

Teil VII

Indexstrukturen für Data Warehouse

Indexstrukturen für Data Warehouse

- 3 Klassifikation von Indexstrukturen
- 4 B-Bäume
- 5 Bitmap-Indexe
- 6 Mehrdimensionale Indexstrukturen

Motivation

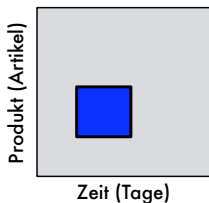
- Faktentabellen in Data Warehouses sind so groß, dass ein Full-Table-Scan nicht sinnvoll ist
- Beispiel: Scan über 10 GB-Tabelle bei 10 MB/s = ca. 17 Minuten
- Anfragen betreffen typischerweise nur einen verhältnismäßig kleinen Anteil der vorhandenen Daten:
 - ▶ Je nach Beschränkung der einzelnen Dimensionen umfasst die Antwort nur wenige Prozent oder Promille (oder noch weniger) aller Daten
- Verwendung von Indexstrukturen ist sinnvoll, um die Anzahl der zu lesenden Datenseiten zu minimieren

Klassifikation von Indexstrukturen

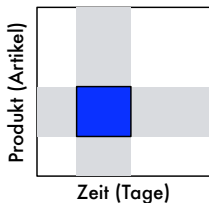
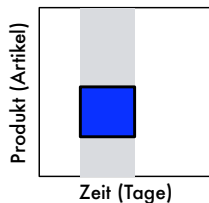
- **Clustering**: Daten, die voraussichtlich oft zusammen verarbeitet werden, werden auch physisch nahe beieinander abgelegt
 - ▶ *Tupel-Clustering*: Ablegen von Tupeln auf der gleichen physischen Seite
 - ▶ *Seiten-Clustering*: Hintereinanderablegen zusammengehöriger Seiten im Sekundärspeicher (erlaubt *Prefetching*)
- **Dimensionalität**: Angabe, wieviele Attribute (Dimensionen) der zugrundeliegenden Relation für Berechnung des Indexschlüssels verwendet werden
- **Symmetrie**: Wenn die Performance unabhängig von der Reihenfolge der Indexattribute ist, handelt es sich um eine symmetrische Indexstruktur, ansonsten um eine asymmetrische
- **Tupelverweise**: Art der Verweise innerhalb der Indexstruktur auf die Tupel
- **Dynamisches Verhalten**: Aufwand zur Aktualisierung der Indexstruktur bei Einfügen, Ändern und Löschen; (sowie ggf. Problem der „Entartung“)

Vergleich von Indexstrukturen

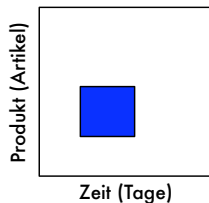
Full Table Scan



Clusternder Primärindex



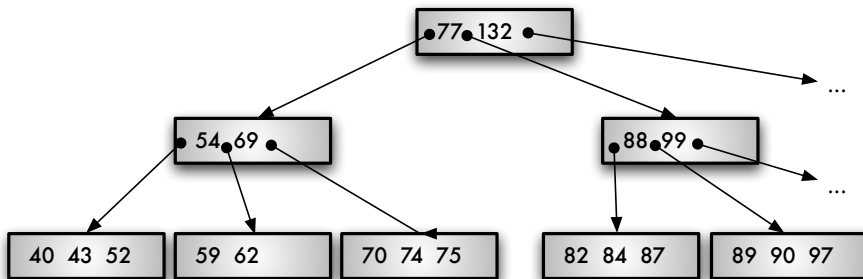
Mehrere Sekundärindexe, Bitmap-Indexe



Multidimensionaler Index

Eindimensionale Baumstrukturen

- B-Baum [Bayer/McCreight 1972]



