3. Abbildung auf das relationale Datenmodell

1. Einführung

2. Spaghetti-Modell

3. TIGER-Modell

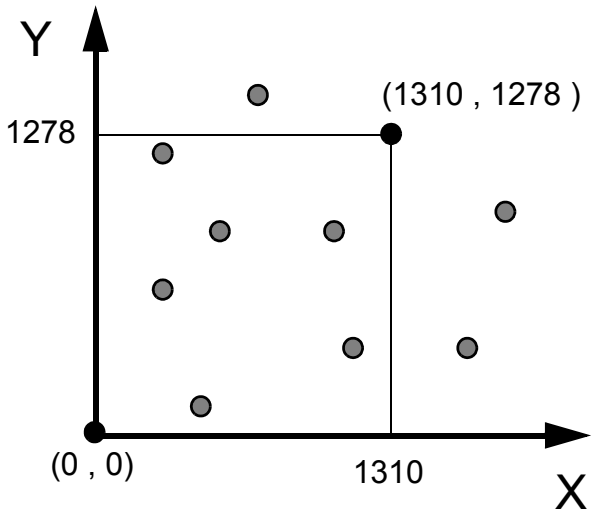
4. Schwächen des relationalen Modells

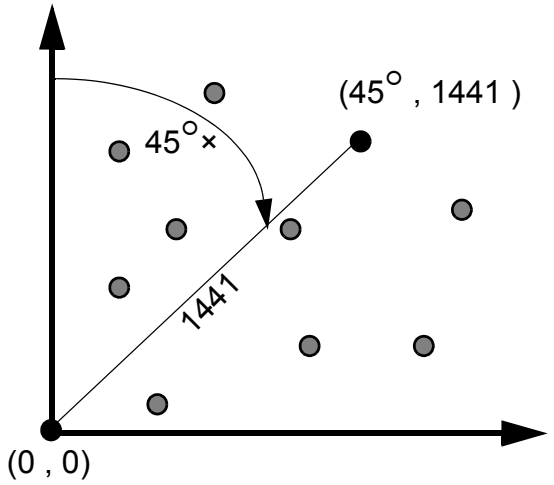
3.1 Punkt (I)

- Träger der geometrischen Information, auf dem die anderen Grundelemente aufbauen
- Die Lage eines Punktes im Datenraum wird über *Koordinaten* bezüglich eines *Koordinationssystems* bestimmt
  - Kartesisches Koordinatensystem (Distanzen in Richtung der Koordinatenachsen)

    z.B. x = 1310, y = 1278
  - Polarkoordinatensystem (Winkel und Distanz vom Ursprung)

