daten ausgezeichnet, verliert aber bei Ungleichverteilungen an Leistungsfähigkeit.

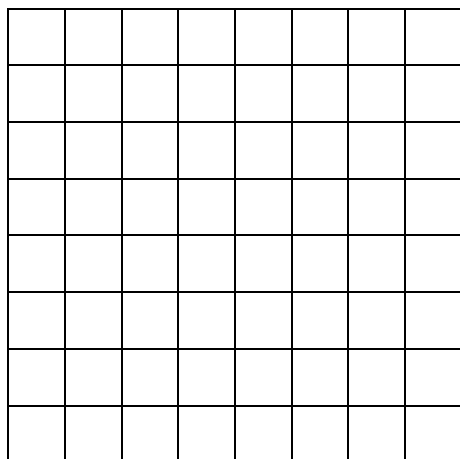
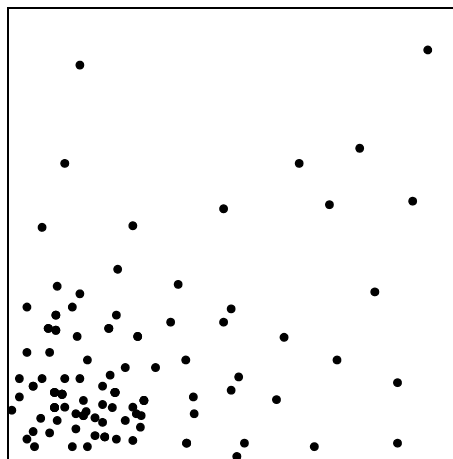
Ziel:

Entwicklung eines Verfahrens, das für jede Kette K_i $\frac{1}{2} \leq bf(K_i) \leq 1$ auch für Nichtgleichverteilung garantiert, indem es die Partitionierungslinien in Abhängigkeit von der Verteilung der Schlüssel wählt.

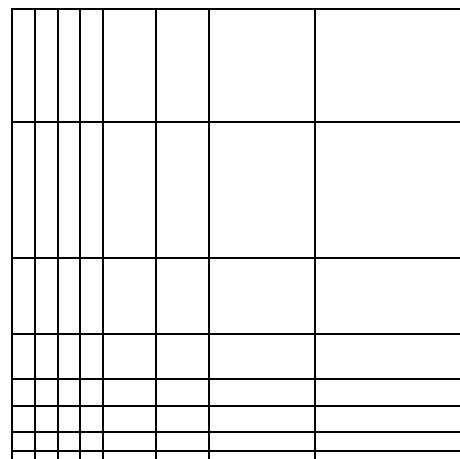
Idee (Quantilverfahren)

- Wir gehen von einer unabhängigen, im voraus unbekanntem Verteilung der Daten aus.
- Damit können wir die Verteilung der Daten für jedes Attribut separat betrachten.
- Wir nähern uns der unbekanntem Verteilung an, indem wir die vorhandenen Schlüssel als Schätzwert verwenden.

“Links-unten”-Verteilung der Daten:



MOLHPE



MOLHPE mit Quantilverfahren

Gesichtspunkte:

- die Schätzung der Datenverteilung
- die Anwendung des Quantilverfahrens auf das multidimensionale lineare Hashing mit partiellen Erweiterungen
- die Reorganisation der Partitionierung bei Veränderung der Verteilung

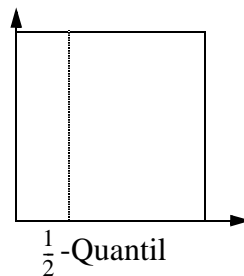
Schätzung der Datenverteilung (informell)*Grundidee:*

Wir nähern uns der unbekanntem Verteilung an, indem wir die vorhandenen Schlüssel als Schätzwert verwenden.

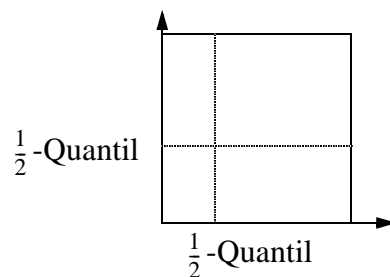
Beispiel:

- Wir beginnen mit dem leeren Datenraum.
- Nach Einfügen von Datensätzen ist der Datenraum aufzuteilen:
Wir wählen die x-Achse.

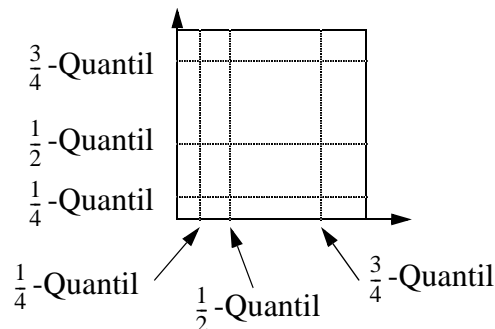
Da die Daten ungleich verteilt sind, legen wir die Partitionierungslinien nicht in die Mitte, sondern so, daß die Datensätze “halbe-halbe” aufgeteilt werden, d.h. das $\frac{1}{2}$ -Quantil ist der Partitionierungspunkt.



- Weiteres Einfügen macht eine weitere Partionierung notwendig.
Nun wählen wir die y-Achse mit dem entsprechenden $\frac{1}{2}$ -Quantil:



- Weitere Datensätze machen zusätzliche Partionierungen bei den $\frac{1}{4}$ - und $\frac{3}{4}$ -Quantilen notwendig:



Schätzung der Datenverteilung (formell)*Begriffe:*

- F k -dimensionale Verteilungsfunktion der Schlüssel
 - $F_i, 1 \leq i \leq k$ 1 -dimensionale Verteilungsfunktion der Randverteilung des i -ten Attributs
- Da wir annehmen, daß F in den Dimensionen unabhängig ist, d.h.
- $$F(x_1, \dots, x_k) = F_1(x_1) * \dots * F_k(x_k),$$
- betrachten wir nur die F_i .
- α -Quantil für $0 \leq \alpha \leq 1, 1 \leq i \leq k$, ist das α -Quantil des i -ten Attributs der Attributswert $x(\alpha) \in A_i$ mit $F_i(x(\alpha)) = \alpha$.