B-Baum

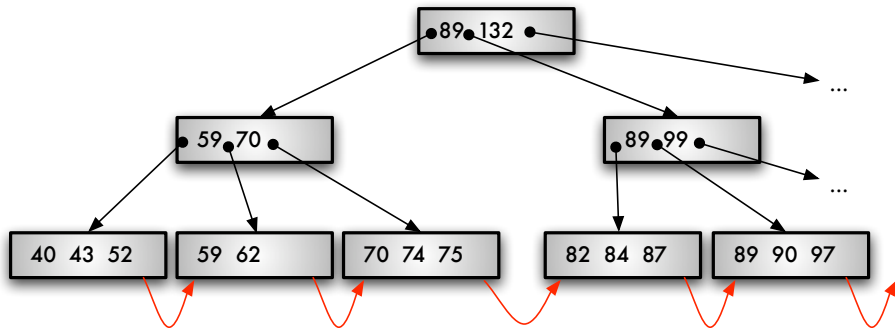
- Ordnung eines B-Baumes: min. Anzahl der Einträge auf den Indexseiten (außer Wurzelseite)
- Definition: Indexbaum ist B-Baum der Ordnung m , wenn
 - ▶ Jede Seite höchstens $2m$ Elemente enthält
 - ▶ Jede Seite außer Wurzelseite mind. m Elemente enthält
 - ▶ Jede Seite entweder Blattseite ohne Nachfolger ist oder $i + 1$ Nachfolger (i : Anzahl ihrer Elemente) hat
 - ▶ Alle Blattseiten auf der gleichen Stufe liegen

B-Baum: Eigenschaften

- n Datensätze in Hauptdatei
 - $\log_m(n)$ Seitenzugriffe von der Wurzel zum Blatt
- Balancierungskriterium führt zu Eigenschaft nahe vollständiger Ausgeglichenheit
- Einfügen, Löschen, Suchen mit $O(\log_m(n))$
- Speicherplatzausnutzung: mindestens 50% (ausser Wurzel)

B^+ -Baum

- B^+ -Baum (Variante des B -Baums): Tupel/TIDs nur in den Blättern; Blätter untereinander verkettet für sequenziellen (Bereichs-) Durchlauf



Eigenschaften von B - und B^+ -Bäumen

- Eindimensionale Struktur (Index über ein Attribut)
- Als Primärindex können Tupel auch direkt im Baum gespeichert werden (ermöglicht einfaches Clustering; insb. beim B^+ -Baum)
- Als Sekundärindex werden nur TIDs im Baum gespeichert
- Balancierte Bäume (Weg von Wurzel zu Blatt überall gleich lang); Balancierung erfordert etwas aufwendiger Reorganisation bei Änderung der Daten
 - ▶ Für Data Warehouses von untergeordneter Bedeutung
- B^+ -Baum besonders geeignet für Bereichsanfragen (durch Verkettung auf Blattebene)

Anwendung von B -/ B^+ -Baum

- B - und B^+ -Baum: eindimensionale Strukturen:
 - ▶ Nur unzureichende Unterstützung mehrdimensionaler Anfragen
- Mögliche Anwendung bei mehrdimensionalen Anfragen
 - ▶ Für jedes beteiligte Attribut gibt es einen B - oder B^+ -Baum als Sekundärindex
 - ▶ Dann kann für jedes Attribut entsprechend der Anfragerestriktion die Menge der TIDs der Tupel bestimmt werden, die die jeweilige Restriktion erfüllen
 - ▶ Nun wird der Schnitt dieser unabhängig voneinander bestimmten TID-Mengen berechnet; die zugehörigen Tupel bilden das Anfrageergebnis

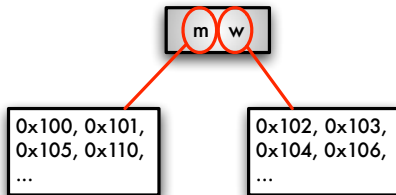
B^+ -Baum: Fazit

- Robuste, generische Datenstruktur
- Unabhängig vom Datentyp (nur Ordnung erforderlich)
- Effiziente Aktualisierungsalgorithmen
- Kompakt
- „Arbeitspferd“ aller RDBMS
- Probleme
 - ▶ Attribute mit geringer Kardinalität → **degenerierte Bäume**
 - ▶ Zusammengesetzte Indexe → **ordnungssensitiv**

