9 Indexstrukturen für hochdimensionale Räume

Anfrageleistung von Indexstrukturen verschlechtert sich mit zunehmender Dimension
"Curse of Dimensionality"
→ Häufig haben scanbasierte Methoden bessere Anfrage-Performanz als z.B. R*-Bäume

9.1 X-Baum (eXtended node tree) [BKK 96]

Idee

- Weiterentwicklung des R*-Baums für hochdimensionale Räume
- der X-Baum vermindert Überlappung im Directory mittels
  - einem Split mit minimaler Überlappung (bei Punktdaten existiert sogar ein überlappungsfreier Split)
  - dem Superknoten Konzept

Eigenschaften

- Überlappung von Directoryknoten wird soweit möglich vermieden
- der X-Baum ist ein Hybrid aus hierarchischer und linearer Struktur
  - eine hierarchische Organisation ist günstig für niedrigdimensionale Daten ($D \leq 5$)
  - eine lineare Organisation ist günstig für hochdimensionale Daten ($D \geq 32$)

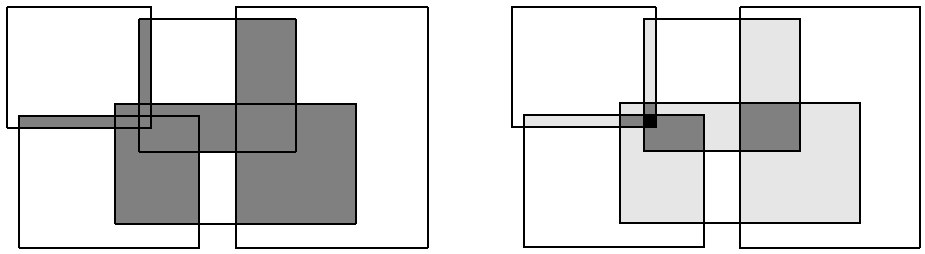
  → der X-Baum erreicht durch dynamische Anpassung eine bestmögliche Kombination für jede Dimensionalität

Definition: *Überlappung*

Die Überlappung eines R-Baum Knotens ist der Prozentsatz des Raums, der von mehr als einem (Hyper-)Rechteck überdeckt wird.

$$\ddot{U}berlappung = \frac{\left\| \bigcup_{i,j \in \{1\ldots n\}, i \neq j} (R_i \cap R_j) \right\|}{\left\| \bigcup_{i \in \{1\ldots n\}} R_i \right\|}$$

wobei $\{R_1,..R_n\}$ die (Hyper-)Rechtecke des R-Baum Knotens sind und $\|R_x\|$ das Volumen bzw. die Fläche des (Hyper-)Rechtecks R_x bezeichnet.



Überlappung und Multi-Überlappung von 2-dimensionalen Daten

Definition: *Split*

Der Split eines Knotens $S = \{mbr_1, ..., mbr_n\}$ in zwei Unterknoten S_1 und S_2 ist definiert als
$\text{Split}(S) = \{ (S_1, S_2) \mid S = S_1 \cup S_2 \ \wedge \ S_1 \cap S_2 = \varnothing \}$.

Der Split wird bezeichnet mit

(1) überlappungsminimal gdw. $\| MBR(S_1) \cap MBR(S_2) \|$ minimal ist

(2) überlappungsfrei gdw. $\| MBR(S_1) \cap MBR(S_2) \| = 0$

(3) balanciert gdw. $-\varepsilon \leq |S_1| - |S_2| \leq \varepsilon$ (für kleine ε)

Problem

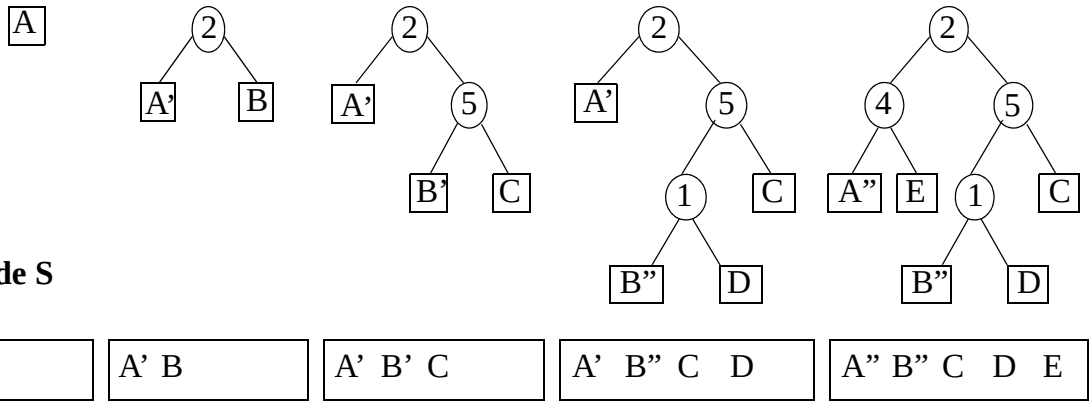
Der Split-Algorithmus von R-Baum und R*-Baum führt zu hoher Überlappung der Directory-Seitenregionen bei hochdimensionalen Räumen