Idee:

Schätzung der jeweiligen α -Quantile auf Basis der bisherigen Daten mit Hilfe einer **stochastischen Approximationsmethode**.

Schätzung der α -Quantile

$x(\alpha)$ ist gesucht.

- $x_1(\alpha)$ ein erster Schätzwert für $x(\alpha)$
- $\{x_i^j\}$ die Folge der n bisher abgespeicherten Werte des i -ten Attributs
- l die Anzahl der bisherigen Schätzwerte
- $m(n, l) := |\{x_i^j \mid x_i(\alpha) \geq x_i^j, 1 \leq j \leq n\}|$
ist die Anzahl der Datensätze, deren i -ter Attributswert unterhalb des l -ten Schätzwertes von $x(\alpha)$ liegt.

Dann läßt sich ein $(l+1)$ -ter Schätzwert wie folgt berechnen:

$$(*) \quad x_{l+1}(\alpha) = x_l(\alpha) - \frac{1}{l} \left(\frac{m(n, l)}{n} - \alpha \right)$$

Für $l \rightarrow \infty$ konvergiert $x_l(\alpha) \rightarrow x(\alpha)$ mit Wahrscheinlichkeit 1 unter sehr allgemeinen Voraussetzungen an F_i (wenn F_i fast überall stetig ist) [RM 55].

Anwendung der Quantilmethode auf MOLHPE

Um das Quantilverfahren auf MOLHPE anwenden zu können, benötigen wir eine Abbildung der Quantil-Partitionierung auf die Partitionierung von MOLHPE.

Partitionierungspunkte

Wir betrachten die Lage der Partitionierungslinien gemäß dem Quantilverfahren:

- Für jede Achse j , $1 \leq j \leq k$, gibt es eine Menge P_j von *Partitionierungspunkten* $pp(\alpha)$, die den neuesten Schätzwert für das α -Quantil gemäß der Quantilmethode bilden ($0 < \alpha < 1$).
- Es gibt $m_j - 1$ Partitionierungspunkte für die Achse j .
Wird die Achse j nicht expandiert, ist $m_j = 2^{L_{ex}}$.
- Die α -Werte können durch Bitstrings (b_1, \dots, b_w) der Länge L_j bzw. $L_j + 1$ dargestellt werden:

$$(**) \quad \alpha = \sum_{i=1}^w b_i \cdot 2^{-i}$$

- Die Menge P_j der Partitionierungspunkte der j -ten Achse ist damit gegeben durch:

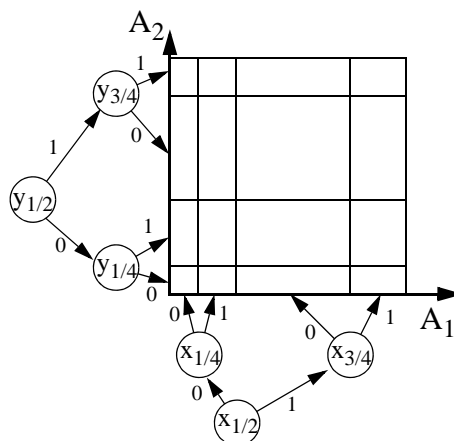
$$P_j = \{pp(\alpha) \mid \alpha = \sum_{i=1}^w b_i \cdot 2^{-i} \quad b_i \in \{0, 1\}\}$$

Organisation der Partitionierungspunkte

- Für jede Achse j organisieren wir P_j in einem binären Suchbaum.
- In diesem Baum wird jedem $pp(\alpha)$ gemäß (**) der Bitstring (b_1, \dots, b_w) von α zugeordnet.
- Dieser Bitstring kann durch Umkehrung (b_w, \dots, b_1) als Komponentenadresse aufgefaßt werden.
- Bei jeglicher Art von Anfrage wird nun für jedes Attribut A_j , $1 \leq j \leq k$, ein nichtgleichverteilter Attributswert $x_j \in \text{Domain}(A_j)$ durch Suchen im binären Baum in ein gleichverteiltes $\alpha \in [0, 1)$ umgewandelt.

Mit diesem gleichverteilten $\alpha \in [0, 1)$ geht man in die entsprechenden Anfragealgorithmen für multidimensionales lineares Hashing mit partiellen Erweiterungen.

- Um neue Schätzwerte für die α -Quantile berechnen zu können, ist im Suchbaum der j -ten Achse außerdem gespeichert, wieviele der bisherigen Schlüssel mit ihrem j -ten Attributswert zwischen zwei Partitionierungspunkte fallen.



Reorganisation der Datei

Im Laufe der Zeit kann sich die Verteilung der Daten im Raum verändern:

⇒ Eine Anpassung der Partitionierung des Datenraums an die momentan gültige Verteilung wird notwendig.

Ablauf der Reorganisation

- Berechnung eines neuen Schätzwertes für einen Partitionierungspunkt $pp(\alpha)$ nach der Gleichung (*).
- Im Bedarfsfall wird eine Partitionierungslinie (bzw. -hyperebene für $k > 2$) verschoben:

- **lokale Reorganisation**

Um ein gleichmäßiges Verhalten der Speicherungsstruktur zu erreichen, wird die Partitionierungsachse nicht in einem Schritt, sondern in mehreren Schritten verschoben.

Eine lokale Reorganisation ist dadurch definiert, daß der Erwartungswert für die Anzahl der Seitenzugriffe während einer lokalen Reorganisation konstant ist.

- *Reorganisationsachse* $ra \in \{1, \dots, k\}$ ist die Achse, deren Partitionierungspunkte als nächstes reorganisiert werden.