Degenerierte B-Bäume

- Beispiel:
 - ▶ Tabelle `customer`
 - ▶ Attribute: u.a. `gender` (`m`, `w`)
 - ▶ Index

```
CREATE INDEX s_idx ON customer(gender)
```



Ordnungsabhängigkeit

- Zusammengesetzter Index
 - ▶ Indexierung concatenierter Attributwerte
 - ▶ Problem: Ordnung der Anfrageprädikate anpassen
- Beispiel:
 - ▶ Tabelle customer
 - ▶ Attribute: gender, profession, cclass
 - ▶ Index:

```
CREATE INDEX csp_idx
  ON customer(cclass, gender, profession)
```

- ▶ Anfragen:

```
SELECT ... FROM ...
  WHERE gender='m' AND profession='lecturer'
  AND cclass=1
SELECT ... FROM ...
  WHERE cclass=1 AND gender='m'
  AND profession='lecturer'
```

B^+ -Baum-Tricks: Oversized Index

- Anfrage:

```
SELECT AVG (age)
FROM customer
WHERE cclass=1 AND gender='m'
      AND profession='lecturer'
```

- Indexnutzung

- ▶ Suchen des Wertes „1||m||lecturer“
- ▶ Zugriff auf Block der Relation `customer` über TID für Wert von `age`

- Besser:

```
CREATE INDEX csp_idx
ON customer(cclass, gender, profession, age)
```

B^+ -Baum-Tricks: Berechnete Indexe

- Berechnung von indexierten Werten durch Angabe einer Funktion
- Beispiel: Index über `customer(name)`

▶ Anfrage:

```
SELECT * FROM customer
WHERE name="Müller" OR name="müller"
      OR name="Mueller"...
```

▶ Indexnutzung nicht möglich

- Besser

```
CREATE INDEX n_idx ON customer(upper(name))
CREATE INDEX n_idx ON customer(soundex(name))
```

Oracle9i: Spezielle Features

- **Index Skip Scan**: nutzt zusammengesetzte Indexe auch, wenn erstes Attribut nicht in Bedingung enthalten ist
- Beispiel:
 - ▶ Tabelle `customer`, Index auf `(state, registration#)`
 - ▶ Index nutzbar für Anfrage mit

```
... WHERE registration# = 4245
```

- Durchsuchung des sekundären Index für alle **DISTINCT** Werte des ersten Attributs
- Meist nur bei niedriger Kardinalität des ersten Attributes sinnvoll
- Infos: <http://otn.oracle.com/products/oracle9i/daily/apr22.html>

Oracle9i: Spezielle Features (2)

- Index-organized Tables
 - ▶ Tupel werden im B^+ -Baum mitgespeichert
 - ▶ Keine Indirektion über TID notwendig
- User-defined Indexes
 - ▶ Implementierung eigener Indexstrukturen für benutzerdefinierte Datentypen
 - ▶ Transparente Nutzung
 - ▶ Angabe eigener Kostenabschätzungen

Bitmap-Indexe

- Idee: **Bit-Array** zur Kodierung der Tupel-Attributwert-Zuordnung
- Vergleich mit baumbasierten Indexstrukturen:
 - ▶ Vermeidet degenerierte B-Bäume
 - ▶ Unempfindlicher gegenüber höherer Zahl von Dimensionen
 - ▶ Einfachere Unterstützung von Anfragen, in denen nur einige (der indexierten) Dimensionen beschränkt werden
 - ▶ Dafür aber i.allg. höhere Aktualisierungskosten
 - ★ In Data Warehouses wegen des überwiegend lesenden Zugriffs unproblematisch

