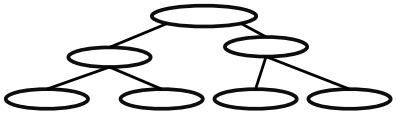
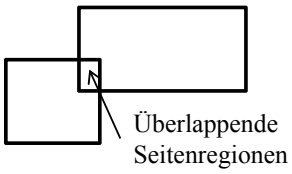
Klassische Indexstrukturen

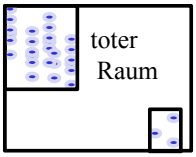
- Verwaltung von räumlichen Objekten kann auch über räumliche Suchbäume erfolgen
- Suchbäume passen ihre Seitenregionen (Zonen) der Datenverteilung an
  - ⇒ Garantie über maximale Füllung einer Seitenregion/Zone
  - ⇒ Bessere Suchperformanz durch Reduktion der betrachteten Objekte
  - ⇒ Es entsteht Aufwand bei der Anpassung des Suchbaumes
- Anpassung durch rekursive Aufteilung des Raumes (Quad-Tree, BSP-Trees)
- Anpassung durch Aufteilen der Daten auf minimale umgebende Seitenregionen

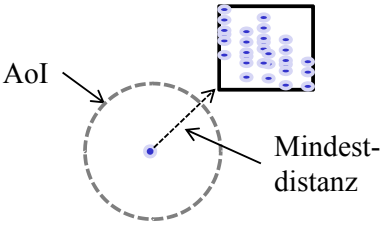
Wichtige Merkmale von räumlichen Suchbäumen

- *Seitenregion*: umgebende Approximation mehrerer Objekte
- *Balancierung*: Unterschiedlichkeit der Pfadlängen von der Wurzel zu Blattknoten
- *Seitenkapazität*: Anzahl der Objekte die mindestens/höchsten in der Seitenregion liegen.
- *Überlappung*: Ist Schnitt zwischen Seitenregionen erlaubt
- *toter Raum*: Raum in dem keine Seitenregionen/Objekte liegen
- *Pruning*: Ausschluß aller Objekte in einer Seitenregion durch Test auf Seitenregionen









Anforderungen in MMO Servern

- i.d.R. liegt der ganze Baum im Hauptspeicher
- hohe Änderungsrate, jede Positionsänderung eines Game Entities
  - Je nach Spiel eine Änderung pro Tick
  - Struktur des Baums kann sich drastisch verändern
- hohe Anfragerate
- Unterstützung mehrerer Anfrage in einem Tick
- Dimensionalität der Objekte ist 2D bzw. 3D
- Objekte haben räumliche Ausdehnungen (Kollision, Hitbox..)

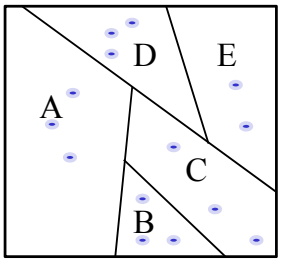
Folgerungen:

⇒ Datenstrukturen, die primär Seitenzugriffe optimieren sind ungeeignet (Baum ist im Hauptspeicher)

⇒ Overhead für Index-Aufbau/Anpassung muss durch die Anfragebearbeitung aufgewogen werden.

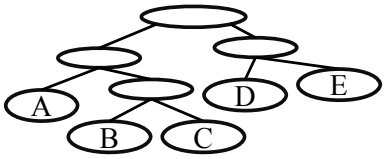
Binary Space Partitioning Trees (BSP-Tree)

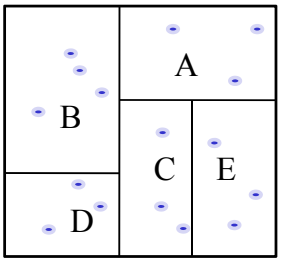
- Wurzel enthält gesamten Datenraum.
- Jeder innere Knoten hat 2 Söhne
- Datenobjekte in den Blättern