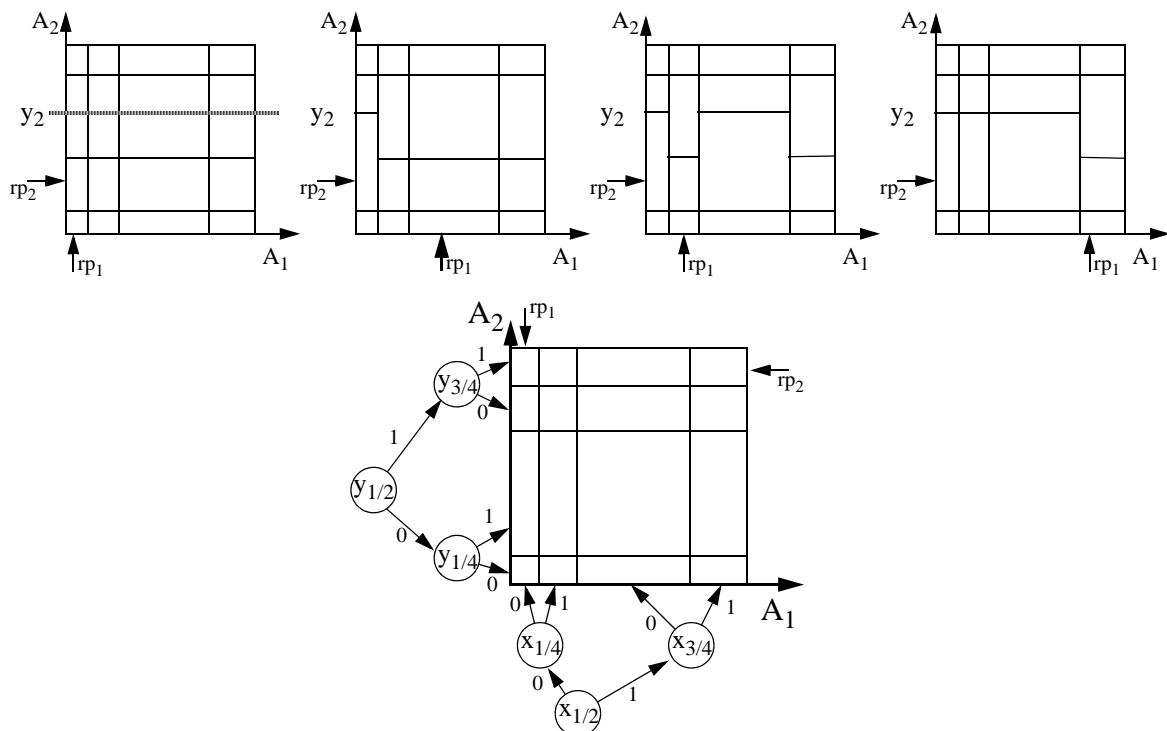
- *Reorganisationszeiger* rp_1, \dots, rp_k bestimmen die Seite, die als nächstens reorganisiert wird:

Während eines Reorganisationsschrittes wird die Seite $G(rp_1, \dots, rp_k)$ und ihre rechte Nachbarseite bezüglich der Achse ra und des neuen gemeinsamen Partitionierungspunktes angepaßt.

- Ein Reorganisationsschritt wird nach jeder Einfügung eines Datensatzes, die keine Expansion erfordert, durchgeführt.

Rechtfertigung:

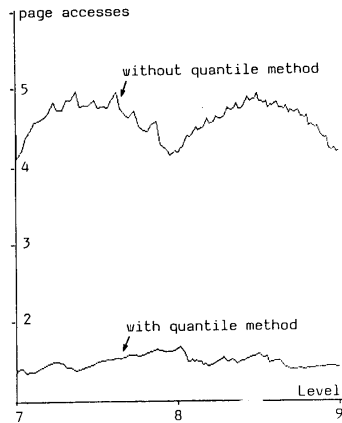
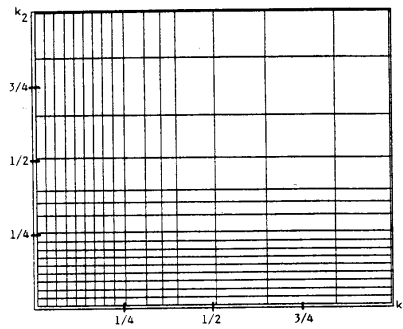
Für die meisten Nichtgleichverteilungen ist die durchschnittliche Anzahl von Seitenzugriffen bei einer Einfügung mit Quantilverfahren um ein Mehrfaches niedriger als ohne Quantilverfahren.



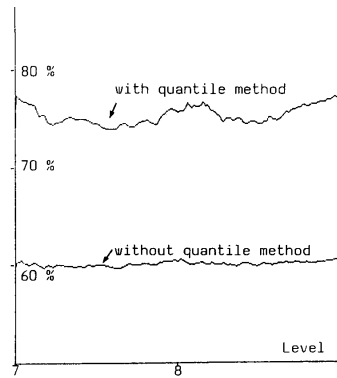
Leistung

1. Experiment: "links-unten" Verteilung

Partitionierung:



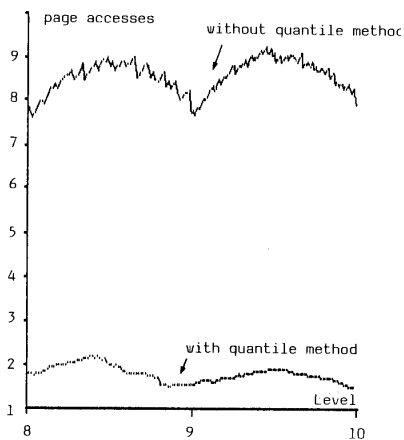
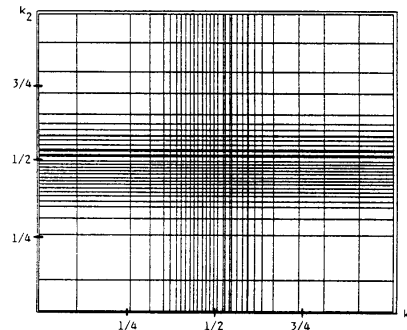
Durchschnittl. Zahl von Seitenzugriffen bei erfolgreichen Exact Match Queries in Abhängigkeit vom Level L.



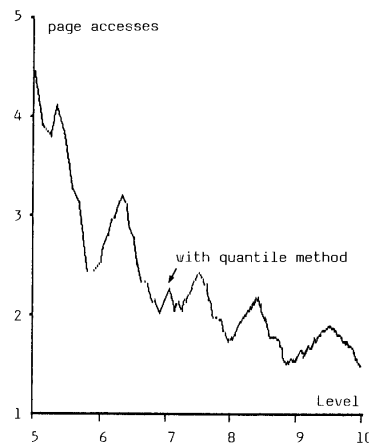
Durchschnittl. Speicherplatzausnutzung in Abhängigkeit vom Level L.