Bitmap-Index: Realisierung

- Prinzip: Ersetzung der TIDs (rowid) für einen Schlüsselwert im B^+ -Baum durch Bitliste
- Knotenaufbau:

Bestellstatus		
$F : 010010 \dots 01$	$O : 0111010 \dots 00$	$P : 100000 \dots 10$

- Vorteil: geringerer Speicherbedarf
 - ▶ Beispiel: 150.000 Tupel, 3 verschiedene Schlüsselwerte, 4 Byte für TID
 - ★ B^+ -Baum: 600 KB
 - ★ Bitmap: $3 \cdot 18750$ Byte = 56KB
- Nachteil: Aktualisierungsaufwand

Bitmap-Index: Realisierung (2)

- Definition in Oracle

```
CREATE BITMAP INDEX bestellstatus_idx  
ON bestellung(status);
```

- Nutzung speziell zur Star-Query-Transformation (Verbund zwischen Dimensions- und Faktentabelle)
- Speicherung in komprimierter Form
- Weiterhin: Verbund-Indexe auf Bitmap-Basis

Standard-Bitmap-Index

- Jede Dimension wird getrennt abgespeichert
- Für jede Ausprägung eines Attributs wird ein Bitmap-Vektor angelegt:
 - ▶ Für jedes Tupel steht ein Bit, dieses wird auf 1 gesetzt, wenn das indexierte Attribut in dem Tupel den Referenzwert dieses Bitmap-Vektors enthält
 - ▶ Die Anzahl der entstehenden Bitmap-Vektoren pro Dimension entspricht der Anzahl der unterschiedlichen Werte, die für das Attribut vorkommen

Standard-Bitmap-Index (2)

- Beispiel: Attribut Geschlecht
 - ▶ 2 Wertausprägungen (m/w)
 - ▶ 2 Bitmap-Vektoren

PersId	Name	Geschlecht	Bitmap-w	Bitmap-m
007	James Bond	M	0	1
008	Amelie Lux	W	1	0
010	Harald Schmidt	M	0	1
011	Heike Drechsler	W	1	0

Standard-Bitmap-Index (3)

- Selektion von Tupeln kann nun durch entsprechende Verknüpfung von Bitmap-Vektoren erfolgen
- Beispiel: Bitmap-Index über Attribute Geschlecht und Geburtsmonat
 - ▶ 2 Bitmap-Vektoren B-w und B-m für Geschlecht
 - ▶ 12 Bitmap-Vektoren B-1, ..., B-12 für die Monate, wenn alle Monate vorkommen
- Anfrage: „alle im März geborenen Frauen“
 - ▶ Berechnung: $B-w \wedge B-3$ (bitweise konjunktiv verknüpft)
 - ▶ Ergebnis: alle Tupel, an deren Position im Bitmap-Vektor des Ergebnis eine 1 steht

Mehrkomponenten-Bitmap-Index

- Bei Standard-Bitmap-Indexen entstehen für Attribute mit vielen Ausprägungen sehr viele Bitmap-Vektoren
- $\langle n, m \rangle$ -Mehrkomponenten-Bitmap-Indexe erlauben es $n \cdot m$ mögliche Ausprägungen durch $n + m$ Bitmap-Vektoren zu indizieren
- Jeder Wert $x (0 \leq x \leq n \cdot m - 1)$ kann durch zwei Werte y und z repräsentiert werden:

$$x = n \cdot y + z \text{ mit } 0 \leq y \leq m - 1 \text{ und } 0 \leq z \leq n - 1$$

- ▶ Maximal m Bitmap-Vektoren für y und n Bitmap-Vektoren für z
- ▶ Speicheraufwand reduziert sich von $n \cdot m$ auf $n + m$ Vektoren
- ▶ Jedoch müssen für Punktanfragen 2 Bitmap-Vektoren gelesen werden

Mehrkomponenten-Bitmap-Index (2)