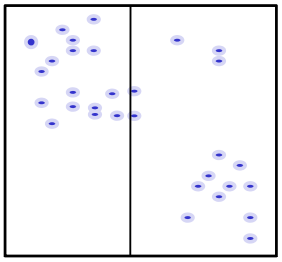
Bekannteste Variante: ***kD-Tree***

- max. Seitenkapazität sind *M* Einträge
- min. Seitenkapazität sind *M/2* Einträge
- bei Überlauf achsenparalleler Split
- nach Löschen Vereinigung von Geschwisterknoten
- Split-Achse wechselt nach jeden Split
- 50%-50% Aufteilung der Daten



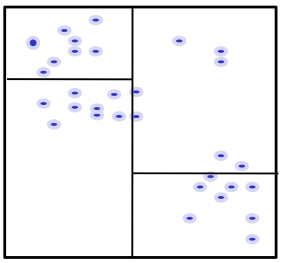


Binary Space Partitioning Trees (BSP-Tree)



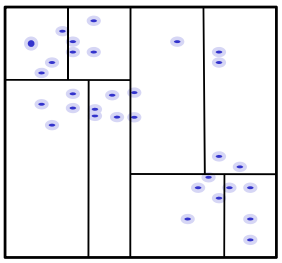
Problem bei dynamischen Verhalten:

- keine Balancierung (Degeneration des Baums)
- Korrektur der Balancierung möglich aber aufwendig
  => Hohe Update-Komplexität



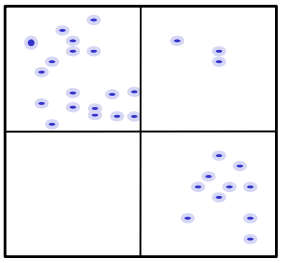
Bulk-Load

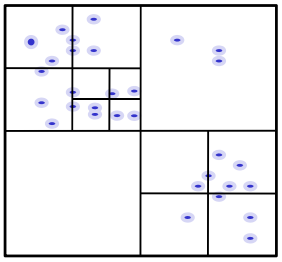
- Annahme: Kenntnis aller Datenobjekte
- Aufbau: durch rekursive 50-50 Aufteilung der Objekte bis jedes Blatt weniger als *M* Objekte enthält
- Bulk-load erzeugt immer einen balancierten Baum
- Datenseite eine Baums mit *n* Objekten und Höhe *h* enthält mindestens $\left\lfloor \frac{n}{2^h} \right\rfloor$ Objekte und höchstens $\left\lfloor \frac{n}{2^h} \right\rfloor + 1$

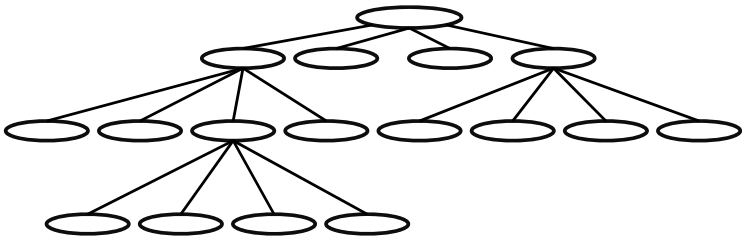


Quad-Tree

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





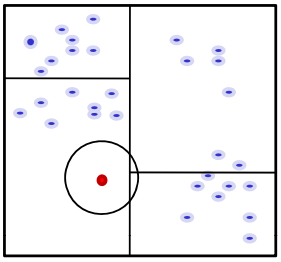


Datenpartitionierende Index-Strukturen

Raumpartitionierende Verfahren:

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

  => evtl. schlechtere Suchperformanz bei räumlichen Anfragen



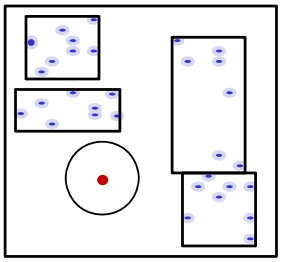
Range-Query in BSP-Tree

Datenpartitionierende Verfahren:

- Beschreibung der Seiten-Region durch ein minimales umgebende Regionen (z.B. Rechtecke)

  => Bessere Pruning Leistung
- Seitenregionen können überlappen

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



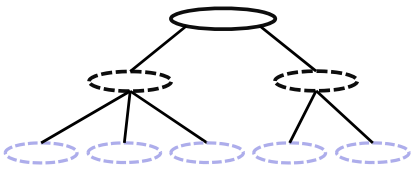
Range-Query in R-Tree

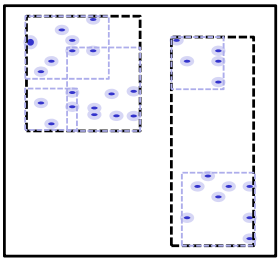
R-Baum

Struktur eines R-Baum:

- Wurzel umgibt den gesamten Datenraum und hat maximal *M* Einträge
- Seitenregionen werden durch Minimale umgebende Rechtecke (MUR) modelliert
- innere Knoten im R-Baum haben zwischen *m* und *M* Nachfolger (wobei $m \leq M/2$)
- Das MUR eines Nachfolgers ist vollständig im MUR des Vorgängers enthalten.
- Alle Blätter sind auf dem gleichen Level
- Datenobjekte werden in den Blättern gespeichert

  Mögliche Datenobjekte:
  - Punkte
  - Rechtecke





Einfügen in den R-Baum

Das Objekt x ist in einen R-Baum einzufügen

Durch Überlappung können 3 Fälle auftreten

- Fall 1: x fällt vollständig in genau ein Directory-Rechteck D
  ⇒ Einfügen in Teilbaum von D
- Fall 2: x fällt vollständig in mehrere Directory-Rechtecke $D_1, \ldots, D_n$
  ⇒ Einfügen in Teilbaum von D_i, das die geringste Fläche aufweist
- Fall 3: x fällt vollständig in kein Directory-Rechteck
  ⇒ Einfügen in Teilbaum von *D*, das den geringsten Flächenzuwachs erfährt
  (in Zweifelsfällen: ... , das die geringste Fläche hat)
  ⇒ *D* muß entsprechend vergrößert werden

Split-Algorithmus im R-Baum

(im folgenden wird von inneren Knoten ausgegangen: Objekte sind MURs)

Der Knoten K läuft mit $|K| = M+1$ über:

⇒ Aufteilung auf zwei Knoten K_1 und K_2, sodaß $|K_1| \geq m$ und $|K_2| \geq m$

Quadratischer Algorithmus

- Wähle das Paar von Rechtecken R_1 und R_2 mit dem größten Wert für den "toten Raum" im MUR, falls R_1 und R_2 in denselben Knoten K_i kämen.

  $$d(R_1, R_2) := \text{Fläche}(\text{MUR}(R_1 \cup R_2)) - \text{Fläche}(R_1) - \text{Fläche}(R_2)$$

  Setze $K_1 := \{R_1\}$ und $K_2 := \{R_2\}$.
- Wiederhole den folgenden Schritt bis zu STOP:
  - wenn alle R_i zugeteilt sind: STOP
  - wenn alle restlichen R_i benötigt werden, um den kleineren Knoten minimal zufüllen: teile sie alle zu und STOP
  - sonst: wähle das nächste R_i und teile es dem Knoten zu, dessen MUR den kleineren Flächenzuwachs erfährt. Im Zweifelsfall bevorzuge den K_i mit kleinerer Fläche des MUR bzw. mit weniger Einträgen.

schnellere Splitstrategie für R-Baum (1)

Linearer Algorithmus

- Der lineare Algorithmus ist identisch mit dem quadratischen Algorithmus bis auf die Auswahl des initialen Paars (R_1, R_2).
- Wähle das Paar von Rechtecken R_1 und R_2 mit dem "größten Abstand", genauer:
  - Suche für jede Dimension das Rechteck mit dem kleinsten Maximalwert und das Rechteck mit dem grössten Minimalwert (*maximaler Abstand*).
  - Normalisiere den *maximalen Abstand* jeder Dimension, indem er durch die Summe der Ausdehnungen der R_i in der Dimension dividiert wird (*setze den maximalen Abstand der Rechtecke ins Verhältnis zur ihrer Ausdehnung*).
  - Wähle das Paar von Rechtecken mit dem größten normalisierten Abstand bzgl. aller Dimensionen. Setze $K_1 := \{R_1\}$ und $K_2 := \{R_2\}$.
- Dieser Algorithmus ist linear in der Zahl der Rechtecke ($2m+1$) und in der Zahl der Dimensionen d.