2. Experiment: Normalverteilung

Partitionierung:



Durchschnittl. Zahl von Seitenzugriffen bei erfolgreichen Exact Match Queries in Abhängigkeit vom Level L.



Durchschnittl. Zahl von Seitenzugriffen bei erfolgreichen Exact Match Queries in Abhängigkeit vom Level L.

5.3 Literatur

Eine Übersicht über multidimensionale Hashverfahren findet man in

- [Wid 91] Widmayer P.: *'Datenstrukturen für Geodatenbanksysteme'*, in: Vossen G., Witt K.-U. (Hrsg.): *'Entwicklungstendenzen bei Datenbank-Systemen'*, Oldenbourg, 1991, pp. 317-361.
- [See 89] Seeger B.: *'Entwurf und Implementierung mehrdimensionaler Zugriffsstrukturen'*, Dissertation, Universität Bremen, 1989.

Originalliteratur (Indexstrukturen)

- [Hin 85] Hinrichs K.: *'The grid file system: implementation and case studies for applications'*, Dissertation No. 7734, Eidgenössische Technische Hochschule (ETH) Zürich, 1985.
- [KW 85] Krishnamurthy R., Whang K.-Y.: *'Multilevel Grid Files'*, IBM Research Center Report, Yorktown Heights, N.Y., 1985.
- [KS 86] Kriegel H.-P., Seeger B.: *'Multidimensional Order Preserving Linear Hashing with Partial Expansions'*, Proc. Int. Conf. on Database Theory, Rome, Italy, 1986, in: Lecture Notes in Computer Science, Vol. 243, Springer, pp. 203-220.
- [KS 87] Kriegel H.-P., Seeger B.: *'Multidimensional Dynamic Quantile Hashing is very Efficient for Non-Uniform Record Distributions'*, Proc. 3rd Int. Conf. on Data Engineering, Los Angeles, CA., 1987, pp. 10-17, in: Information Science, Vol. 48, 1989, pp. 99-117.
- [KS 88] Kriegel H.-P., Seeger B.: *'PLOP-Hashing: A Grid File without Directory'*, Proc. 4th Int. Conf. on Data Engineering, Los Angeles, CA., 1988, pp. 369-376.
- [NHS 84] Nievergelt J., Hinterberger H., Sevcik K. C.: *'The Grid File: An Adaptable, Symmetric Multikey File Structure'*, ACM Trans. on Database Systems, Vol. 9, No. 1, 1984, pp. 38-71.
- [Oto 84] Otoo E. J.: *'A Mapping Function for the Directory of a Multidimensional Extendible Hashing'*, Proc. 10th Int. Conf. on Very Large Databases, Singapore, 1984, pp. 493-506.
- [Ouk 85] Ouksel M.: *'The Interpolation Based Grid File'*, Proc. 4th ACM SIGACT/SIGMOD Symp. on Principles of Database Systems, 1985, pp. 20-27.

Literatur (Leistungsuntersuchungen)

- [Chr 81] Christodoulakis S.: Dissertation, University of Toronto, 1981.
- [Kri 84] Kriegel H.-P.: *'Performance Comparison of Index Structures for Multi-Key Retrieval'*, Proc. ACM SIGMOD Int. Conf. on Management of Data, Boston, MA., 1984, pp. 186-196.
- [KSS89] Kriegel H.-P., Schiwietz M., Schneider R., Seeger B.: *'Performance Comparison of Point and Spatial Access Methods'*, Proc. 1st Symp. on the Design and Implementation of Large Spatial Databases, Santa Barbara, CA., 1989, in: Lecture Notes in Computer Science, Vol. 409, Springer, 1990, pp. 89-114.
- [Reg 85] Regnier M.: *'An Analysis of grid file algorithms'*, BIT 25, 1985, pp. 335-357.

Literatur (Nichtstandard-Datenbanksysteme)

- [HR 85] Härder T., Reuter A.: '*Architektur von Datenbanksystemen für Non-Standard-Anwendungen*', GI-Fachtagung Datenbanksysteme für Büro, Technik und Wissenschaft (BTW), Informatik-Fachberichte 94, Springer, 1985, pp. 253-286.
- [Küs 86] Küspert K.: '*Non-Standard-Datenbanksysteme*', Informatik-Spektrum, Vol. 9, No. 3, 1986, pp. 184-185.

Quellenverweis:

- [RM 55] Robbins H., Monro S.: '*A Stochastic Approximation Method*', Annals of Mathematical Statistics, Vol. 22, 1955, pp. 400-407.

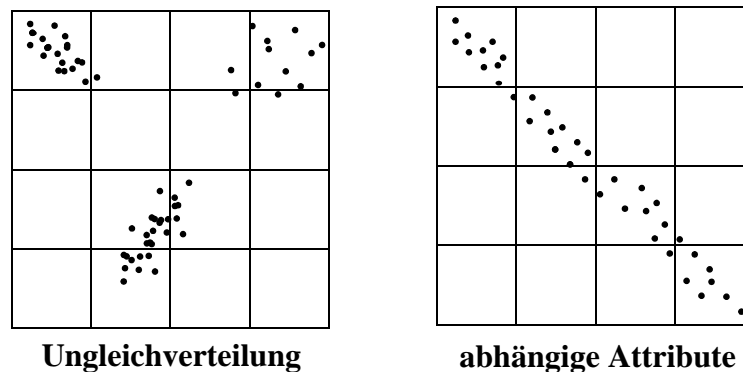
6 Suchstrukturen für multidimensionale Punktdaten

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:



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.
⇒ 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.
⇒ 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:

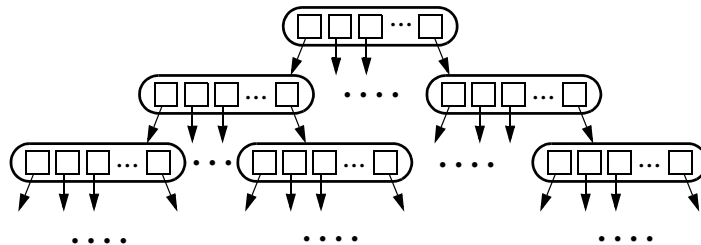
Die Dimensionen werden nicht gleichberechtigt behandelt.

