2.2 Algorithmische Paradigmen zur Anfragebearbeitung

2.2 Algorithmische Paradigmen zur Anfragebearbeitung

2.2.1 Übersicht

- Typen von Ähnlichkeitsanfragen
  - Bereichsanfragen
  - Nächste Nachbarn Anfragen
  - Reverse Nächste Nachbarn
  - Skyline Anfragen
- Algorithmische Paradigmen
  - Naive (sequentielle) Suche
    - Für alle *n* Objekte der Datenbank wird das Anfrage-Prädikat (d.h. meist eine Distanzberechnung zum Anfrageobjekt) ausgewertet
    - Kosten: O($n \cdot$ Kosten für das Anfrage-Prädikat)
  - Indexbasierte Suche
  - Mehrstufige Anfragebearbeitung

2.2.2 Indexstrukturen

2.2.2 Indexstrukturen

- Prinzip [Böhm, Berchtold, Keim. ACM Computing Surveys, 2001]
  - Organisiere Objekte der Datenbank so, dass während einer Ähnlichkeitsanfrage nur auf „relevante“ Objekte zugegriffen werden muss
  - Baumartige Organisation; jedem Knoten des Baumes ist zugeordnet
    - Seite des Hintergrundspeichers
    - Region des Datenraums
  - Typen von Knoten (= Seiten)
    - Blattknoten sind Datenseiten, speichern Objekte
    - Innere Knoten sind Directoryseiten, speichern Directory-Einträge
      - Verweis zur Kindseite (Adresse auf dem Hintergrundspeicher)
      - Beschreibung der Region der Kindseite

2.2.2 Indexstrukturen

- Physische vs. logische Seiten
  - ursprünglich: eine physische Seite des Hintergrundspeichers wird verwendet
    - kleinste Informationseinheit, die zwischen Plattenspeicher und RAM übertragen werden kann
    - ABER: physische Seiten meist zu klein
  - daher: logische Seiten fassen aufeinanderfolgende physische Seiten zusammen
  - Meistens: einheitliche Seitengröße für alle Seiten eines Indexes (um komplizierte Freispeicherverwaltung zu vermeiden)
  - Kapazität der Directory-/Datenseiten (max. Anzahl der Einträge)

$$c_{\text{Data}} = \left\lfloor \frac{\text{Seitengröße} - \text{Verwaltungsoverhead}}{\text{Größe eines Datensatzes}} \right\rfloor \qquad c_{\text{Directory}} = \left\lfloor \frac{\text{Seitengröße} - \text{Verwaltungsoverhead}}{\text{Größe eines Directoryeintrags}} \right\rfloor$$

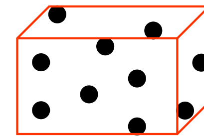
  - Meist keine 100% Füllung der Seiten (Platz für neue Datensätze) aber minimale Füllung (z.B. 40%) wegen Speicherauslastung
  - Speicherauslastung ist die durchschnittliche Anzahl besetzter Einträge

2.2.2 Indexstrukturen

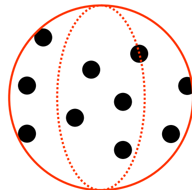
- Jedes Objekt wird in genau einer Datenseite gespeichert
- Regionen gewährleisten, dass ähnliche Objekte möglichst auf den selben Datenseiten (oder Teilbäumen) gespeichert werden

2.2.2 Indexstrukturen

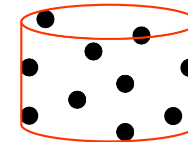
- Gestalt der Seitenregionen
  - Vektordaten:
    - Achsenparallel Rechtecke, die minimal um die Punktmenge gespannt werden (MUR=minimal umgebendes Rechteck/ MBR=minimum bounding rectangle) (R-Tree, R*-Tree, X-Tree, …)



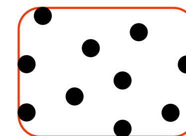
    - Kugeln (SS-Tree)



    » Zylinder (TV-Tree)



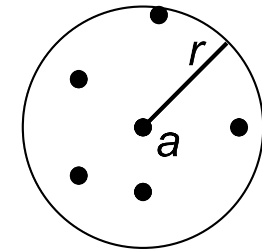
    - Kombinationskörper (Kugel+MBR, SR-Tree)



2.2.2 Indexstrukturen

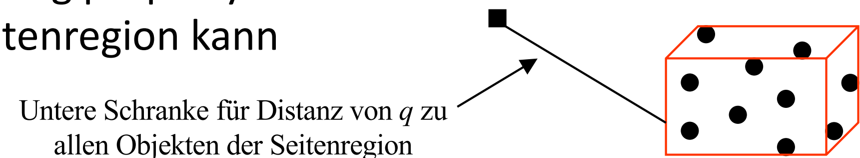
- Gestalt der Seitenregionen (cont.)
  - Algemein metrische Daten
    - Kein „Raum“ im Euklidischen Sinne, keine Geometrie
    - Ankerobjekt *a* (anchor object) + Hüllradius *r* (covering radius)