- Beispiel: Zweikomponenten-Bitmap-Index
- Für $M = 0..11$ etwa $x = 4 \cdot y + z$
- y-Werte: B-2-1, B-1-1, B-0-1
- z-Werte: B-3-0, B-2-0, B-1-0, B-0-0

x	y			z			
M	B-2-1	B-1-1	B-0-1	B-3-0	B-2-0	B-1-0	B-0-0
5	0	1	0	0	0	1	0
3	0	0	1	1	0	0	0
0	0	0	1	0	0	0	1
11	1	0	0	1	0	0	0

Beispiel: Postleitzahlen

- Kodierung von Postleitzahlen
- Werte 00000 bis 99999
- Direkte Umsetzung: 100.000 Spalten
- Zweikomponenten-Bitmap-Index (erste 2 Ziffern + 3 letzte Ziffern): 1.100 Spalten
- Fünf Komponenten: **50 Spalten**
 - ▶ Geeignet für Bereichsanfragen „PLZ 39***“
- Binärkodiert (bis 2^{17}): 34 Spalten
 - ▶ Nur für Punktanfragen!
- *Bemerkung: Kodierung zur Basis 3 ergibt sogar nur 33 Spalten....*

Bereichskodierter Bitmap-Index

- Standard- und Mehrkomponenten-Bitmap-Indexe
 - ▶ Für Punktanfragen sehr gut geeignet
 - ▶ Für große Bereiche ineffizient, da viele Bitmap-Vektoren verknüpft werden müssen
- Idee der bereichskodierten Bitmap-Indexe:

Im Bitmap-Vektor wird ein Bit auf 1 gesetzt, wenn der Wert des Attributs des zu dieser Position gehörigen Tupels kleiner oder gleich dem gegebenen Wert ist.

- Bereichsanfrage $2 \leq attr \leq 7$ benötigt zur Beantwortung nur 2 Bitmap-Vektoren: B-1 und B-7.
- Ergebnis-Bitmap-Vektor ist $((\neg B-1) \wedge B-7)$.
 - ▶ Für Bereichsanfragen müssen max. 2 Bitmap-Vektoren gelesen werden (nur einer bei nur einseitig begrenzten Bereichen)
 - ▶ Für Punktabfragen müssen genau 2 Bitmap-Vektoren gelesen werden

Bereichskodierter Bitmap-Index

Monat M	Dez B-11	Nov B-10	Okt B-9	Sep B-8	Aug B-7	Jul B-6	Jun B-5	Mai B-4	Apr B-3	Mär B-2	Feb B-1	Jan B-0
Juni - 5	1	1	1	1	1	1	1	0	0	0	0	0
April - 3	1	1	1	1	1	1	1	1	1	0	0	0
Jan. - 0	1	1	1	1	1	1	1	1	1	1	1	1
Feb. - 1	1	1	1	1	1	1	1	1	1	1	1	0
April - 3	1	1	1	1	1	1	1	1	1	0	0	0
Dez. - 11	1	0	0	0	0	0	0	0	0	0	0	0
Aug. - 7	1	1	1	1	1	0	0	0	0	0	0	0
Sept. - 8	1	1	1	1	0	0	0	0	0	0	0	0

- Bereichsanfrage $Februar \leq Datum \leq August$ benötigt B-0 und B-7.
- Ergebnis-Bitmap-Vektor ist $((\neg B-0) \wedge B-7)$

Mehrkomponenten-bereichskodierter Bitmap-Index

- Kombination der beiden Techniken
- Zunächst wird ein Mehrkomponenten-Bitmap-Index angelegt
- Auf jede entstehende Gruppe von Bitmap-Vektoren wird die Bereichskodierung angewendet
 - ▶ Aufgrund der Mehrkomponenten-Technik geringer Speicherbedarf (da geringe Anzahl von Bitmap-Vektoren)
 - ▶ Die Bereichskodierung erlaubt zudem effiziente Unterstützung für Bereichsanfragen
 - ▶ Durch die Bereichskodierung wird in jeder Gruppe von Bitmap-Vektoren (Komponenten) ein Vektor überflüssig (der den Wert $n - 1$ bzw. $m - 1$ repräsentiert, da dort immer alle Bits auf 1 gesetzt sein müssen); also nur $n + m - 2$ Bitmap-Vektoren benötigt