Gesucht:

- Eine Partitionierungsstrategie, die
 - die Dimensionen gleichberechtigt behandelt,
 - die sich an die gegebene Datenverteilung entsprechend anpasst (gleichmäßige Speicherplatzausnutzung) 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* ($S_i \cap S_j = \emptyset$ für alle $1 \leq i, j \leq m, i \neq j$)

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 Quadrees

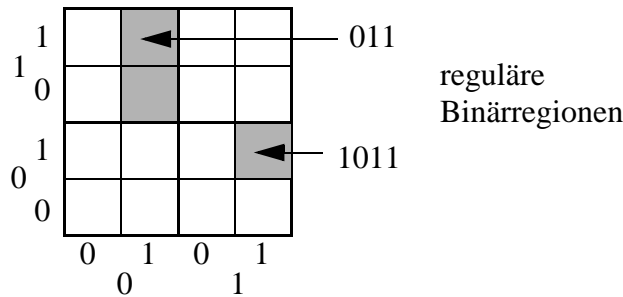
Grundidee:

- 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.

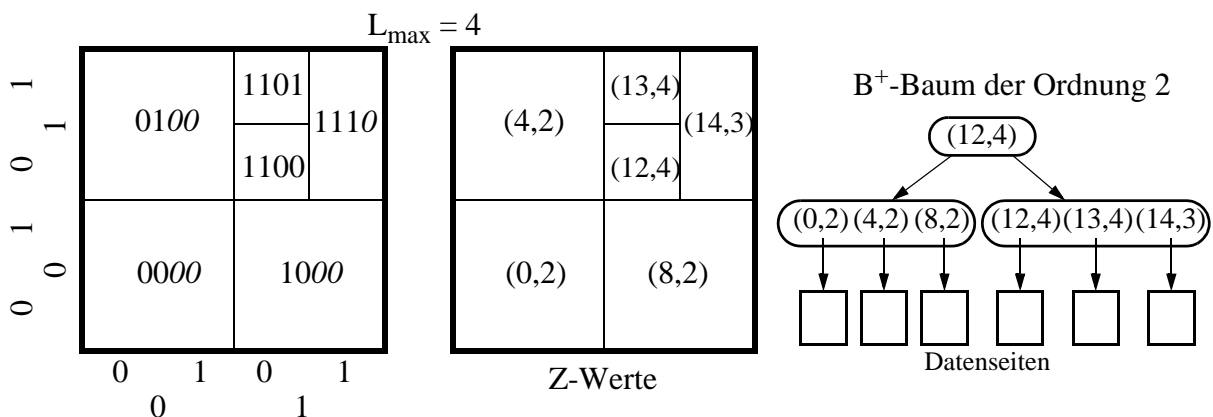
Beispiel:



- **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.

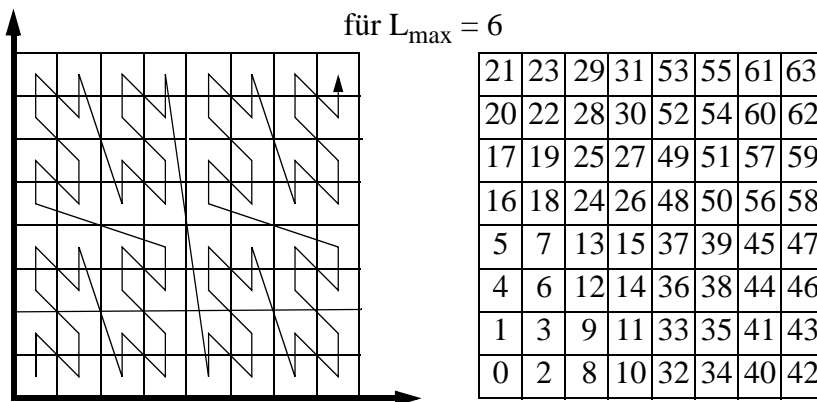


• **Z-Ordnung**

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.



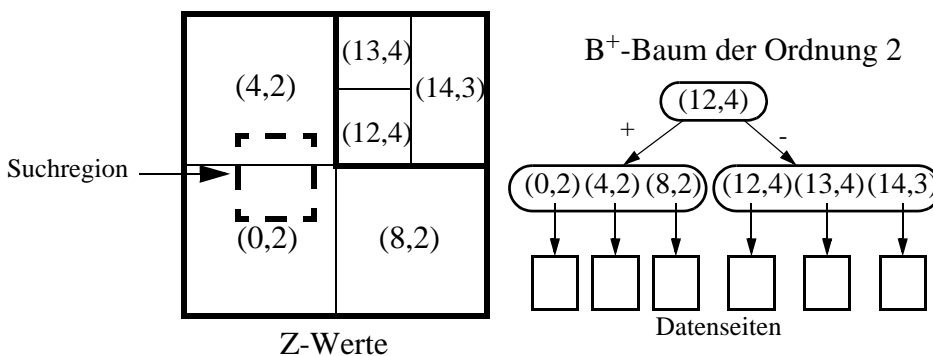
Anfragen

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]

- Quadrees 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.

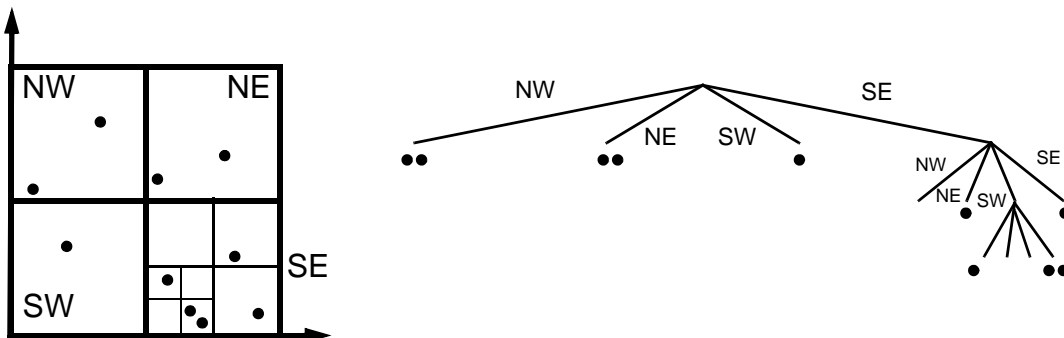
- Die Partitionierung terminiert, falls ein spezifisches *Abbruchkriterium* erfüllt ist.