      Für alle Objekte *o* der Seitenregion gilt: dist(*o*,*a*) $\leq r$



2.2.2 Indexstrukturen

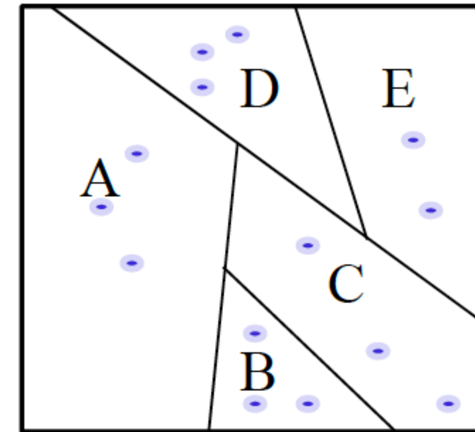
- Seitenregion umfasst immer alle Objekte der Datenseite bzw. des Teilbaums der Directoryseite
  - Konservative Approximation, lower-bounding property: die Distanz von einem Objekt zu einer Seitenregion kann immer mit einer unteren Schranke abgeschätzt werden

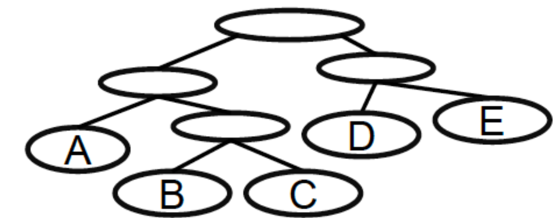


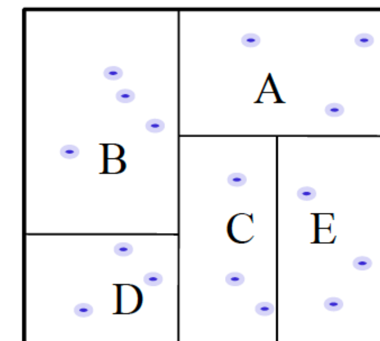
  - Damit kann man Vollständigkeit der Anfrageergebnisse gewährleisten
- Bäume sind meist balanciert (alle Blätter auf selbem Level)
- Indexstrukturen sind dynamisch, d.h. Insert/Delete sind effizient
  - Tiefensuche nach entsprechender Datenseite (auf einen Pfad beschränkt)
  - Einfügen/Löschen des Objektes
  - Überlauf/Unterlauf-Behandlung durch Split/Merge
    - Verschiedene Kriterien (minimale Überlappung, toter Raum, …)
    - Verschiedene Algorithmen (linear, quadratisch, …)
    - Entsprechendes Einfügen/Löschen im Elternknoten (rekursiv, notfalls bis Wurzel)

2.2.2 Indexstrukturen

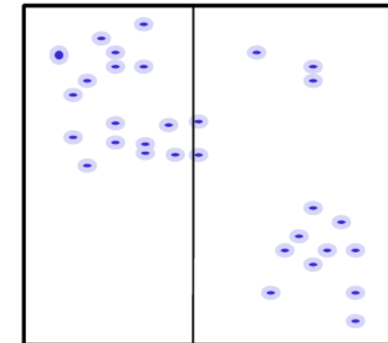
- Binary Space Partitioning Tree (BSP-Tree):
  - Wurzel enthält gesamten Datenraum
  - Jeder innere Knoten hat 2 Söhne
  - Datenobjekte in den Blättern
  - Bekannteste Variante: kD-Tree
    - max. Seitenkapazität sind M Einträge
    - min. Seitenkapazität sind M/2 Einträge
    - bei Überlauf achsenparalleler Split
    - nach Löschen Vereinigung von Geschwisterknoten
    - Split-Achse wechselt nach jedem Split
    - 50%-50% Aufteilung der Daten





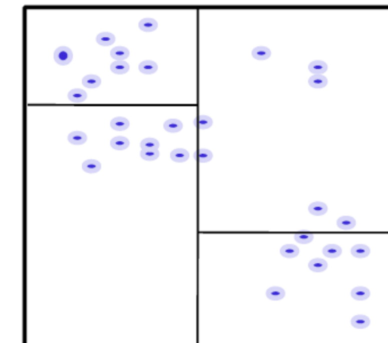


2.2.2 Indexstrukturen

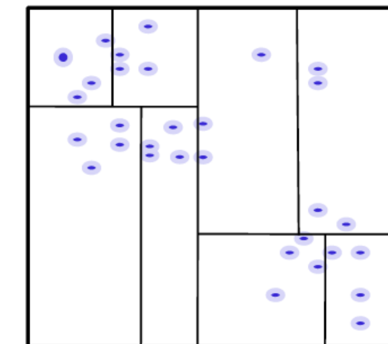


- Problem
  - keine Balancierung (Degeneration des Baums)
  - Korrektur der Balancierung möglich aber aufwendig