Grund: Es gibt nur wenige (meist eine) geeignete Splitebene bei Split der Directoryseite

- Seiten sind in vielen Dimensionen ungeteilt (Ausdehnung [0..1])
- Benötigt wird eine Dimension, in der alle Kindseiten geteilt wurden

$\Rightarrow$ Konzept hierfür: Split-Tree eines Directory-Knoten

Split Tree



Node S

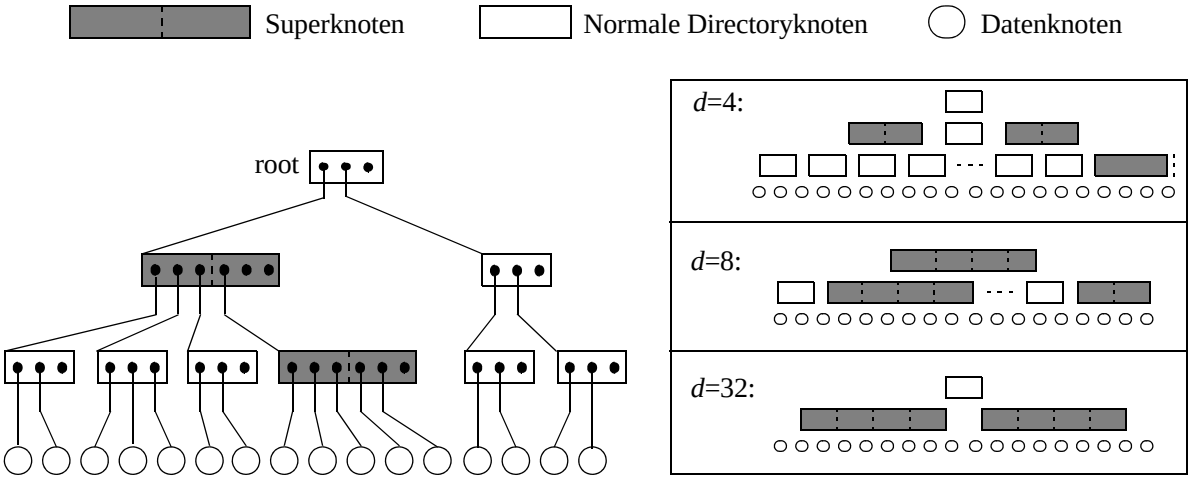
A	A' B	A' B' C	A' B" C D	A" B" C D E

- Idee des Split-Tree:
  - Jede neue Seite entsteht dadurch, daß eine alte Seite aufgeteilt wurde
  - Die Hierarchie von Split-Operationen läßt sich als binärer Baum darstellen, wobei in den inneren Knoten die Split-Dimension vermerkt ist
  - Ist d_i ein Vorgängerknoten eines Blatts X, dann wurde X in Dimension d_i geteilt
- Nur wenn in einem Level des Split-Tree alle Split-Ebenen identisch sind, kann ein Directoryknoten in dieser Dimension überlappungsfrei geteilt werden
  - Dies ist meist nur bei der Wurzel des Splitbaums der Fall
- Beispiel: Directoryknoten wird (fälschlicherweise) in Dimension 1 geteilt
  - Die Seite B" kommt in die 1. Nachfolgeseite
  - Die Seite D kommt in die 2. Nachfolgeseite
  - Die Seiten A", C und E sind in Dimension 1 nicht gesplittet. Werden sie einer der beiden Nachfolgeseiten zugeordnet, so überlappt diese die andere Nachfolgerseite vollständig
- Directoryknoten wird (richtig) in Dimension 2 geteilt
  - A" und E kommen in die erste Nachfolgeseite
  - B", C und D kommen in die 2. Nachfolgeseite
  - Die beiden Nachfolgeseiten sind überlappungsfrei

Weiteres Problem:

Split-Tree kann unbalanciert sein ⇒ unbalancierter Split (z.B. nur eine Kindseite in der einen der Nachfolgeseiten)

⇒ Lösung: Superknoten
Für diesen Fall sieht der X-Baum vor, die Directoryseite gar nicht zu splitten, sondern einen Superknoten mit der doppelten Länge anzulegen



Struktur des X-Baums

Beispiele für X-Bäume mit Daten unterschiedlicher Dimension *d*

- Bei hohen Dimensionen entstehen deshalb zunehmend Superknoten

Vorteile von Superknoten

- Falls große Teile des Directory gelesen werden müssen, ist ein linearer Scan des Superknoten schneller als mehrere wahlfreie Zugriffe
- Die Speicherplatzausnutzung ist höher

  Der zu erwartende Blockfüllungsgrad normaler Directoryknoten liegt genau wie beim R*-Baum bei 66%. Für Superknoten gilt dieser Erwartungswert jedoch nicht, denn Superknoten bestehend aus *n* Blöcken sind in *n*-1 Blöcken komplett und im letzten Block zwischen 0% und 100% gefüllt. Der durchschnittliche Füllungsgrad eines Superknoten der Größe *Y = m * Block-Size* berechnet sich daher wie folgt:

  1.) der Superknoten ist maximal gefüllt, dann belegt er trivialerweise *m* Blöcke

  2.) der Superknoten ist minimal gefüllt, dann belegen die *Y* Bytes an Daten *m*+1 Blöcke

  ⇒ im Durchschnitt ergeben sich also $\frac{m + (m + 1)}{2}$ belegte Blöcke, der durchschnittliche Blockfüllungsgrad ist also $\frac{m}{m + \frac{1}{2}}$

  Bereits ab einer Größe von 3 Blöcken ist der Füllungsgrad mit über 85% weitaus höher als der des R*-Baums.

- Das Directory des X-Baums ist kleiner

  ⇒ größere Teile des Directory können in einem Cachespeicher gepuffert werden

  Extremfall: das Directory besteht nur aus einem einzigen Superknoten (in diesem ist das Directory des X-Baums völlig linear organisiert). Im Vergleich zum korrespondierenden R*-Baum Directory enthält das X-Baum Directory also genau den tiefsten Directory Level des R*-Baums. Daher ist das Directory des X-Baums offensichtlich kleiner als das des korrespondierenden R*-Baums.

Eine Abschätzung für *d*-dimensionale Daten liefert die Formel:

$$DirSize = \frac{DatabaseSize}{BlockSize \cdot StorageUtilization} \cdot 2 \cdot BytesFloat \cdot d$$

Beispiel:*DatabaseSize*: 1 GByte (16-dimensionale Daten)

BlockSize: 4 KBytes

StorageUtilization: 66%

BytesFloat: 4 Bytes pro Float

$\Rightarrow$ *DirSize* = 48 MBytes für einen großen Superknoten

DirSize = 78 MBytes für ein komplett hierarchisches Directory

Operationen

- Insert und Split für Datenknoten: R*-Baum Algorithmus
- Insert und Split für Directoryknoten
  1.) einen topologischen Split anwenden (z.B. den R*-Baum Split)
  2.) falls der topologische Split zuviel Überlappung verursacht, den überlappungsfreien Split benutzen (sofern möglich)
  3.) falls der überlappungsfreie Split nicht balanciert ist, einen Superknoten anlegen

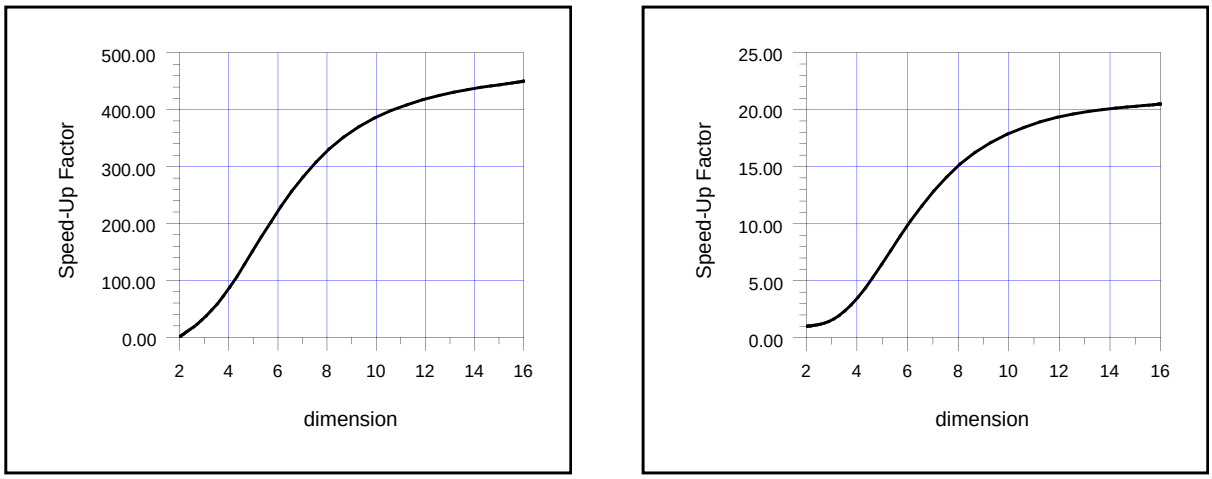
```
bool X_DirectoryNode::split(SET_OF_MBR *in, SET_OF_MBR *out1, SET_OF_MBR *out2)
{
    SET_OF_MBR t1, t2;
    MBR r1, r2;

    // first try topological split, resulting in two sets of MBRs t1 and t2
    topological_split(in, t1, t2);
    r1 = t1->calc_mbr(); r2 = t2->calc_mbr();

    // test for overlap
    if (overlap(r1, r2) > MAX_OVERLAP)
    {
        // topological split fails -> try overlap minimal split
        overlap_minimal_split(in, t1, t2);

        // test for unbalanced nodes
        if (t1->num_of_mbrs() < MIN_FANOUT || t2->num_of_mbrs() < MIN_FANOUT)
              // overlap-minimal split also fails (-> caller has to create supernode)
              return FALSE;
    }

    *out1 = t1; *out2 = t2;
    return TRUE;
}
```