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:

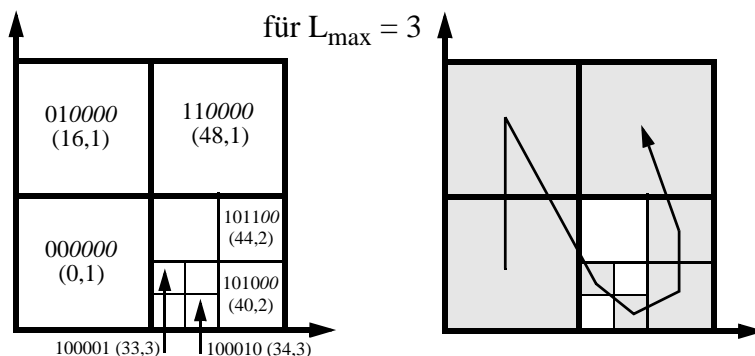


Partitionierung des PR-Quadrees

Abbruchkriterium: max. 2 Punkte pro Zelle

Die nicht-leeren Zellen des Quadrees 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 Quadrees 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.

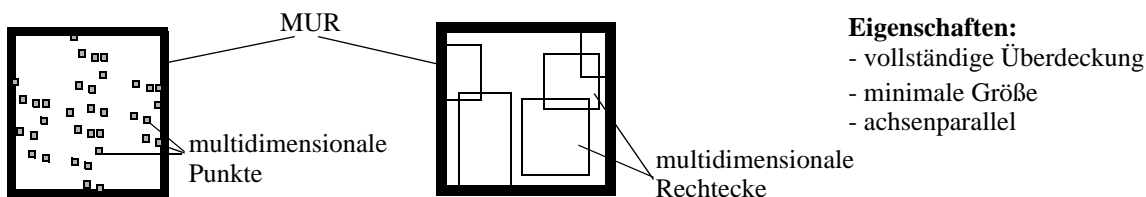
Idee

- 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)



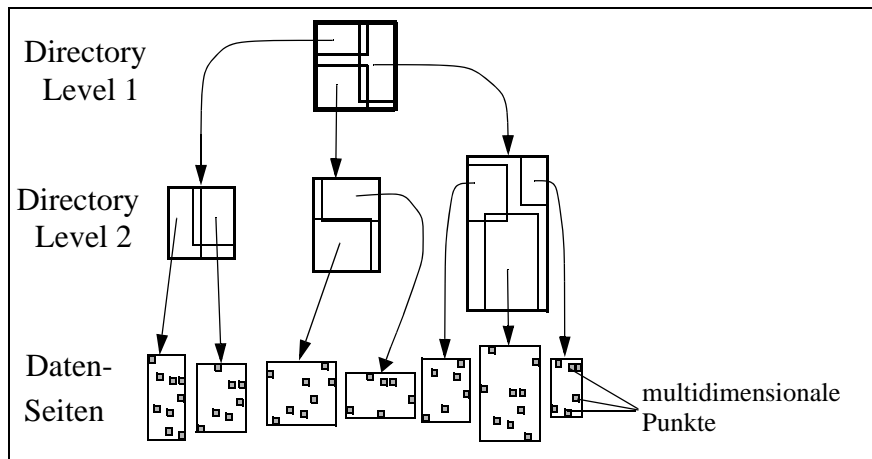
Eigenschaften:

- vollständige Überdeckung
- minimale Größe
- achsenparallel

- Ein Eintrag in einer Datensite 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 Datensiten, 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:

```

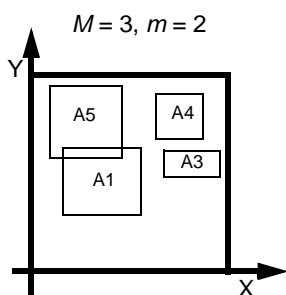
Window Query (Page, Window);
  FOR ALL Entry ∈ Page DO
    IF Window INTERSECTS Entry.Rectangle THEN
      IF Page = DataPage THEN
        Write (Entry)
      ELSE
        WindowQuery (Entry.Subtree^, Window);

```

- 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.

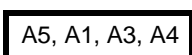
Optimierungsziele

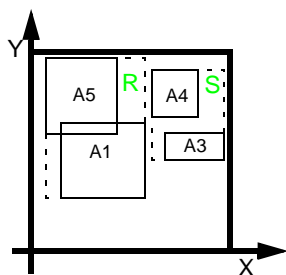
- geringe Überlappung der Seitenregionen
- Seitenregionen mit geringem Flächeninhalt
⇒ geringe Überdeckung von totem Raum
- Seitenregionen mit geringem Umfang



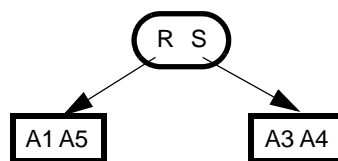
Start:  leere Datenseite (= Wurzel)

Einfügen von: A5, A1, A3, A4

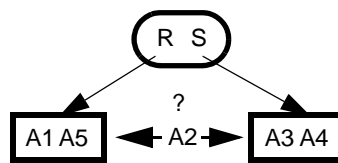
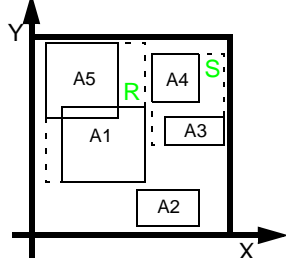
 * (Überlauf)



⇒ Split in 2 Seiten



Frage: Wie wird aufgeteilt? (Splitstrategie)



Frage: Wo wird eingefügt? (Einfügestrategie)

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, \dots, 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 ?

⇒ Wir benötigen Heuristiken, die die Aufteilung vornehmen, → 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 Directory-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.