    => Hohe Update-Komplexität

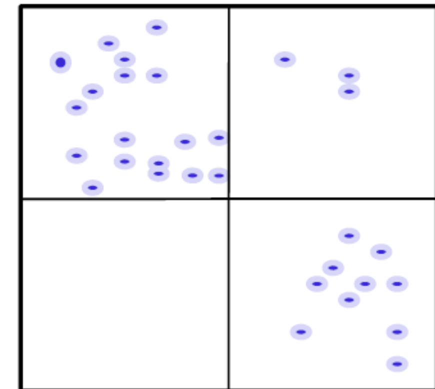


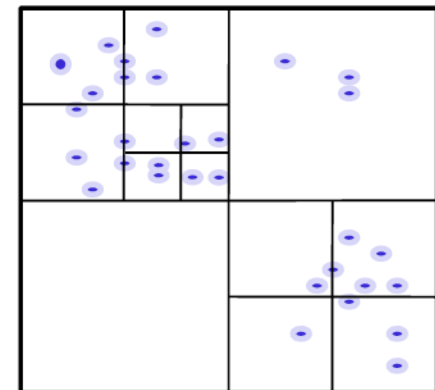
- Bulk-Load
  - Annahme: Kenntnis aller Datenobjekte
  - Aufbau: durch rekursive 50%-50% Aufteilung der Objekte bis jedes Blatt weniger als M Objekte enthält
  - Bulk-Load erzeugt immer einen balancierten Baum

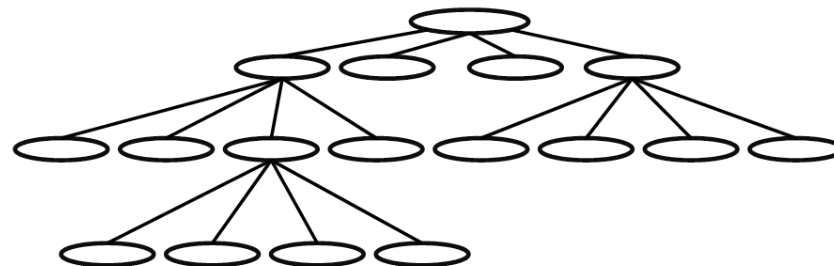


2.2.2 Indexstrukturen

- QuadTree für 2D Daten
  - Wurzel stellt den ganzen Datenraum dar
  - Jeder innere Knoten hat 4 Nachfolger
  - Geschwisterknoten teilen den Raum ihres Elternknotens in 4 gleich große Teile ein
  - Quad-Trees sind i.d.R. nicht balanciert
  - Seiten haben einen max. Füllungsgrad M, aber keine Mindestfüllung
  - Datenobjekte in den Blättern



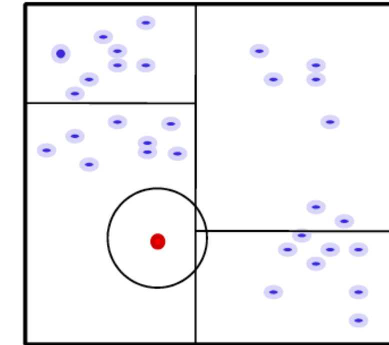




2.2.2 Indexstrukturen

- Raumpartitionierende Verfahren:
  - Aufteilung des gesamten Datenraums durch Splits in den Dimensionen
  - Seitenregionen enthalten toten Raum

    => evtl. schlechtere Suchperformanz bei räumlichen Anfragen

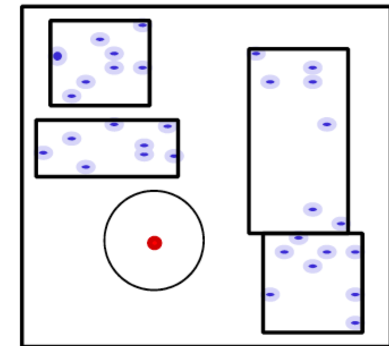


Range-Query in BSP-Tree

- Datenpartitionierende Verfahren:
  - Beschreibung der Seiten-Region durch minimal umgebende Regionen (z.B. Rechtecke)

    => Bessere Pruning Leistung
  - Seitenregionen können überlappen

    => Degeneration bzgl. Überlappung



Range-Query in R-Tree

2.2.2 Indexstrukturen

- Split- und Einfüge-Algorithmen minimieren:
  - Überlappung der Seitenregionen
  - Toten Raum in den Seiten
  - Balancierung bzgl. des Füllungsgrades