- Query, Delete, und Update: ähnlich wie der R*-Baum Algorithmus, mit Modifikationen zur Berücksichtigung der Superknoten

Nachteile des X-Baums

- Durch Seiten unterschiedlicher Größe entstehen Probleme der Freispeicherverwaltung
- Überlappung entsteht nicht nur bei der Split-Operation sondern z.B. auch beim normalen Einfügen, wenn eine Seitenregion vergrößert werden muß
- Weniger Flexibilität der Struktur, um sich an die Datenverteilung anzupassen
  (meist steht nur eine Splitebene zur Auswahl)
- Konzept der Superknoten wird nur für Directory genutzt, und auch nur um unbalancierten Split zu verhindern.
  $\Rightarrow$ Später: Wähle Blockgröße so, daß Anfragebearbeitung optimiert wird

Experimente



Point Query 10 Nearest-Neighbor Query

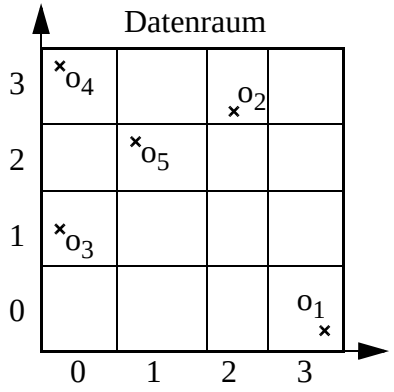
Speed-Up of X-tree over R*-tree on Real Point Data (70 MBytes)

9.2 VA-File (Vector Approximation) [WSB 98]

Keine hierarchische Indexstruktur, sondern eine Variante des sequenziellen Scan (alle Punkte werden betrachtet)

Entwickelt, um Nächste-Nachbar-Anfragen in hochdimensionalen Räumen zu beschleunigen. Es wird kein daten-partitionierender, sondern einen objekt-approximierender Ansatz benutzt. Bei einer NN-Anfrage werden nur die kleineren Approximationen gescannt. Jede Approximation bestimmt eine untere und eine obere Grenze für die Entfernung zwischen dem dazugehörigen Datenpunkt und der Anfrage. Mittels der Grenzen kann man die überwiegende Mehrheit der Vektoren von der Suche ausschließen. I/O Kosten fallen nur für die wenigen Kandidaten an, die nach dem Filterschritt übrig bleiben.

- Aufbau
  - 2 Dateien: vector file (Datenpunkte) und vector approximation file (Approximationen der Vektoren, typischerweise 6 mal kleiner als das vector file)
  - die Ordnung der beiden Dateien ist identisch (dem Vektor x an *i*-ter Position im vector file entspricht die Approximation an *i*-ter Position im approximation file)

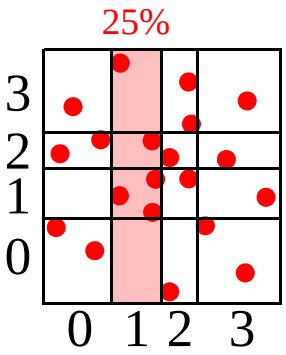


vector file	
o_1	(0.9, 0.1)
o_2	(0.6, 0.8)
o_3	(0.1, 0.4)
o_4	(0.1, 0.9)
o_5	(0.3, 0.7)

approximation file	
o_1	(3, 0)
o_2	(2, 3)
o_3	(0, 1)
o_4	(0, 3)
o_5	(1, 2)

- Operationen
  - Insert: am Ende anfügen (d.h. einfache Einfüge-Operation) und Approximation berechnen
  - Update: anpassen der Werte
  - Delete: Einträge in beiden Dateien auf NULL setzen

- Die Daten werden verlustbehaftet komprimiert:
  - ein unregelmäßiges Gitter (quantilbasiert) über den Datenraum legen (r Unterteilungen pro Dimension d)
  - dadurch Aufteilung des Datenraums in $2^{d \cdot r}$ Zellen
  - statt der exakten Punktkoordinaten: Speicherung der Zellennummer
  - Reduktion des Datenvolumens auf $r/32$ (bei 32-Bit-**float**s)
  - die Lesezeit für die Daten reduziert sich um denselben Faktor



- Die CPU-Zeit für die Berechnung der Distanzen reduziert sich durch Trick geringfügig:

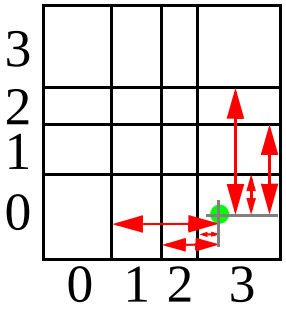
  Pro Dimension werden die Abstandsquadrate zwischen dem Anfragepunkt und den einzelnen Partitionierungslinien vorberechnet und tabelliert (Lookup Table).