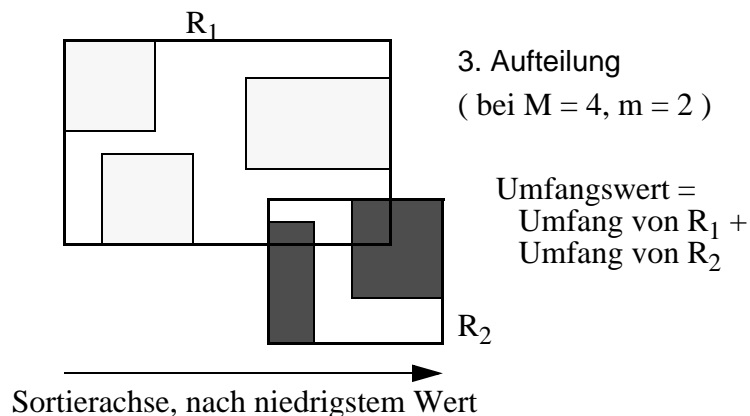
Split von Seiten

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 Überdeckung 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.

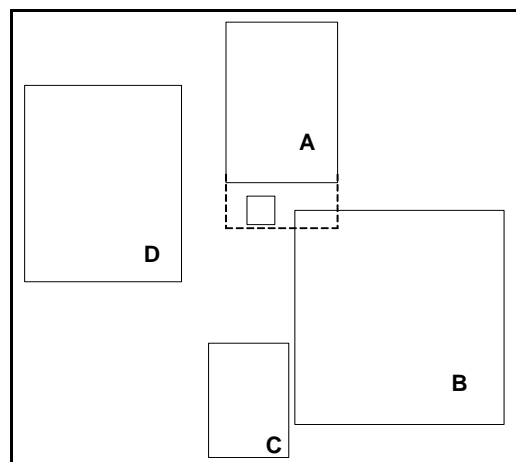
Our Engineering-Type Approach

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.

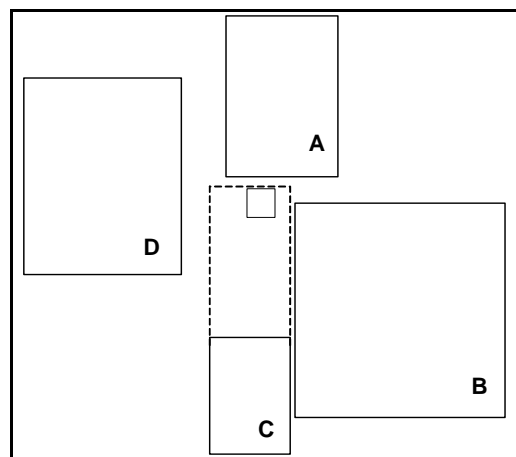
ChooseSubtree Algorithm:

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

R-tree



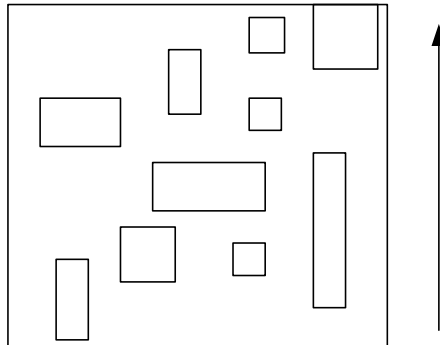
The R*-tree minimizes the overlap
increase of the directory rectangles
pointing to data rectangles
(on the lowest level of the directory).



R*-tree Split:

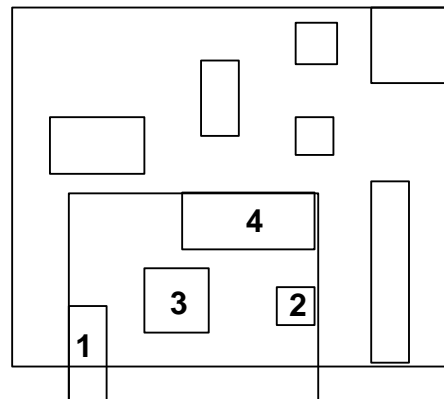
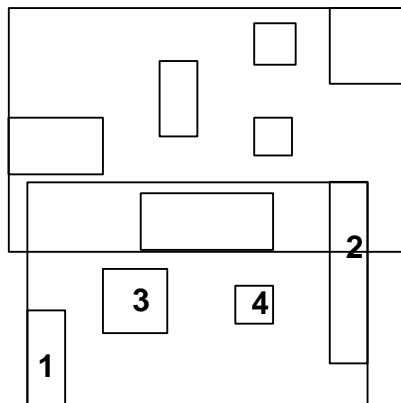
Determine split axis:

- 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:

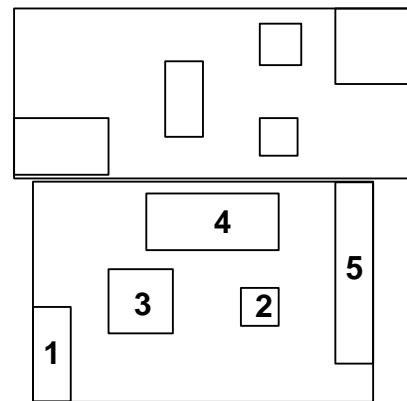
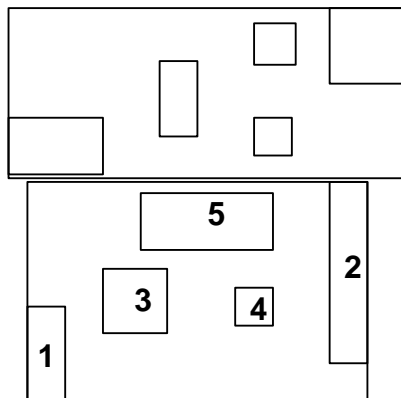


sorted by low values

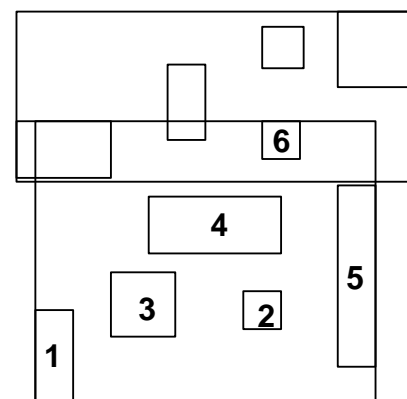
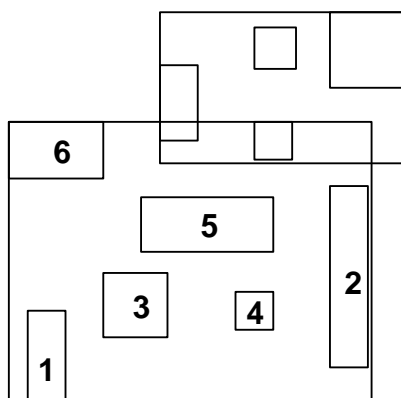
sorted by high values