Beispiel MKBKBMI

● Beispiel Mehrkomponenten-bereichskodierter Bitmap-Index

- ▶ $B-0-1' = B-0-1$
- ▶ $B-1-1' = B-1-1 \vee B-0-1'$
- ▶ $B-2-1' = B-2-1 \vee B-1-1' = B-2-1 \vee B-1-1 \vee B-0-1 = 1$

M	B-1-1'	B-0-1'	B-2-0'	B-1-0'	B-0-0'
5	1	0	1	1	0
3	1	1	0	0	0
0	1	1	1	1	1
11	0	0	0	0	0

Intervallkodierte Indexierung

- Prinzip: jeder Bitmap-Vektor repräsentiert definiertes Intervall
- Beispiel: Intervalle $I_{-0} = [0, 5]$, $I_{-1} = [1, 6]$, $I_{-2} = [2, 7]$, $I_{-3} = [3, 8]$, $I_{-4} = [4, 9]$, $I_{-5} = [5, 10]$

M	I-5	I-4	I-3	I-2	I-1	I-0
5	1	1	1	1	1	1
3	0	0	1	1	1	1
0	0	0	0	0	0	1
11	0	0	0	0	0	0
10	1	0	0	0	0	0

- Anfrage: $(2 \leq M \leq 8) \rightsquigarrow$ Auswertung von $I_{-2} \vee I_{-3}$

Auswahl von Indexstrukturen

- In Data Warehouses i.d.R. multidimensionale Bereichsanfragen
- Auswahl der Indexstruktur abhängig vom Anfrageprofil
 - ▶ Wird ein bestimmtes Attribut bevorzugt eingeschränkt?
 - ⇒ Bei asymmetrischen Indexstrukturen Reihenfolge der Indexattribute nach ihrer Vorkommenshäufigkeit im Anfrageprofil wählen
 - ▶ Wenn kein Attribut als besonders wichtig ausgezeichnet werden kann bzw. sehr viele Ad-hoc-Anfragen auftreten, sind symmetrische Strukturen (Sekundärindexe, multidimensionale Indexe) sinnvoll

Auswahl von Indexstrukturen (2)

- Standard-Bitmap-Index
 - ▶ Schnelle, effiziente Implementierung
 - ▶ Großer Speicherplatzbedarf bei großer Anzahl von Ausprägungen
- Mehrkomponenten-Bitmap-Index
 - ▶ Für Punktanfragen kleinste Anzahl von Leseoperationen
- Bereichskodierter Bitmap-Index
 - ▶ Einseitig beschränkte Bereichsanfragen
- Intervallkodierter Bitmap-Index
 - ▶ Zweiseitig beschränkte Bereichsanfragen

Verbundindex

- Beschleunigung von Verbundberechnungen durch Indexierung von Attributen „fremder“ Relationen
- Vorberechnung des Verbundes und Speichern als Indexstruktur
- Verbund/Gruppierung teilweise ohne Zugriff auf fremde Relation (z.B. Dimensionstabelle) möglich

Verbundindex: Beispiel

Verkauf	V_ROWID	GeoID	ZeitID	Verkäufe	...
	0x001	101	11	200	
	0x002	101	11	210	
	0x003	102	11	190	
	0x004	102	11	195	
	0x005	103	11	100	
	0x006	103	11	95	

Geographie	G_ROWID	GeoID	Filiale	Stadt	...
	0x100	101	Allee-Center	Magdeburg	
	0x101	102	Bördepark	Magdeburg	
	0x102	103	Anger	Erfurt	
	0x103	104	Erfurter Str.	Ilmenau	

```
CREATE INDEX joinidx ON Verkauf (Geographie.GeoID)
USING Verkauf.GeoID = Geographie.GeoID
```

