4.3 Optimierung von Indexstrukturen

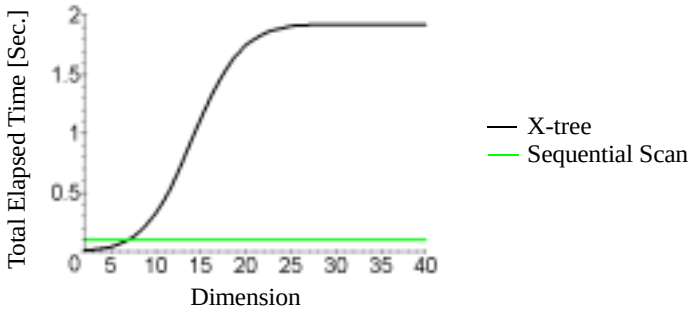
4.3.1 Dynamische Blockgrößenoptimierung

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

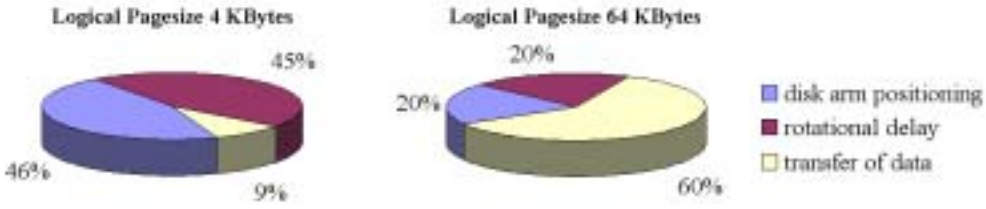
Das Kostenmodell aus Abschnitt 4.1 kann für verschiedene Optimierungen von R-Baum-artigen Indexstrukturen genutzt werden.

Hauptproblem solcher Indexstrukturen bei hochdimensionalen Räumen:

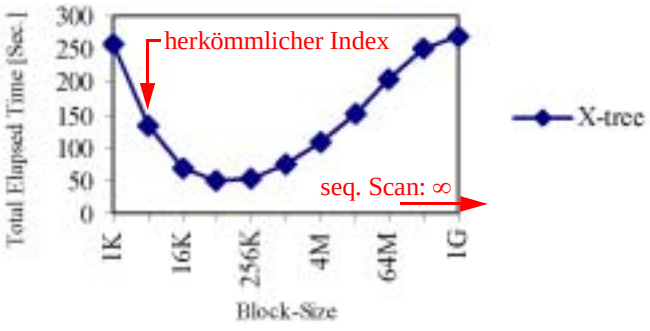
- Viele Seiten (insbesondere Datenseiten) werden zugegriffen
- Zugriffe erfolgen wahlfrei, d.h. meist ist für jeden Zugriff eine Repositionierung des Plattenarms erforderlich
- Die indexbasierte Anfragebearbeitung ist deshalb häufig dem sequenziellen Scan (bzw. VA-File) unterlegen:



- 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

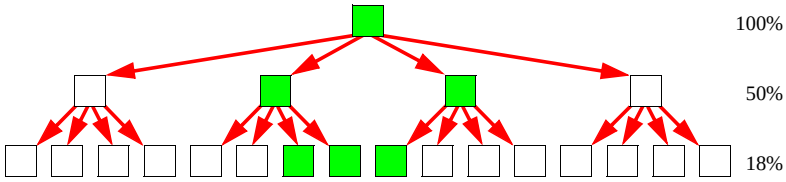


- Bei sehr großen Blockgrößen weniger Overhead für die wahlfreien Zugriffe
- **Die Blockgröße muß geeignet optimiert werden:**



- 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 Supernodes:
  — Nur Directory-Knoten werden zu Supernodes
  — Zweck: Überlappung vermeiden
  — Nicht: Kosten/Overhead Tradeoff optimieren
  — Im Hochdimensionalen sind die Directory-Kosten ohnehin unerheblich

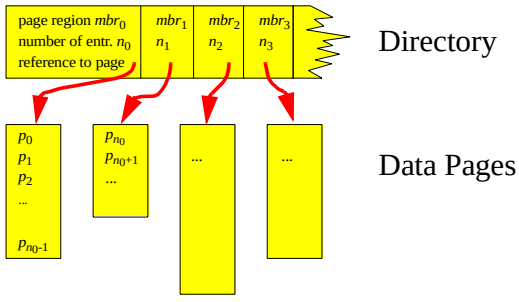
Lohnt sich ein *hierarchisches* Directory aus Sicht der Anfragebearbeitung?