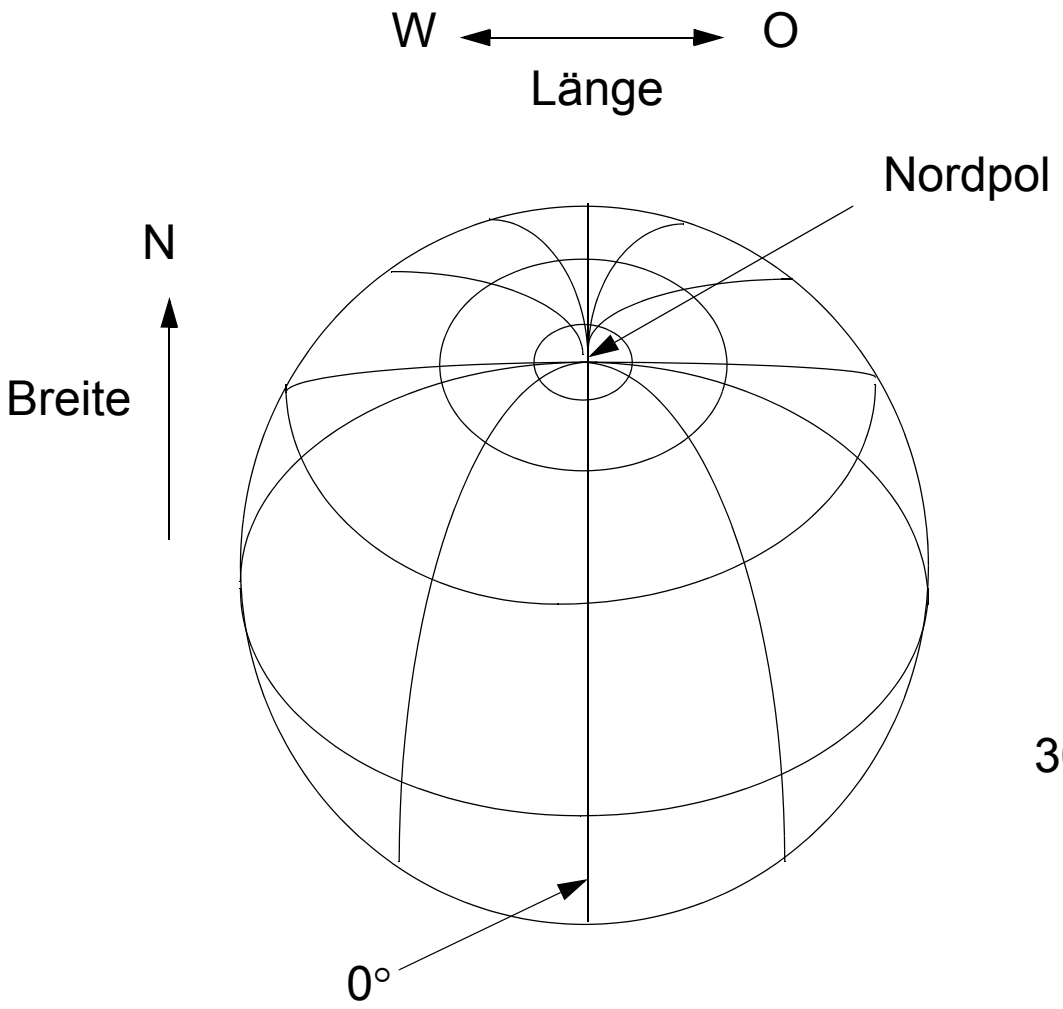
    z.B. φ = 45°, r = 1441

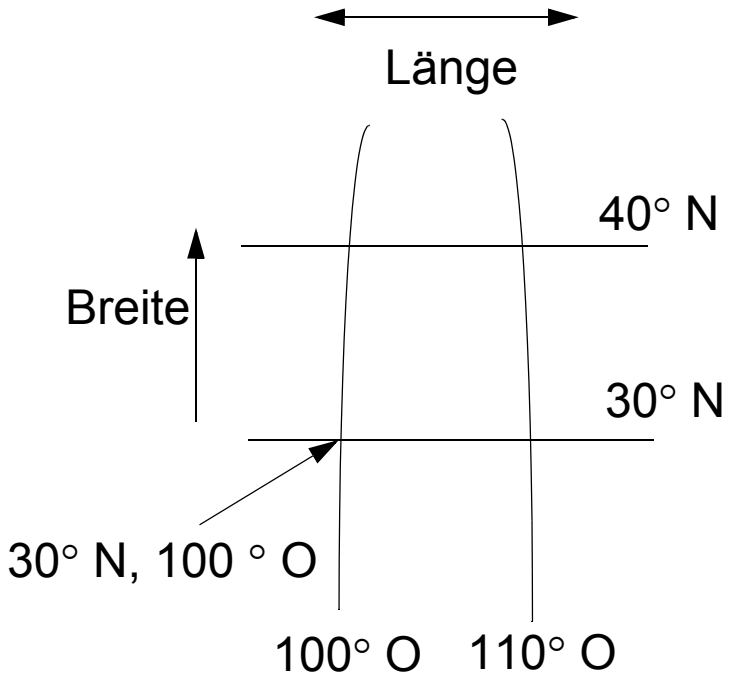




3.1 Punkt (II)

- Geographisches Koordinatensystem (Einteilung in Längen- und Breitengrade)



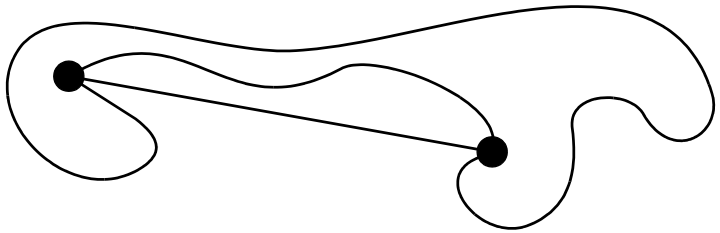


3.1 Punkt (III)

- Der Datenraum ist ein *metrischer Raum*:
  - Zwischen zwei Punkten P_1 und P_2 ist eine (nicht-negative) *Distanz d* (Metrik) definiert, so daß
    - $d(P_1,P_2) = 0 \Leftrightarrow P_1 = P_2$ (Identität)
    - $d(P_1,P_2) = d(P_2,P_1)$ (Symmetrie)
    - $d(P_1,P_2) \leq d(P_1,P_3) + d(P_3,P_2)$ (Dreiecks-Ungleichung)
  - z.B. Euklidische Distanz: $d((x_1,y_1),(x_2,y_2)) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$
  - z.B. Manhattan Distanz: $d((x_1,y_1),(x_2,y_2)) = |x_1 - x_2| + |y_1 - y_2|$
- Es können weitere vermessungstechnische Informationen zugeordnet sein:
  Höhe, Nummer, Art der Erfassung, Genauigkeit, Datum der Erfassung, ...