Splitalgorithmus im R*-Baum

Idee der R*-Baum Splitstrategie

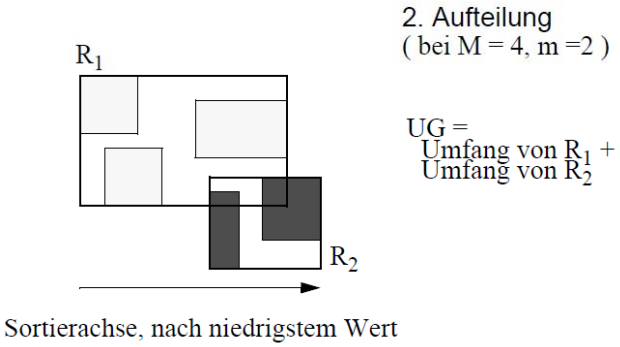
- sortiere die Rechtecke in jeder Dimension nach beiden Eckpunkten und betrachte nur Teilmengen nach dieser Ordnung benachbarter Rechtecke
- Laufzeitkomplexität ist $O(d * M * \log M)$ für d Dimensionen und M Rechtecke

Bestimmung der Splitdimension

- Sortiere für jede Dimension die Rechtecke gemäß beider Extremwerte
- Für jede Dimension:
  - Für jede der beiden Sortierungen werden $M-2m+2$ Aufteilungen der $M+1$ Rechtecke bestimmt, so daß die 1. Gruppe der j-ten Aufteilung die ersten $m-1+j$ Rechtecke und die 2. Gruppe die übrigen Rechtecke enthält
  - UG sei die Summe aus dem Umfang der beiden MURs R_1 und R_2 um die Rechtecke der beiden Gruppen
  - US sei die Summe der *UG* aller berechneten Aufteilungen

⇒ Es wird die Dimension mit dem geringsten *US* als Splitdimension gewählt.

Splitalgorithmus im R*-Baum



Bestimmung der Aufteilung

- Es wird die Aufteilung der gewählten Splitdimension 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 $m = 0{,}4*M$ ergeben

Bulk-Loads im R-Baum

- Vorteile:
  - schneller im Aufbau
  - Struktur ermöglicht i.d.R. schnellere Anfragebearbeitung
- Optimierungskriterien:
  - möglichst hoher Füllungsgrad der Seiten (geringe Höhe)
  - geringe Überlappung
  - wenig toter Raum

Sort-Tile-Recursive:

- Bottom-up Aufbau des R-Baum
- keine Überlappung auf Blattebene bei Punktobjekten
- Zeitkomplexität: O(n log(n))

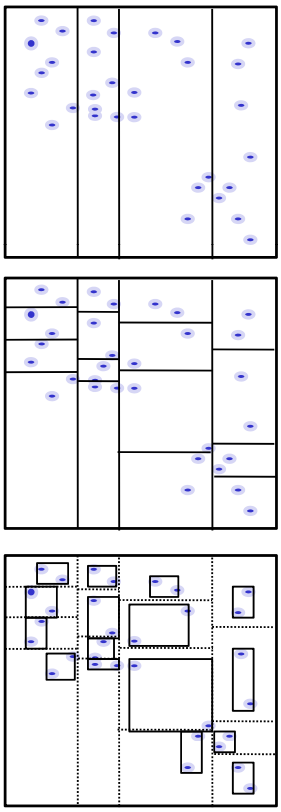
Sort-Tile Recursive

Algorithmus:

1. Setze DB gleich der Objektmenge P mit |P| = n
2. Berechne Anzahl der Quantile: $q = \left\lceil \sqrt{\frac{n}{M}} \right\rceil$
3. Sortiere Datenelemente in Dimension 1
4. bilde Quantile nach jeweils $q*M$ Objekten in Dim. 1
5. Sortiere Objekte in jedem Quantil nach Dimension 2
6. bilde Quantile nach jeweils M Objekten in Dim. 2
7. Bilde eine MUR um die Punkte in jeder Zelle
8. Starte den Algorithmus mit der Menge abgeleiteten MURs oder stoppe falls q < 2
   (alle verbleibenden MURs fallen in die Wurzel)

Anmerkung:

- bei Punkten entstehen überlappungsfreie MUR
- bei MURs kann es zu Überlappungen kommen
- bei MURs ist die Einteilung in Quantile nach Minimalwerten, Maximalwerten oder komplizierteren Heuristiken möglich



Löschen im R-Baum

Das Objekt x ist aus dem R-Baum zu löschen

Löschen :

- Teste ob Seite S nach entfernen von x unterfüllt ist: |S| < m
- Falls nicht entferne x und STOP
- Falls ja bestimme, welche Vorgängerknoten ebenfalls unterfüllt sind.
- Für jede unterfüllten Knoten:
  - Lösche die unterfüllte Seite aus dem Vorgängerknoten
  - Füge die restlichen Elemente der Seite in den R-Baum ein
  - Falls Wurzel nur noch 1 Kind enthält wird Kind zur neuen Wurzel (Höhe verringert sich)

Bemerkungen:

- Löschen ist mit diesem Algorithmus nicht auf einen Pfad beschränkt
- Erfordert das Einfügen eines Teilbaums auf Ebene l in den R-Baum
- Im worst-case sehr teuer

Suchalgorithmen auf Bäumen

Bereichsanfragen:

```
FUNCTION List RQ(q,ε):
List L // Kandidatenliste (MURs/Objekte)
List  Result // Liste aller Objekte in ε-Umgebung von q
L.insert(Wurzel)
WHILE(not L.isEmpty())
        E := L.firstElement()
        IF E.isMUR()
            FOREACH F ∈ E.children()
                IF minDist(F,q) < ε
                    C.insert(F)
        ELSE
            Result.insert(E)
RETURN Result
```

Bemerkung: BOX und Intersection-Queries funktionieren nach demselben Prinzip.

Nächste-Nachbaranfragen

NN-Anfragen: Top-Down Best-First-Search

```
FUNCTION  LIST  NNQuery(q)
PriorityQueue Q // aus MURs/Objekten  sortiert
    // nach minimaler Distanz zu q (minDist)
Object bestResult // bisher gefundener NN
Float minDist //Dist(bestResult,q)
Q.insert(Wurzel)
WHILE(not Q.isEmpty() and Q.topDist() < minDist )
    E := Q.getFirstElement()
    IF E.isMUR()
            prio := minDist(E,q)
            Q.insert(prio,E.children())
    ELSE
       IF Dist(E, q) < minDist
            minDist := D(E,q)
            bestResult := E
RETURN bestResult
```

Spatial Joins

Idee: Definiere Join-Anfragen über räumliche Prädikate

Vorteil: Parallele Verarbeitung vieler Anfragen mit Durchlauf

Beispiel: ε -Range-Join

Seien G und S Mengen räumlicher Objekte mit $G,S \subseteq D$, $dist: D \times D \rightarrow IR$ ein Distanzfunktion und $\varepsilon \in IR$.

Dann heißt

$$S \bowtie_{dist(s,r) < \varepsilon} G = \{(g,s) \in G \times S \mid dist(g,s) < \varepsilon\}$$

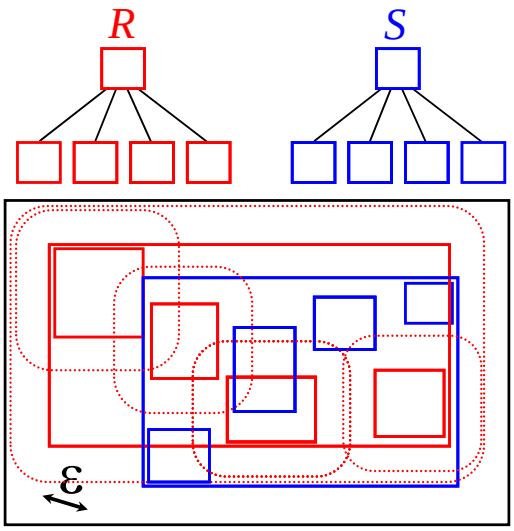
ε-Range-Join von G und S.

Anwendung: BestimmeAoI für alle Spielerentitäten in einem Tick.