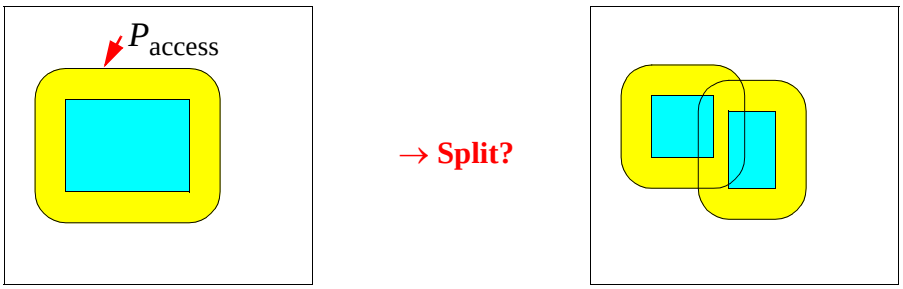
- Je höher der Index-Level, umso schlechter wird die Selektivität
  — Grund: Zu jeder zugegriffenen Blattseite müssen alle Vorgänger zugegriffen werden
- Für hochdimensionale Anfragebearbeitung:
  — typischerweise ist nur *ein* Directory-Level (Wurzel + Datenseiten) gerechtfertigt

Struktur des selbstoptimierenden Index

- Einstufiges Directory
- Jede Seite hat eine individuelle Kapazität, die im Directory vermerkt ist und die dynamisch nach Kostenmodell angepaßt wird
- Nachteil:
  Komplexes Management des freien Speichers



- Seiten sind immer zu 100% gefüllt
  - Vorteil: keine nutzlose Information (leerer Anteil) wird zur Platte transferiert
  - Nachteil: Für jeden neu gespeicherten Punkt muß die Seite an einem neuen Platz auf der Platte gespeichert werden. Dies wirkt sich aber nicht negativ auf die Performanz aus, da die Seite ohnehin nach Einfügen gespeichert werden muß
  - Es gibt keine “Überlaufbedingung” mehr
- Statt Überlaufbedingung:
  Führe jeweils nach x (Parameter) Einfügungen folgenden Test durch:



- Folgende zu erwartende Kosten treten für eine Seite auf:
  $cost = P_{\text{access}} \cdot (t_{\text{pos}} + t_{\text{lat}} + |pg| \cdot t_{\text{transfer}}) + |dirent| \cdot t_{\text{transfer}}$.
- Stelle die Kosten der ungesplitteten Seiten den kumulierten Kosten der beiden neuen Seiten gegenüber

Verhalten

- Bei sehr großen Seiten (d.h. mit vielen Einträgen):
  - Kosten sind durch die Transferzeit dominiert
  - Split lohnt sich, auch wenn der Selektivitätsgewinn nur geringfügig ist
- Bei sehr kleinen Seiten:
  - Kosten sind durch Seek- und Latenzzeit dominiert
  - Split lohnt sich nur bei sehr hohen Selektivitätsgewinnen
- Dazwischen
  - Durch die Kostenvergleiche wird automatisch das Optimum eingestellt

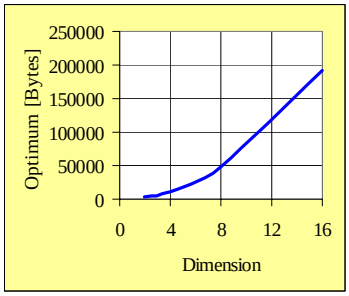
Vorteile der dynamischen Technik

- “Lokale” Ermittlung der Zugriffswahrsch. wesentlich einfacher als globales Modell:
  - Globales Modell: Ausdehnung einer *typischen* Seite wird geschätzt
  - Lokales Modell: Ausdehnung einer konkreten Seite ist bekannt
  - In der Seite gespeicherte Punkte können zur Schätzung des NN genutzt werden
  - Dies führt zu höherer Genauigkeit der Schätzung
  - Kein *a priori* Wissen der Datenverteilung ist erforderlich.
- Index adaptiert sich an Veränderungen in der Datenverteilung über Zeit und Ort

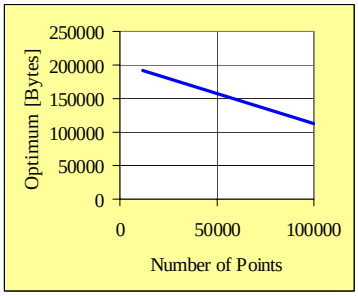
Experimentelle Bewertung

- Optimale Blockgröße (gleichverteilte Daten)

12,000 Points, uniform/indep.