3.1 Linien

- als topologische Eigenschaft: *Kante*
  - direkte Verbindung zwischen zwei Punkten
  - alle direkten Verbindungen zwischen zwei Punkten sind topologisch äquivalent



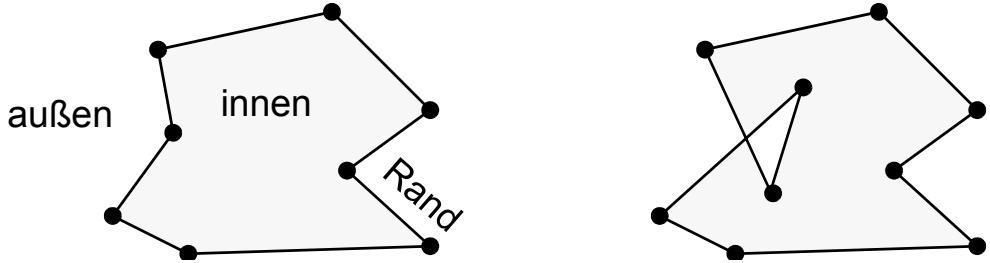
  - topologisch äquivalent: die relative Lage zweier Objekte zueinander ist gleich

- als geometrische Eigenschaft: *Strecke* oder *Streckenzug*
  - *Strecke*: geradlinige Verbindung zwischen zwei Punkten,
    Streckenzug: Folge von Strecken
  - alternativ: Kurven (z.B. Kreisbögen, Bézier-Kurven)

3.1 Fläche

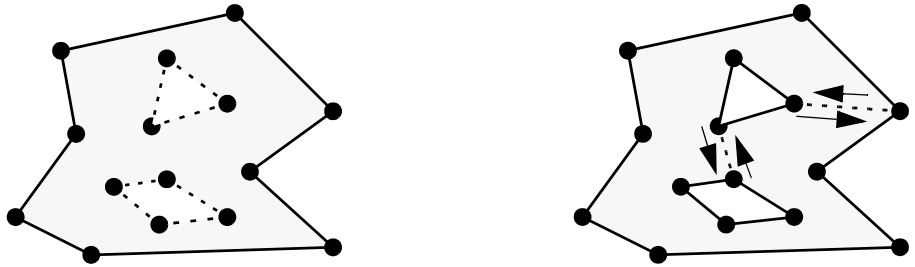
Fläche ohne Löcher

- über geschlossenen Kanten- / Streckenzug definiert
  - *Polygon:* geschlossener Streckenzug
  - *einfaches Polygon:* Strecken sind bis auf die Eckpunkte überlappungsfrei



Fläche mit Löchern

- Modellierungsalternativen:
  - über ein *äußeres Polygon* und 0, 1 oder mehrere *innere Polygone*
  - mit Hilfe von *Pseudokanten*



3.2 Spaghetti-Modell

- ist ein einstufiges Vektormodell
- Jedes Objekt wird als Folge ausgezeichneter (x ,y) - Koordinaten (z.B. der Eckpunkte) definiert. Die Repräsentation zweier Objekte ist unabhängig voneinander (d.h. gleiche Punkte werden mehrfach gespeichert).

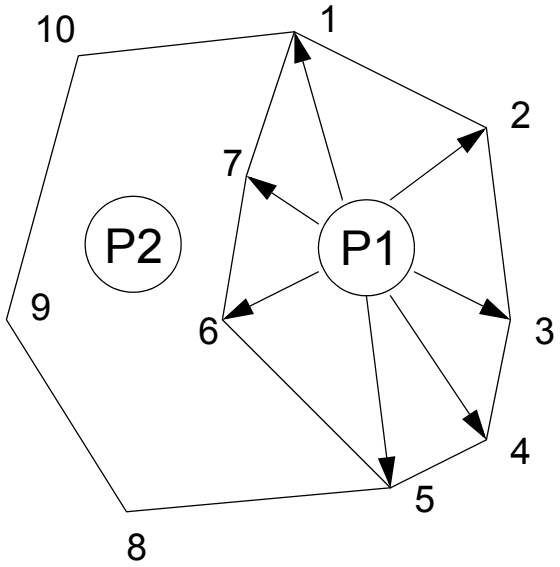
- Objektklassen
  - Punkt
  - Streckenzug
  - einfaches Polygon

- Eigenschaften:
  - \+ **einfach**
  - – keine explizite topologische Information

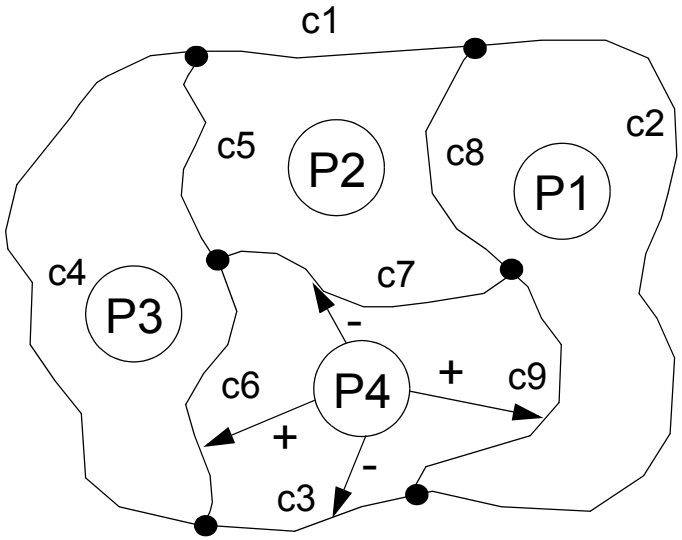
    (insbesondere keine Löcher in Polygonen explizit modellierbar)
  - – starke Redundanz