  Hierdurch wird die gewöhnlicheAbstandsberechnung

$$\text{dist}^2 = \sum_{0 \le i < d} (p_i - q_i)^2$$

  ersetzt durch folgende Operation

$$\text{dist}^2_{\text{va}} = \sum_{0 \le i < d} \text{lookup}_i[c_i]$$



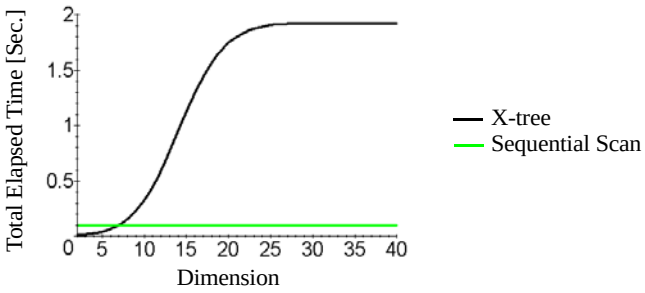
- Durch verlustbehaftete Kompression Informationsverlust:
  - nur wenn Zelle ganz von Bereichsanfrage eingeschlossen, sicherer Treffer
  - fast immer Mehrdeutigkeiten bei nächstem Nachbarn
- Verfeinerungsschritt ist nötig:
  - Lade exakte Punktinfo (1 wahlfreier Zugriff pro Kandidat) und ermittle Distanz.
    Die exakten Punktinformationen sind in einer separaten Datei gespeichert, die genauso wie das VA-File sortiert ist.
  - Bereichsanfrage: Jede (echt) geschnittene Zelle muß verfeinert werden
  - NN-Algorithmus: Wie im indexbasierten Fall sind verschiedene Strategien denkbar
- Bei grober Gitterauflösung r hohe Kosten für die Verfeinerungsschritte
  - typischerweise 6-8 Bit optimal, jedoch abhängig von der Datenverteilung

9.3 Dynamische Blockgrößenoptimierung [BK 00]

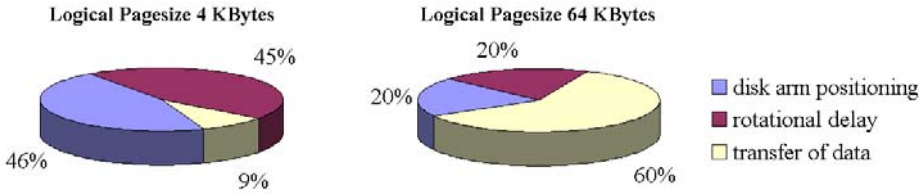
Hauptproblem von R-Baum-artigen Indexstrukturen bei hochdimensionalen Räumen:

- Viele Seiten (insbesondere Datenseiten) werden zugegriffen
- Zugriffe erfolgen wahlfrei, d.h. meist ist für jeden Zugriff eine Repositionierung des Plattenarms erforderlich

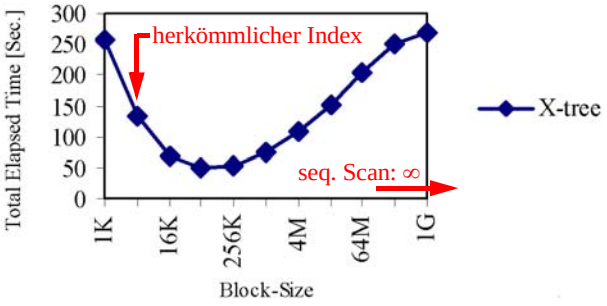
- Die indexbasierte Anfragebearbeitung ist deshalb häufig dem sequenziellen Scan (bzw. VA-File) unterlegen:



- Problem: Pro wahlfreiem Zugriff werden zu wenige Daten eingelesen. Z.B. 4KB-Blöcke:
  - 4 ms Seektime = Zeit um den Plattenarm auf die richtige Spur zu bringen
  - 4 ms Latenzzeit = Zeit um die Rotation an die richtige Stelle abzuwarten
  - < 1 ms Transferzeit = Zeit um produktiv Daten zu lesen



- Bei sehr großen Blockgrößen weniger Overhead für die wahlfreien Zugriffe
- Die Blockgröße muß geeignet optimiert werden:



- Konventionelle Vorgehensweise:
  - Kosten gemäß Kostenmodell für verschiedene Blockgrößen schätzen
  - Index bei der Erzeugung entsprechend parametrisieren
- Nachteile:
  - Kosten abhängig von der Datenverteilung (fraktale Dimension)
    erst bekannt, wenn Daten vorhanden sind, nicht unbedingt bei Erzeugung
  - Datenverteilung kann sich auch mit der Zeit ändern
  - Kosten auch abhängig von der Anzahl gespeicherter Punkte
- Besser: Index, der seine Seitengröße dynamisch anpaßt
- Vgl. X-tree Superknoten:
  - Nur Directory-Knoten werden zu Superknoten
  - Zweck: Überlappung vermeiden
  - Nicht: Kosten/Overhead Tradeoff optimieren
  - Im Hochdimensionalen sind die Directory-Kosten ohnehin unerheblich