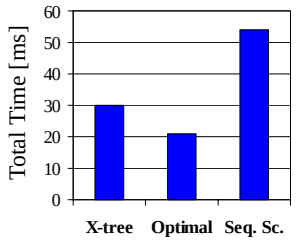


16d Points, uniform/indep.

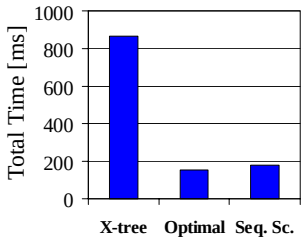


- Verbesserung gegenüber X-tree und Sequential Scan

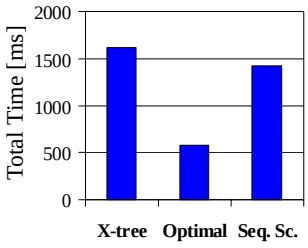
4d, 12,000 Points, unif.



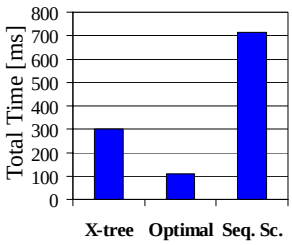
16d, 12,000 Pts, unif.



16d, 100,000 Pts, unif.



16d, 50,000 Pts, CAD



4.3.2 IQ-tree (Independent Quantization)

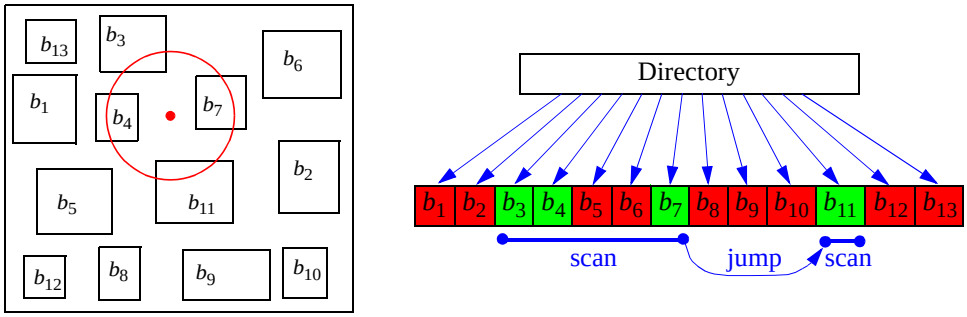
[BBJ+ 00] Berchtold S., Böhm C., Jagadish H.V., Kriegel H.-P, Sander J.: *Independent Quantization: An Index Compression Technique for High-Dimensional Spaces*, Int. Conf. on Data Engineering, ICDE 2000.

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

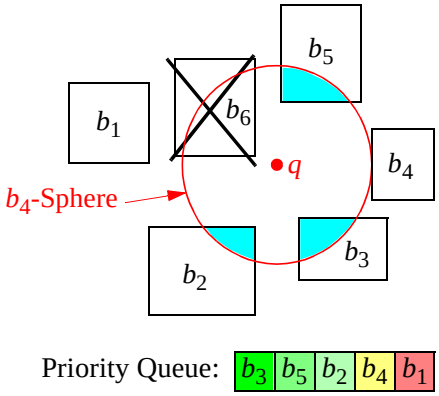


- Annahme hier (nur für Grafik):
  Ein Seek ist um den Faktor 2.5 teurer als das sequenzielle Lesen einer Seite
- Erklärung der Grafik:
  — b_3 und b_4 werden gemeinsam gelesen, weil beide benötigt und aufeinanderfolgend
  — b_5 und b_6 nicht gebraucht, aber trotzdem gelesen, weil billiger als überspringen
  — 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

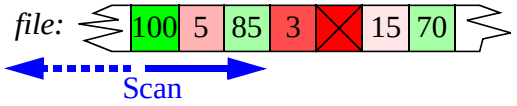
- Es ist nicht von vornherein bekannt, welche Seiten benötigt werden
- Bei flachem Directory:
  Prioritätsalgorithmus kennt die Reihenfolge, in der die Seiten gelesen werden
  — aufsteigender Abstand vom Anfragepunkt,
  aber nicht, bei welcher Seite er stoppen wird
  — sobald der NN-Abstand kleiner als 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

$$P(b_i) = \prod_{b_k \in \%} \left(1 - \frac{V_{\text{intersection}}(b_k)}{V_{\text{MBR}}(b_k)}\right)^{C(b_k)}$$



% = alle Blöcke, die in der APL vor b_i liegen, d.h.
MINDIST (b_k) < MINDIST (b_i)
und noch nicht verarbeitet sind.