3.3 Zweistufige Vektormodelle

- mit *Koordinatenreferenzierung*
  - Polygon: geordnetes Verzeichnis von Punktreferenzen
- mit *Streckenzugreferenzierung*
  - Polygon: geordnetes Verzeichnis von Steckenzugreferenzen (Orientierungsinformation notwendig)



Koordinatenreferenzierung



Streckenzugreferenzierung

- Polygon über Verzeichnis repräsentiert (Datenstruktur variabler Größe)
- keine Löcher, keine Nachbarschaftsinformation modelliert

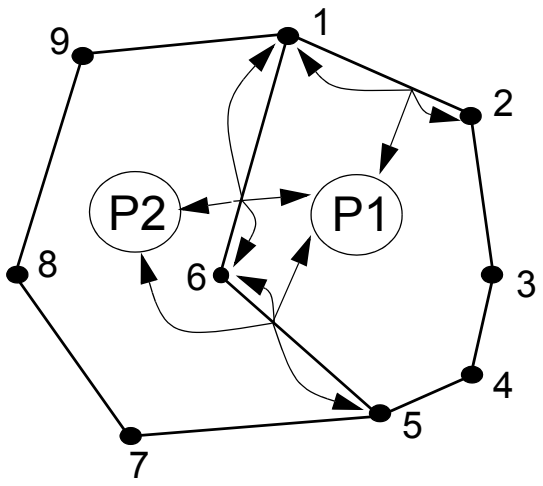
3.3 Topologische Vektormodelle (I)

- explizite Speicherung der topologischen Struktur
  - $\Rightarrow$ einfache Bestimmung von benachbarten Objekten
  - $\Rightarrow$ höherer Aufwand bei Änderungen
  - $\Rightarrow$ höherer Speicheraufwand

DIME

- *D*ual *I*ndependent *M*ap *E*ncoding System (U.S. Bureau of Census, 1970)
- Grundelement: Kante

  hat vier Zeiger (Anfangs- und Endpunkt, links und rechts liegende Fläche)



3.3 Topologische Vektormodelle (II)

TIGER-Modell

- *T*opologically *I*ntegrated *G*eographic *E*ncoding and *R*eferencing Model (Marx, 1986)

- Datentypen:
  - 0-Zellen: Knoten: Endpunkte von Steckenzügen, Schnittpunkte
  - 1-Zellen: Kante: Verbindung zweier 0-Zellen
  - 2-Zellen: Fläche, die durch Verknüpfung von 1-Zellen beschrieben werden kann (geschlossener Kantenzug)
  - Cover: Vereinigung von 2-Zellen

- Organisation: Listen für 0-, 1- und 2-Zellen, Directories für 0-Zellen und Cover

- 0-Zelle
  - besteht aus ihren Koordinaten, verschiedenen Attributen und *einem* Verweis zu einer 1-Zelle
- 2-Zelle
  - besteht insbesondere aus verschiedenen Attributen und *einem* Verweis zu einer 1-Zelle

3.3 Topologische Vektormodelle (III)

TIGER-Modell (Fortsetzung)