0x100: { 0x001, 0x002, ... }
0x101: { 0x003, 0x004, ... }

Bitmap-Verbundindex

- Bisher:
 - ▶ Prädikate für Bitmap-Indexe nicht auf Fremdschlüssel angewendet
 - ▶ Verbund muss weiterhin ausgeführt werden
- Bitmap-Indexe nur für Star-Join-Optimierung hilfreich
- Kombination von
 - ▶ Bitmap-Index und
 - ▶ Verbund-Index

Bitmap-Verbundindex mit Oracle

- Definition

```
CREATE BITMAP INDEX sr_idx  
  ON sales(region.region_name)  
  FROM sales, region  
  WHERE sales.region_id = region.id
```

- Macht Joins überflüssig (kein Zugriff auf `region` notwendig)
- Verknüpfung mit anderen Bitmap-Indexten auf Tabelle `sales` möglich

```
SELECT SUM(sales.amount)  
FROM sales, region  
WHERE sales.region_id = region.id AND  
  region.region_name = 'Thüringen'
```

- Infos: Oracle Dokumentation 11g2 - Part E25789-01

Indexierte Sichten

- SQL Server 2008: Indizierung von Sichten
- Materialisierung der betroffenen Daten
- Automatische Aktualisierung bei Änderung der Basisdaten → materialisierte Sichten

```
CREATE VIEW Verkaeufe2009 AS  
SELECT Stadt, Verkäufe, V.ZeitID, V.GeographieID  
FROM Verkauf V, Zeit Z, Geographie G  
WHERE V.ZeitID = Z.ZeitID AND Z.Jahr = 2009  
      AND V.GeographieID = G.GeographieID;
```

```
CREATE UNIQUE CLUSTERED INDEX V2009_IDX  
      ON Verkaeufe2009(ZeitID, GeographieID);
```

Mehrdimensionale Indexstrukturen

- Hash-basierte Strukturen
 - ▶ Grid-Files
 - ▶ Mehrdimensionales dynamisches Hashen
- Baum-basierte Strukturen
 - ▶ kdB-Baum
 - ▶ R-Baum [Gutman 1984]
 - ▶ UB-Baum [Bayer 1996]

Grid-File

- Mehrdimensionale Dateiorganisationsform
 - ▶ Kombination von Elementen der Schlüsseltransformation (Hash-Verfahren) und Indexdateien (Baumverfahren)
- Idee
 - ▶ Dimensionsverfeinerung: gleichmäßige Aufteilung des mehrdimensionalen Raumes in der ausgewählten Dimension durch vollständigen Schnitt (Einziehen einer Hyperebene)

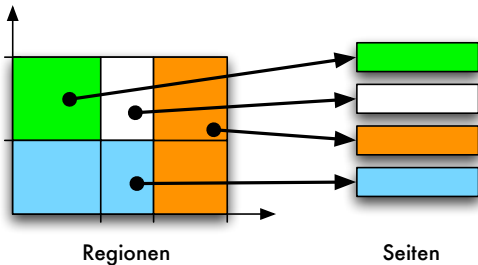
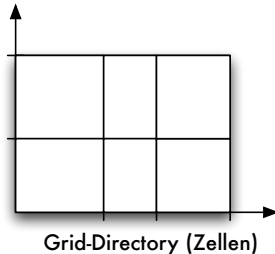
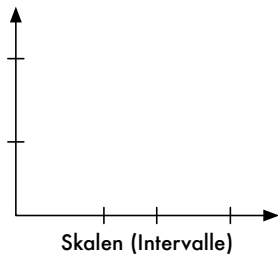
Grid-File: Prinzipien

- Zerlegung des Datenraums in Quader (Suchregion: k -dimensionale Quader)
- Nachbarschaftserhaltung: Speicherung ähnlicher Objekte auf gleicher Seite
- Symmetrische Behandlung aller Raum-Dimensionen: partial-match-Anfragen
- Dynamische Anpassung der Struktur beim Einfügen und Löschen
- **Prinzip der 2 Plattenzugriffe** bei exact-match-Anfragen