9.4 IQ-tree (Independent Quantization) [BBJ+ 00]

Motivation

Hauptnachteile der dynamischen Blockgrößenoptimierung:

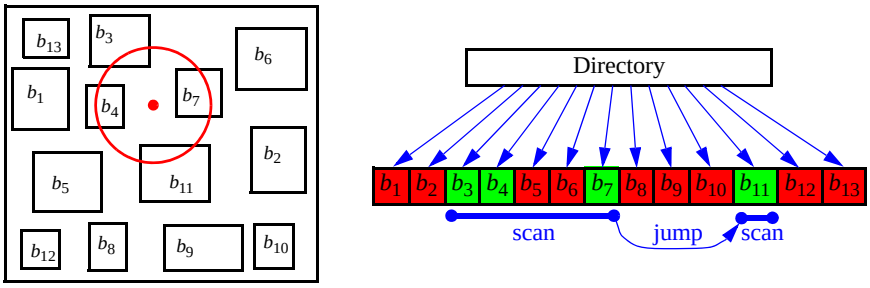
- aufwändige Speicherverwaltung durch völlige Freigabe der Blockgröße
- VA-file ist in vielen Fällen immer noch überlegen

Ideen des IQ-tree:

- statt Optimierung der Blockgröße: Optimierter Index-Durchlauf: **Fast Index Scan**
  (mehrere aufeinanderfolgende Seiten mit einem I/O-Auftrag einlesen)
- flaches Directory
- kombiniere Indexstruktur mit der Idee der Vektor-Approximation:
  Über jede Seite Gitter mit individ. Auflösung gelegt (IQ=Independent Quantization)
- optimiere auch (dynamisch) die Gitterauflösung

Fast Index Scan

- Einfach für Range-Queries, da aufgrund der Directory-Information unmittelbar entscheidbar ist, welche Seiten benötigt werden



- Annahme hier (nur für Grafik):
  Ein Seek ist um den Faktor 2.5 teurer als das sequenzielle Lesen einer Seite

  Erklärung der Grafik:
  - b_3 und b_4 werden gemeinsam gelesen, weil beide benötigt und aufeinanderfolgend
  - b_5 und b_6 nicht gebraucht, aber trotzdem gelesen, weil billiger als überspringen
  - für Zugriff von b_{11} lohnt sich der Seek, da das seq. Lesen von b_8...b_{10} zu teuer

  → in diesem einfachen Fall würde evtl. auch das Laufwerk selbst den Zugriff optimieren

Fast Index Scan für Nearest Neighbor Queries

- Es ist nicht von vornherein bekannt, welche Seiten benötigt werden
- Bei flachem Directory:

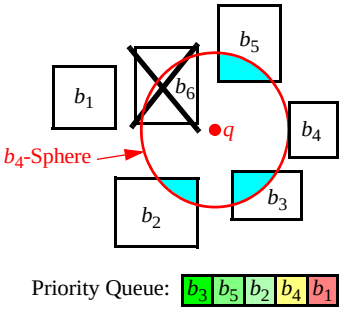
  Prioritätsalgorithmus kennt die Reihenfolge, in der die Seiten gelesen werden

  → aufsteigender Abstand vom Anfragepunkt,

  aber nicht, bei welcher Seite er stoppen wird

  → sobald der NN-Abstand kleiner als der Seitenabstand ist
- Daher kann man (fast) nie sicher entscheiden, ob Seiten, die man vor oder nach der “aktuellen” Seite zusätzlich liest, auch im Endeffekt gebraucht werden

- Aber man kann die Wahrscheinlichkeit dafür schätzen, ob sie gebraucht werden. Beispiel:
  - Ann.: b_6 wurde bereits früher bearbeitet
  - Gesucht: Zugriffswahrscheinlichkeit von b_4
  - b_4 wird zugegriffen, wenn die anderen Seiten *keinen* Punkt enthalten, der näher als die MINDIST von b_4 ist, d.h. in dem gefärbten Volumen



$$P_{\text{access}}(b_i) = \prod_{b_k \in \%} \left(1 - \frac{V_{\text{intersection}}(b_k)}{V_{\text{MBR}}(b_k)}\right)^{C_{page}(b_k)}$$

wobei

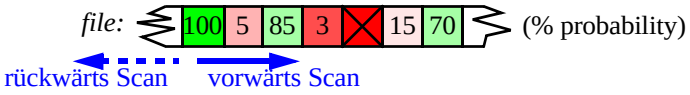
i) % = alle Blöcke, die in der APL (Active Page List, implementiert als Priority Queue) vor b_i liegen, d.h.

MINDIST (b_k) < MINDIST (b_i)

und noch nicht verarbeitet sind

ii) $C_{page}(b_x)$ die Kapazität der Seite b_x, d.h. die Anzahl der Punkte in der Seite

- Wie wird die Zugriffswahrscheinlichkeit berücksichtigt?