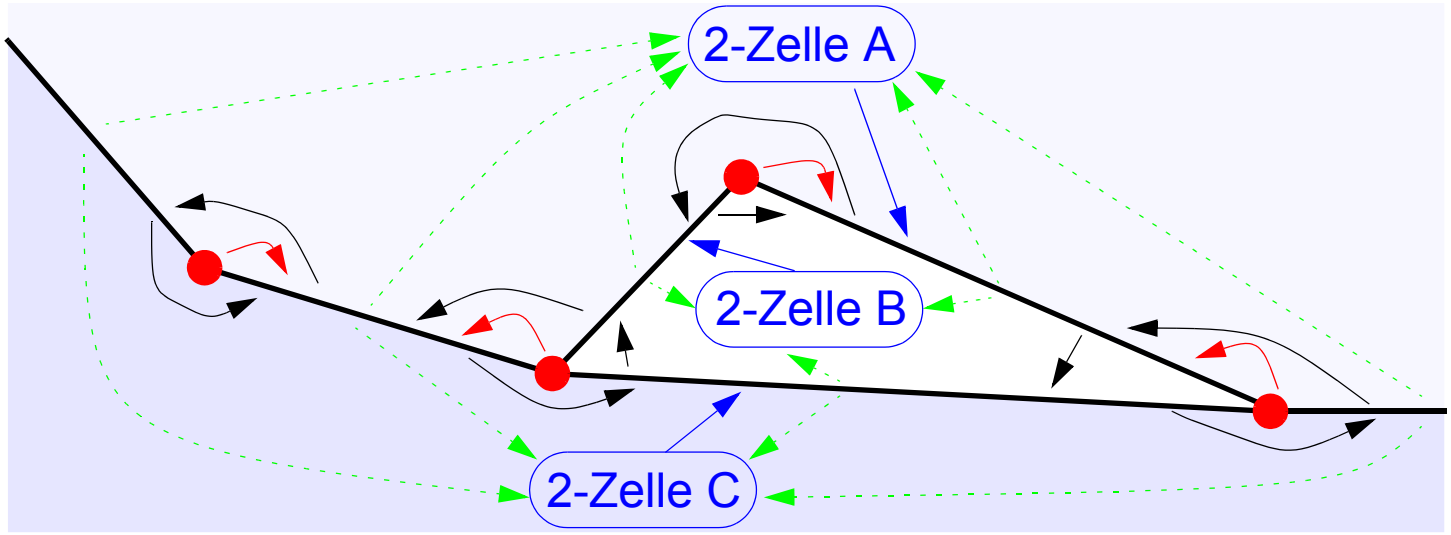
- 1-Zelle (Kante)
  - zentrale Struktur: verbindet 0-Zellen und umschließt 2-Zellen
  - besteht insbesondere aus

    zwei Verweisen auf 0-Zellen (Anfangs- und Endpunkt)

    zwei Verweisen auf 2-Zellen (linke und rechte Fläche)

    je *einen* Verweis auf die nächste 1-Zelle, die zu der entsprechenden 2-Zelle gehört

    einem Verweis auf eine Beschreibungsliste (für interne Punkte)



⇒ Datenstrukturen fester Größe !

3.3 Topologische Vektormodelle (IV)

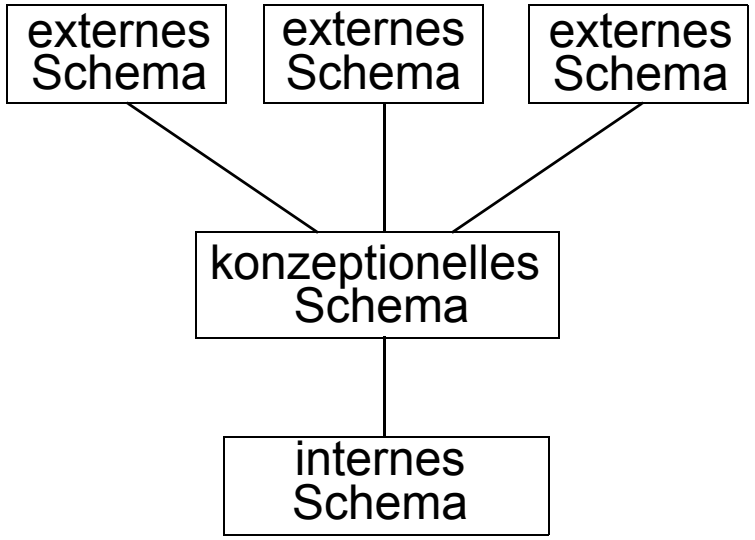
Äquivalentes Modell

- Knoten
  - Verweise zu *allen* Kanten (umlauforientiert)
  - Koordinaten (Geometrie)
- Kante (auch geschlossen)
  - zwei Verweise auf Eckknoten
  - zwei Verweise auf benachbarte Flächen
  - Verweis auf eine Beschreibungsliste (für interne Punkte, Geometrie)
- Fläche (mit Löchern)
  - Verweise zu *allen* umgebenden Kanten (ablauforientiert)
  - Verweise zu *allen* eingeschlossenen Kanten (Menge, ablauforientiert)
  - Verweis auf Cover
- Cover
  - Verweise auf Flächen

$\Rightarrow$ Datenstrukturen variabler Größe

3.4 Integration in ein relationales DBS

Ebenen eines DBS



unterschiedliche Sichten verschiedener Benutzer / Anwenderprogramme auf die Daten

logische Gesamtsicht aller Daten

physische Datenorganisation

- externe und konzeptionelle Ebene
  - logisches Datenmodell
  - Anfragesprache
- interne Ebene
  - physische Datenrepräsentation
  - Anfragebearbeitung und -optimierung