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:

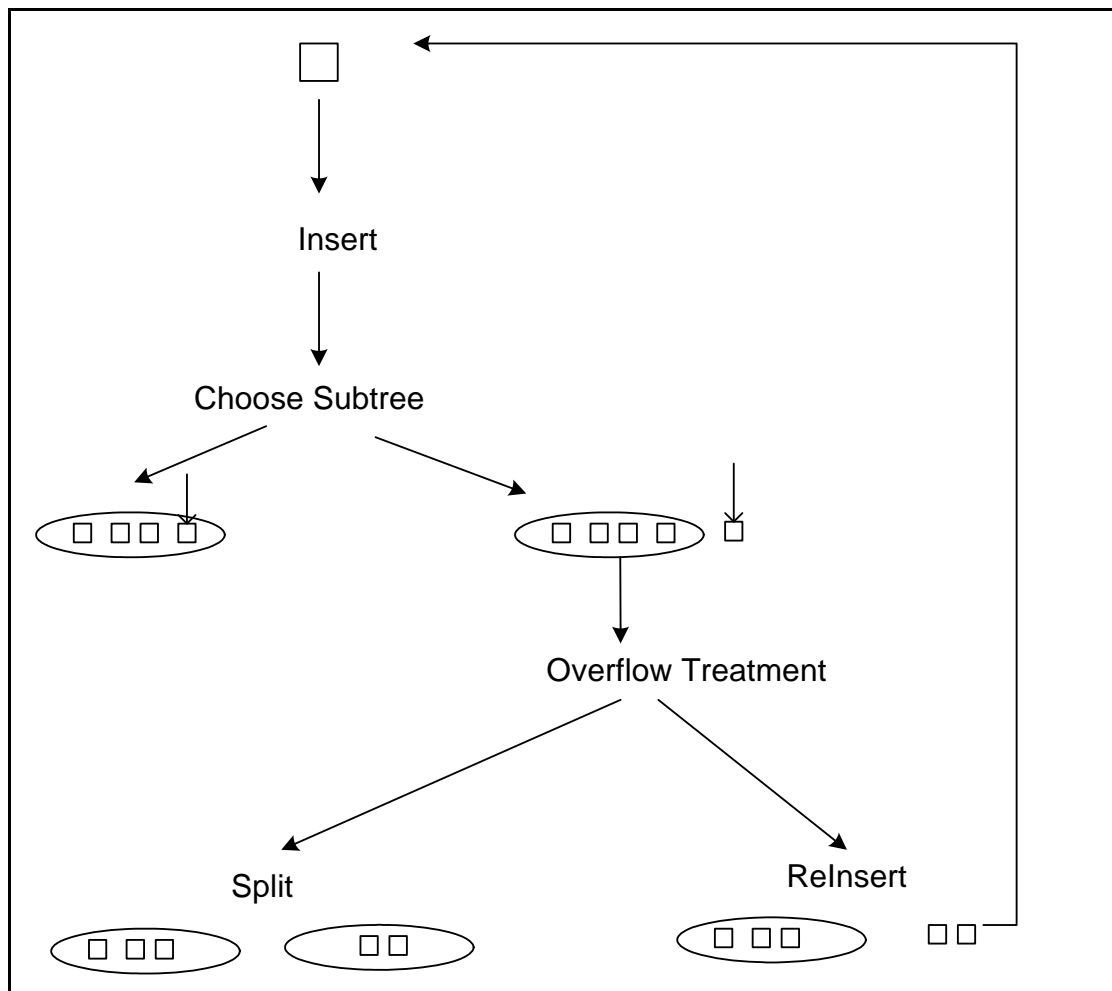
The retrieval performance of the R-tree may suffer from data rectangles inserted during the early growth of the tree.

Improvement:

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

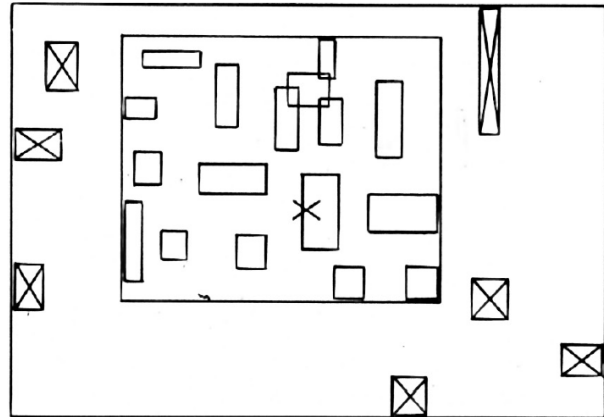
Insertion Algorithm for the R*-tree:

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

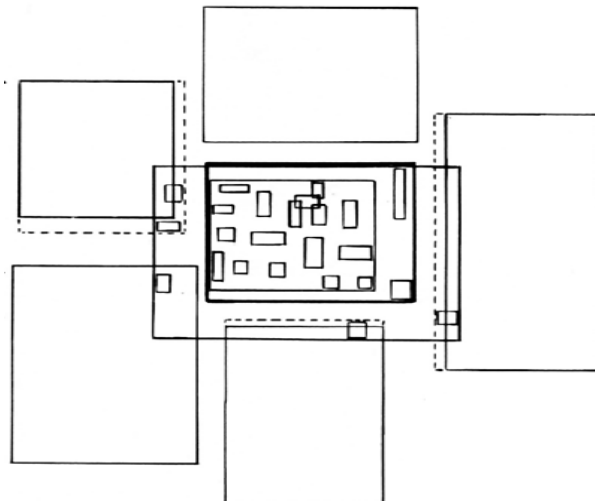


Algorithm ReInsert:

- (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:

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

With 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 *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 *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 herkömmlichen 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 Berechnung 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 Datenbank-Objekten, denen eine Routing Rolle zugewiesen wurde
 - Wenn ein Knoten überläuft und geplittet werden muß, vergibt der Splitalgorithmus 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