  - Bei jedem Plattenzugriff wird auf jeden Fall die Seite mit der höchsten Priorität eingelesen (Zugriffswahrscheinlichkeit 100%)
  - Seiten, die schon bearbeitet wurden, haben eine Zugriffswahrscheinlichkeit von 0%, da auf keine Seite ein zweites mal zugegriffen wird
  - Ausgehend von dieser Seite werden auch die Nachbarseiten (Nachbar bzgl. Position in Indexdatei) betrachtet
  - Informal: wenn Seiten mit hoher Zugriffswahrscheinlichkeit in der Nähe der Seite liegen, die die höchste Priorität hat, lies diese Seiten mit
- Formal: Trade-off zwischen
  - Augenblicklichen Aufwändungen (es wird *mehr* gelesen, also augenblicklich teurer)
  - Späteren eventuellen Einsparungen (der teure Einzelzugriff wird evtl. überflüssig)

  $\Rightarrow$ Das Laden der Seite b_i mit der Zugriffswahrscheinlichkeit $P_{access}(b_i)$ hat die folgende Auswirkung auf die kumulierte Kostenbilanz:

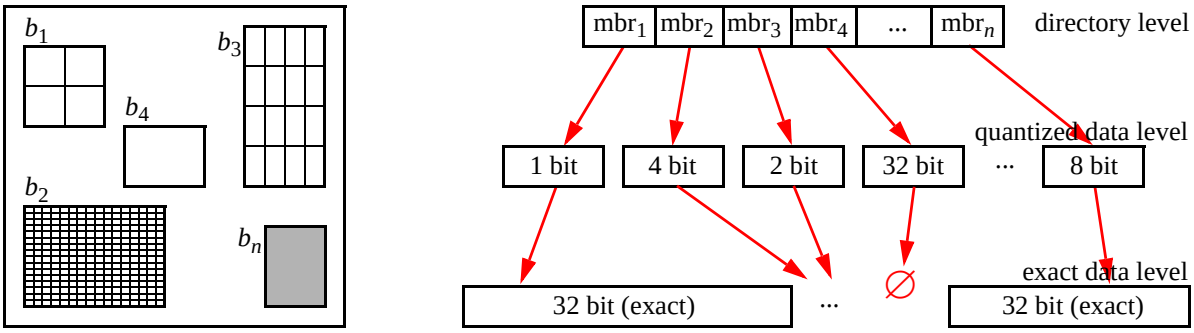
  $$c_i = t_{\text{transfer}} - P_{\text{access}}(b_i) \cdot (t_{\text{seek}} + t_{\text{lat}} + t_{\text{transfer}})$$

  Wenn c_i negativ ist, dann ist die geschätzte Einsparung größer als die zusätzlichen Transferkosten.
- Optimiere die kumulierte Kostenbilanz $CCB = \sum_i c_i$

  Der Algorithmus wird einmal in Vorwärts- und einmal in Rückwärtsrichtung gestartet und stoppt jeweils, wenn *CCB* größer als $t_{seek} + t_{lat}$ ist.

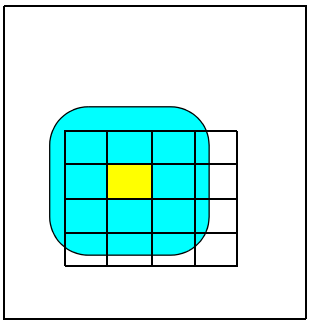
  Nachdem die Sequenz aus Blöcken mit der minimalen Kostenbilanz bestimmt wurde, werden die Seiten geladen und bearbeitet.

Struktur des IQ-tree



- 1. Ebene: Flaches Directory mit folgenden Informationen:
  - MBR der Seite
  - Verweis auf Datenseite (2. Ebene) mit komprimierter Information
  - Verweis auf Datenseite (3. Ebene) mit exakter Information
- 2. Ebene: Quantisierte Daten:
  - einheitliche Blockgröße in Bytes; unterschiedliche Seitenkapazität
  - über jede Datenseite ist ein *regelmäßiges* Gitter gelegt
    experimentell ermittelt, daß folgende Quantisierungsarten ähnliche Leistung haben
    - quantilbasiertes Gitter über dem gesamten Datenraum (VA-File)
    - reguläres Gitter über den einzelnen Datenseiten (IQ-tree)
  - jede Datenseite ist mit einer individuellen Auflösung r quantisiert,
    wobei $r \in \{1,2,4,8,16,32\}$ (32 = *nicht komprimiert, floating-point-Werte*)
    im Unterschied zum VA-file sind die Quantisierungen relativ zur Geometrie des MUR der zugehörigen Datenseite → benötigt weniger Bits
  - die Seitenkapazität ergibt sich aus der Auflösung
    (eine Stufe gröber → doppelte Kapazität)
- 3. Ebene: Exakte Daten
  - unterschiedliche Blockgröße, je nach Kapazität der äquiv. Seite auf der 2. Ebene
  - Blockgröße variiert aber nur in 2er-Potenzen, d.h. keine echte Fragmentierung