3.4 Schwächen auf konzeptioneller Ebene

Schema für mehrstufiges Vektormodell

Parzellen

OID	PolID	Q-Meter
Par1	Pol1	1200
Par2	Pol2	1435

Polygone

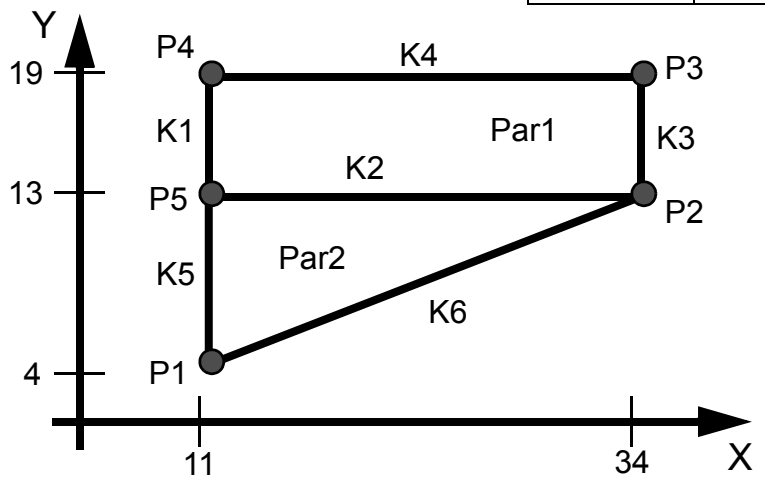
PolID	KID
Pol1	K1
Pol1	K2
Pol1	K3
Pol1	K4
Pol2	K2
Pol2	K5
Pol2	K6

Kanten

KID	Von	Nach
K1	P4	P5
K2	P5	P2
K3	P2	P3
K4	P3	P4
K5	P5	P1
K6	P1	P2

Punkte

PID	x	y
P1	11	4
P2	34	13
P3	34	19
P4	11	19
P5	11	13



3.4 Schwächen auf konzeptioneller Ebene

Anfragen an dieses Schema

Bestimme alle Koordinaten der Parzelle Par1:

```
SELECT Punkte.x, Punkte.y
FROM Parzellen, Polygone, Kanten, Punkte
WHERE (Parzellen.OID=Par1) AND
        (Parzellen.PolID = Polygone.PolID) AND
        (Polygone.KID = Kanten.KID) AND
        (Kanten.Von = Punkte.PID);
```

Probleme

- hoher Aufwand für Zugriff auf alle Komponenten eines Polygons (Joins)
- Antwort auf Anfrage ist unsortiert (kein “Objekt” Polygon)

  $\Rightarrow$ Objektorientiertes Datenmodell

3.4 Objektorientiertes Datenmodell (I)

Objekte

- Objekte sind vom Benutzer definierte Entities, die aus Werten oder anderen Objekten aufgebaut sind (Attribute).
- Attribute können listen- oder mengenwertig sein.

Objekttypen

- Ein Objekttyp (“Klasse”) beschreibt die Charakteristika einer Menge ähnlicher Objekte.
- Ein Objekt ist eine konkrete Instanz seines Objekttyps, hat also dessen Charakteristika.

```
type Vertex is
    public . . .
    body [x, y, z: float;]
end type Vertex;

type Cuboid is
    public . . .
    body [mat: Material;
          value: float;
          vertices: <Vertex>;]// Typ: Liste von Vertices
end type Cuboid;
```

3.4 Objektorientiertes Datenmodell (II)

Operationen

- Jeder Objekttyp bietet an seiner Benutzerschnittstelle eine Menge von Operationen an, mit deren Hilfe Instanzen dieses Typs manipuliert werden können.
- Neben diesen Operationen gibt es keine andere Möglichkeit, von außen den internen Zustand eines Objekts abzufragen oder zu ändern.

```
type Vertex is
    public translate, scale, rotate, distance, inOrigin, . . .    // Benutzerschnittstelle
    body [x, y, z: float;]
    operations
        declare translate: Vertex → void;                         // Änderungsfunktion
        declare scale: Vertex → void;                             // Änderungsfunktion
        declare rotate: float, char → void;                       // Änderungsfunktion
        declare distance: Vertex → float;                         // Beobachterfunktion
        declare inOrigin: → bool;                                 // Beobachterfunktion
        . . .
    implementation
    . . .
end type Vertex;
```

3.4 Objektorientiertes Datenmodell (III)

Modellierung von Spatial Data Types

```
type Vertex is
    public distance, contained_in, . . .
    body [x, y: float;]
    . . .
end type Vertex;
```

```
type Edge is
    public . . .
    body [begin, end: Vertex;]
    . . .
end type Edge;
```

```
type Polygon is
    public intersect, . . .
    body [boundary: <Edge>;
        . . .]
end type Polygon;
```

```
type Region is                    // Parzelle
    public . . .
    body    [name: string;
             extension: Polygon; area: float;]
end type Region;
```

Bestimme alle Koordinaten der Parzelle Par1:

```
select x, y
from    (select begin
          from (select r.extension.boundary
                from r in Region
                where r.name = “Par1”))
```