- Wie wird die Zugriffswahrscheinlichkeit berücksichtigt?



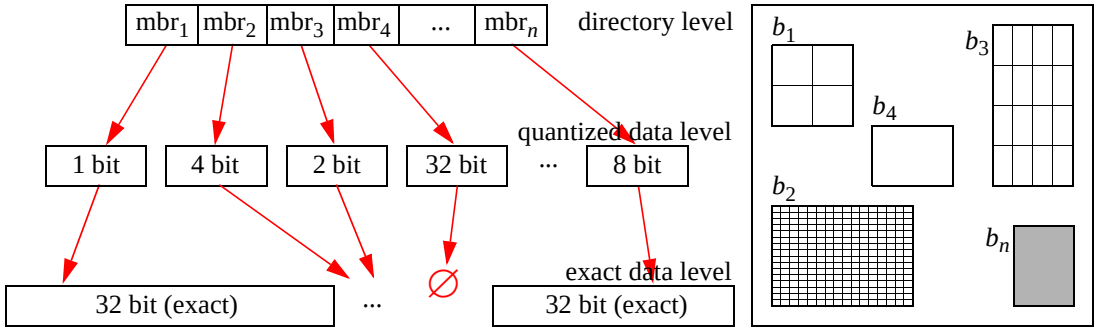
  — Bei jedem Plattenzugriff wird auf jeden Fall die Seite mit der höchsten Priorität eingelesen (Zugriffswahrscheinlichkeit 100%)
  — Ausgehend von dieser Seite werden auch die Nachbarseiten (Nachbar bzgl. Position in Indexdatei) betrachtet
  — Informal: wenn Seiten mit hoher Zugriffswahrscheinlichkeit 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)

- Optimiere die kumulierte Kostenbilanz

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

Struktur des IQ-tree



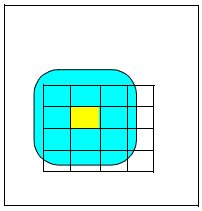
- 1. Ebene: Flaches Directory mit folgenden Informationen:
  — MBR der Seite
  — Verweis auf Datenseite (2. Ebene) mit komprimierter Information
  — Verweis auf Datenseite (3. Ebene) mit exakter Information

- 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*)
— 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)

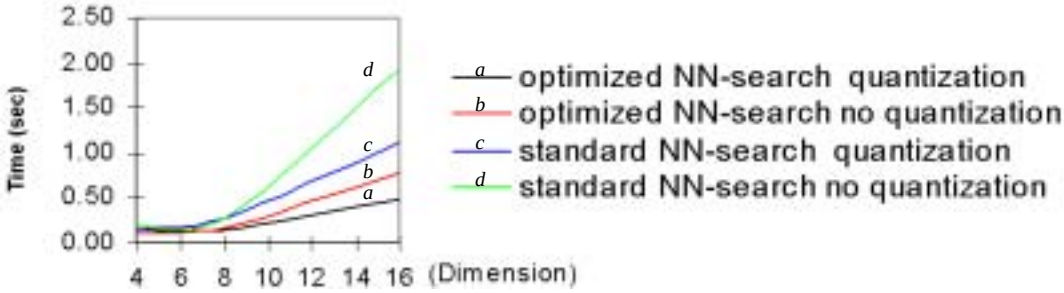
Optimierung der Quantisierung



- Wahrscheinlichkeit, daß eine Gitterzelle von beliebiger Query zugegriffen wird, läßt sich mit der Minkowski-Summe ermitteln
- Erweiterung erforderlich, um den Fast-Index-Scan mit zu berücksichtigen
- Trade-Off informell:
  — Gitter zu grob → viele Zugriffe auf 3. Ebene
  — Gitter zu fein → höhere Kosten auf der 2. Ebene (mehr Seiten)

Experimente

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



- Vergleich mit Konkurrenztechniken für CAD- und Multimediadaten