R-tree Spatial Join (RSJ)

Algorithmus:

```
FUNCTION rTreeSimJoin (R, S,result,ε)
  IF R.IsDirpg() ∧ S.IsDirpg()
   FOREACH r ∈ R.children()
     FOREACH s ∈ S.children()
       IF minDist(r,s) ≤ ε
          rTreeSimJoin(r,s,result,ε) ;
  ELSE //assume R,S both DataPg
    FOREACH p ∈ R.points
      FOREACH q ∈ S.points
        IF dist(p,q) ≤ ε
           result.insertPair(p,q)
```



Probleme durch Datenvolatilität

Probleme bei räumlicher Fortbewegung aller Objekte:

In Spielen bewegt sich Mehrzahl der Objekte bewegt sich mehrmals pro Sekunde.

- Positions-Änderungen durch Löschen und Einfügen
  - dynamische Änderungen können Struktur negativ beeinflussen
    (Verlust der Balance, Erhöhung von Überlappungen, Überfüllung einer Micro-Zone)
  - Änderung hat hohen Overhead
    (Suche nach Objekt, Folge-Einfügungen, Underflow und Overflow-Behandlungen)
- Positions-Änderung durch spezielle Änderungsoperationen
  - Ausdehnen von Seitenregionen: Overlap der Seiten kann extrem ansteigen
    (nur bei Datenpartitionierung möglich)
  - Wechsel der Seitenregionen:
    - *Balance kann negativ beeinflusst werden*
    - *Unterfüllung und Überfüllung von Seiten*

Fazit: Dynamische Änderungen sind entweder aufwendig in der Berechnung oder können die Organisation der Daten ungünstig für Anfragealgorithmen beeinträchtigen.

Throw-Away Indices

Idee:

- Bei sehr volatilen Daten ist die Änderung vorhandener Datenstrukturen teurer als ein Neuaufbau mittels Bulk-Load.
- Ähnlich wie beim Game-State existieren immer 2 Indexstrukturen:
  - Index I1 organisiert Positionen des letzen konsistenten Ticks und wird zur Anfragebearbeitung genutzt
  - Index I2 wird parallel dazu I1 aufgebaut:
    - Aufbau per Bulkload: geringere Nebenläufigkeit, aber schneller Aufbau, gute Struktur
    - Dynamischer Aufbau: mehr Rechenaufwand und evtl. schlechtere Struktur, aber Aufbau bei jeder neuen Position möglich
  - bei Beginn des neuen Ticks wird auf I2 angefragt und I1 gelöscht und anschließend neu aufgebaut.

Fazit: Entscheidend ist das der Tick mit Aufbau des Baumes schneller verarbeitet wird als ohne unterstützende Datenstruktur.

(Abhängigkeit vom Game-Design)

Game Design

die räumliche Problematik hängt stark vom Spiel-Design ab:

- Anzahl und Verteilung der räumlichen Objekte
- Anzahl und Verteilung der Spieler
- Umgebungsmodell Felder, 2D oder 3D
  (3D Umgebung macht 3D Indexing nicht unbedingt notwendig)
- Bewegungsgeschwindigkeit und Bewegungsart der Objekte

Lernziele

- Game State und Game Entities
- Aktionen und Zeitsteuerung
- Game Loop und Synchronisation mit anderen Subsystemen
- typische Verarbeitungsschritte in einer Iteration
- Zusammenhang Scripting-Engine, Physics Engine und Spatial Management
- Zoning, Sharding und Instanziierung
- Micro-Zoning und Spatial-Publish-Subscribe
- BSP-Tree, KD-Tree, Quad-Tree und R-Tree
- Einfügen, Löschen, Bulk-Load
- Anfragebearbeitung: Range-Query,NN-Query und Range-Join
- Probleme bei hoch volatilen Daten

Literatur und Material

- Shun-Yun Hu, Kuan-Ta Chen
  VSO: Self-Organizing Spatial Publish Subscribe
  In 5th IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2011, Ann Arbor, MI, USA, 2011.
- Jens Dittrich, Lukas Blunschi, Marcos Antonio Vaz Salles
  Indexing Moving Objects Using Short-Lived Throwaway Indexes
  In Proceedings of the 11th International Symposium on Advances in Spatial and Temporal Databases, 2009.
- Hanan Samet. 2005. **Foundations of Multidimensional and Metric Data Structures** *(The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.