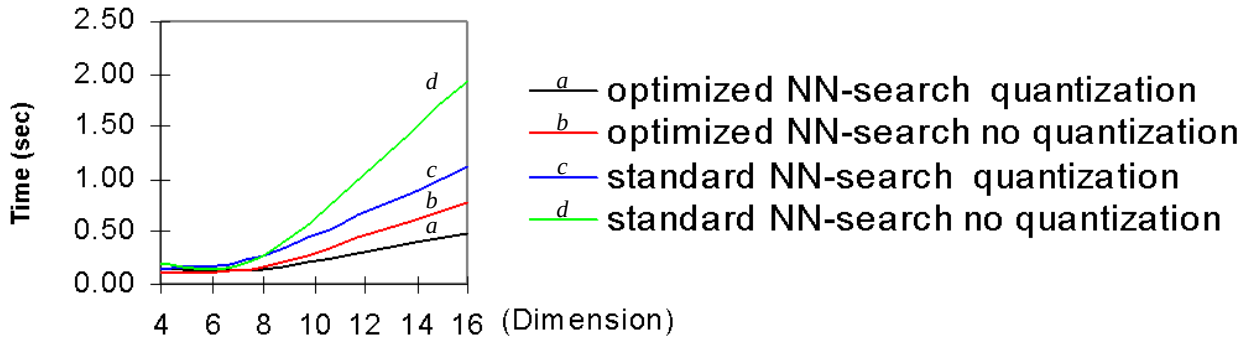
Optimierung der Quantisierung



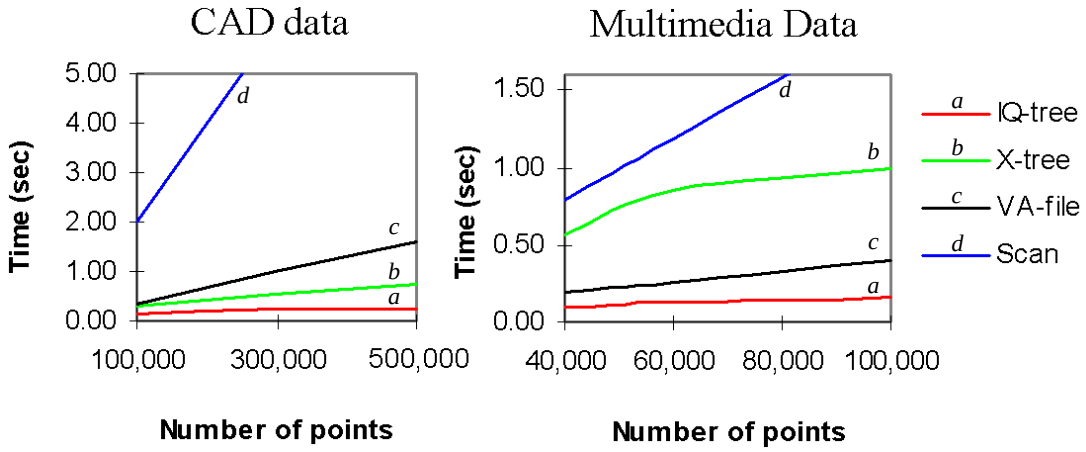
- Wahrscheinlichkeit, daß eine Gitterzelle von beliebiger Query zugegriffen wird, läßt sich abhängig von der Gitterauflösung schätzen (entspricht der dunkel eingefärbten Fläche)
- Trade-Off informell:
  - Gitter zu grob → viele Zugriffe auf 3. Ebene
  - Gitter zu fein → höhere Kosten auf der 2. Ebene (mehr Seiten)
- Optimierungsaufgabe: optimiere die Gesamtwahrscheinlichkeit mit Hilfe obiger Schätzung

Experimente

- Einfluß von optimaler Quantisierung und Fast Index Scan (gleichverteilte Punkte)



- Vergleich mit Konkurrenztechniken für CAD- und Multimediadaten



9.5 Literatur

Literatur

[BBJ+ 00]Berchtold S., Böhm C., Jagadish H.V., Kriegel H.-P, Sander J.: *Independent Quantization: An Index Compression Technique for High-Dimensional Spaces*, Int. Conf. on Data Engineering, ICDE 2000.

[BK 00] Böhm C., Kriegel H.-P.: *Dynamically Optimizing High-Dimensional Index Structures*, Int. Conf on Extending Datab. Techn. (EDBT), 2000.

[BKK 96]Berchtold S., Keim D., Kriegel H.-P.: *The X-tree: An Index Structure for High-Dimensional Data*. VLDB 1996, 28-39.

[WSB 98]Weber R., Schek H.-J., Blott S.: *A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces.* Proc. 24th Int. Conf. on Very Large Databases (VLDB), New York, USA. Morgan Kaufmann, 1998, pp. 194-205