3.4 Objektorientiertes Datenmodell (IV)

Vorteile

- Spatial Data Types können definiert werden, die in Attributen referenziert werden können.
- Die Modellierung von Eigenschaften variabler Kardinalität ist einfach mit Hilfe listen- bzw. mengenwertiger Attribute.
- Logisch zusammengehörige Objekte werden physisch benachbart gespeichert (Clustering), so daß das Einlesen / Ändern eines Geo-Objektes relativ effizient möglich ist.
- In den Anfragen können die Operationen der benutzerdefinierten Objekttypen verwendet werden.
  - $\Rightarrow$ Geo-Objekte als Objekte modellierbar
  - $\Rightarrow$ effizienter Zugriff auf ein Geo-Objekt

Schwächen

- Zugriff auf mehrere benachbarte Geo-Objekte ist noch nicht effizient unterstützt (Erweiterung analog TIGER-Modell möglich)

3.4 Schwächen auf interner Ebene

Aufgaben der internen Ebene

- physische Datenrepräsentation
- Anfragebearbeitung und -optimierung
- ...

Ziele

- dauerhafte (persistente) Speicherung der Daten
  - $\Rightarrow$ Einsatz von Magnetplattenspeichern etc.
- effiziente Bearbeitung von thematischen und geometrischen Anfragen, von Kombinationen, von topologischen Operationen usw.
- Kriterien für Effizienz:
  - Antwortzeit für einzelne Anfragen
  - Durchsatz über alle Anfragen
- dynamisches Einfügen, Löschen und Verändern von Daten
- hohe Speicherplatzausnutzung

3.4 Schwächen auf interner Ebene

Physische Datenspeicherung

- seitenorientiert
  - Seite: Transfereinheit zwischen Haupt- und Sekundärspeicher
  - Wahlfreier (direkter) Zugriff
  - Feste Größe: zwischen 512 Byte und 8 KByte
- Zugriff über einen Kamm mit Schreib-/Leseköpfen, der quer zu sich rotierenden Magnetplatten bewegt wird
  - Positionierung des Schreib-/Lesekopfes [6 msec]
  - Warten auf Seite [4 msec]
  - Kontrolle + Übertragung der Seite [1 msec + 0,2 msec / 2KByte Seite]
- Entwicklung:
  - Kapazität von Plattenspeichern erhöht sich drastisch
  - Zugriffszeiten sinken relativ langsam

$\Rightarrow$ Zugriffe auf Sekundärspeicher sind verglichen mit Hauptspeicheroperationen sehr zeitaufwendig

$\Rightarrow$ Minimierung der Anzahl der Seitenzugriffe

3.4 Konventionelle Indexstrukturen (I)

B-Bäume

- Ein Baum heißt *Suchbaum*, wenn für jeden Eintrag in einem inneren Knoten alle Schlüssel im linken Teilbaum kleiner sind und alle Schlüssel im rechten Teilbaum größer sind als der Schlüssel im Knoten.
- Ein *B-Baum der Ordnung m* ist ein Suchbaum mit folgenden Eigenschaften:

(1) Jeder Knoten enthält höchstens 2*m* Schlüssel.

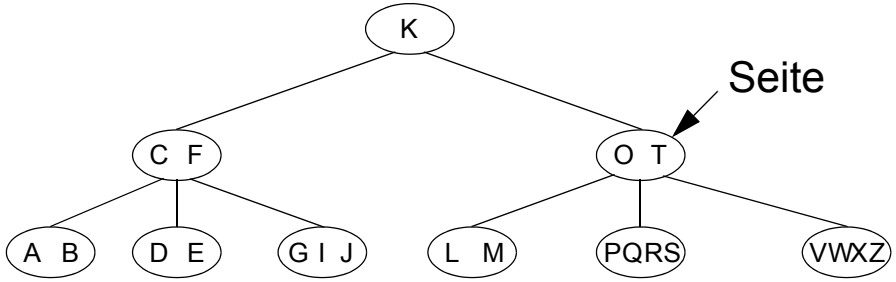
(2) Jeder Knoten außer der Wurzel enthält mindestens *m* Schlüssel.

(3) Die Wurzel enthält mindestens einen Schlüssel.

(4) Ein Knoten mit *k* Schlüsseln hat genau *k*+1 Söhne.

(5) Alle Blätter befinden sich auf demselben Level.

- B-Baum der Ordnung 2



- Höhe eines B-Baums für n Schlüssel ist O(log n)

3.4 Konventionelle Indexstrukturen (II)

Leistung

- Einfügen, Löschen und Punktanfrage im B-Baum sind auf einen Pfad beschränkt.
- Zeitkomplexität dieser Operationen O(log n)

Weitere Ziele

- *Sequentielles Auslesen aller Datensätze*, die von einem B-Baum organisiert werden.
- *Unterstützung von Bereichsanfragen* der Form:

  "Nenne mir alle Studenten, deren Nachname im Bereich [Be ... Brz] liegt."

Idee

- Trennung der Indexstruktur in *Directoryknoten* (innere Knoten) und *Datenknoten (*Blätter*)*.
- *Sequentielle Verkettung* der Datenknoten.

B^+-Baum

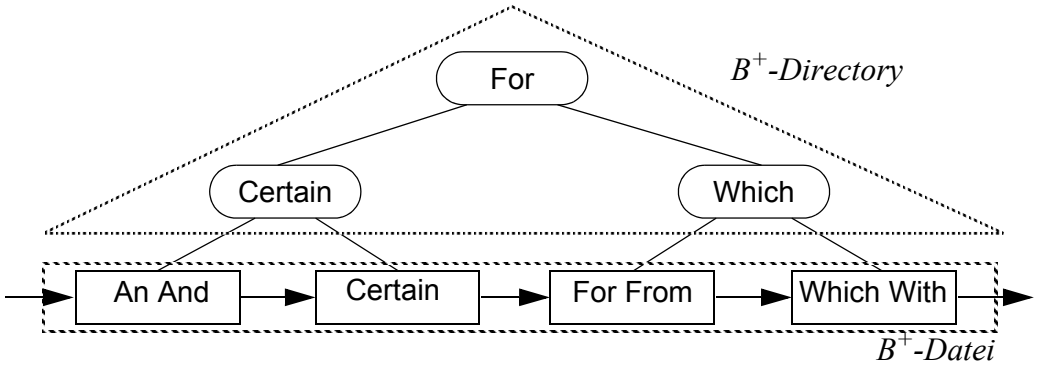
- Die Datenknoten enthalten alle Datensätze.
- Directoryknoten enthalten nur noch *Separatoren* s.

3.4 Konventionelle Indexstrukturen (III)

Separatoreigenschaft

- Für jeden Separator s(u) eines Knotens u gelten folgende *Separatoreneigenschaften*:
  - s(u) > s(v)für alle Directoryknoten v im linken Teilbaum von s(u).
  - s(u) < s(w)für alle Directoryknoten w im rechtenTeilbaum von s(u).
  - s(u) > k(v')für alle Primärschlüssel k(v') und alle Datenknoten v' im linken Teilbaum von s(u).
  - s(u) ≤ k(w')für alle Primärschlüssel k(w') und alle Datenknoten w' im rechten Teilbaum von s(u).

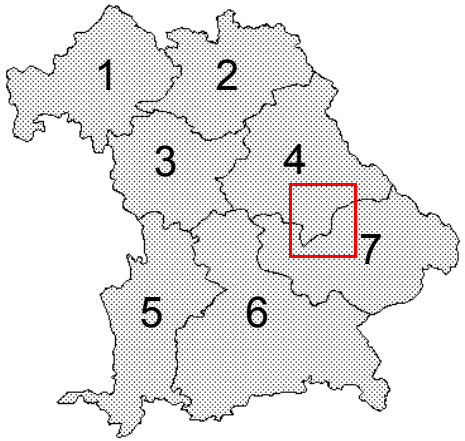
Beispiel: B^+-Baum für Zeichenketten .



3.4 Konventionelle Indexstrukturen (IV)

B-Bäume für Geo-Objekte

- Lineare Ordnung des 2D Raums nötig
- Beispiel



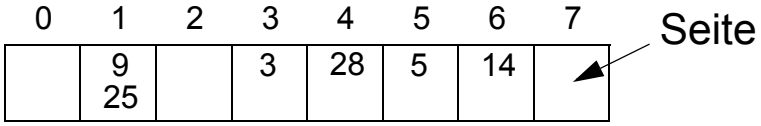
- im 2D benachbarte Objekte werden häufig miteinander angefragt
- es gibt keine lineare Ordnung, die alle Nachbarschaften des 2D-Raums erhält

$\Rightarrow$ die Antworten auf eine Fensteranfrage sind über den ganzen Index verteilt

3.4 Konventionelle Zugriffsstrukturen (V)

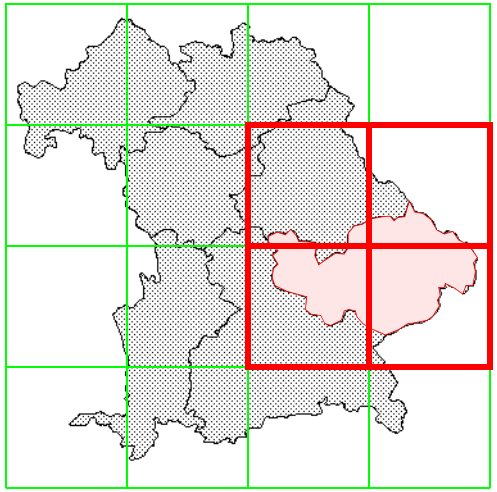
Hashverfahren

- Zuordnung der Objekte zu Einträgen einer Hashtabelle



Hashfunktion: Schlüssel MOD 8

- Voraussetzung: Definition einer Hashfunktion mit möglichst wenigen Kollisionen



⇒ im 2D benachbarte Objekte auf dieselbe bzw. benachbarte Adresse abbilden

⇒ ähnliche Probleme wie bei B-Bäumen