Grid-File: Aufbau

- Grid: k eindimensionale Felder (Skalen), jede Skala repräsentiert Attribut
- Skalen: bestehen aus Partition der zugeordneten Wertebereiche in Intervalle
- Grid-Directory: besteht aus Grid-Zellen, die Datenraum in Quader zerlegen
- Grid-Zellen: bilden eine Grid-Region, der genau eine Datensatz-Seite zugeordnet ist
- Grid-Region: k -dimensionales, konvexes Gebilde (paarweise disjunkt)

Grid-File: Aufbau (2)



Grid-File: Operationen

- Ausgangszustand: Zelle = Region = eine Datensatz-Seite
- Seitenüberlauf
 - ▶ Teilen von Seiten
 - ▶ Falls zur Seite gehörende Grid-Region nur eine Zelle: Unterteilung des Intervalls auf einer Skala
 - ▶ Falls Region aus mehreren Zellen: Zerlegung dieser Zellen in einzelne Regionen
- Seitenunterlauf
 - ▶ Zusammenfassen von zwei Regionen falls Ergebnis eine konvexe Region

Mehrdimensionales Hashen – MDH

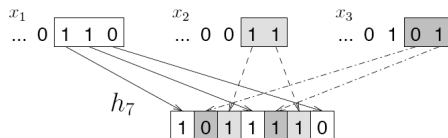
- Idee: Bit Verschränkung (Interleaving)
- Abwechselnd von verschiedenen Zugriffsattributwerten die Bits der Adresse berechnen
- Beispiel: zwei Dimensionen

	*0*0	*0*1	*1*0	*1*1
0*0*	0000	0001	0100	0101
0*1*	0010	0011	0110	0111
1*0*	1000	1001	1100	1101
1*1*	1010	1011	1110	1111

Idee MDH

- MDH baut auf *linearem Hashen* auf
 - ▶ Dynamisches Hash-Verfahren
 - ▶ Bitsequenz-Prefixe adressieren Speicherblöcke
- Hash-Werte sind Bit-Folgen, von denen jeweils ein Anfangsstück als aktueller Hash-Wert dient
 - ▶ Bei binären Zahlen: oft Invertierung der Bit-Darstellung vor Prefix-Bildung
- Je ein Bit-String pro beteiligtem Attribut berechnen
- Anfangsstücke nach dem Prinzip des Bit-Interleaving zyklisch abarbeiten
- Hash-Wert reihum aus den Bits der Einzelwerte zusammensetzen
- Familie von Hash-Funktionen h_i für Bitfolgen der Länge i
 - ▶ Dynamisches Wachstum: gehe von i auf $i + 1$

MDH Veranschaulichung



- Verdeutlicht Komposition der Hash-Funktion h_i für drei Dimensionen und den Wert $i = 7$
- Bei $i = 8$ wird ein weiteres Bit von x_2 (genauer: von $h_{8_2}(x_2)$) verwendet
- MDH Komplexität
 - ▶ Exact-Match-Anfragen: $O(1)$
 - ▶ Partial-Match-Anfragen, t von k Attributen festgelegt, Aufwand $O(n^{1-\frac{t}{k}})$
 - ▶ Ergibt sich aus Zahl der Seiten, wenn bestimmte Bits „unknown“
 - ▶ Spezialfälle: $O(1)$ für $t = k$ und $O(n)$ für $t = 0$

kdB-Baum

- k-dimensionale Indexbäume

- ▶ kd-Baum: binärer Baum für mehrdimensionale Grundstruktur; Hauptspeicheralgorithmen [Bentley 1975]
- ▶ kdB-Baum: Kombination aus kd-Baum und B-Baum (höherer Verzweigungsgrad)
- ▶ kdB-Baum: Verbesserung des kd-Baums

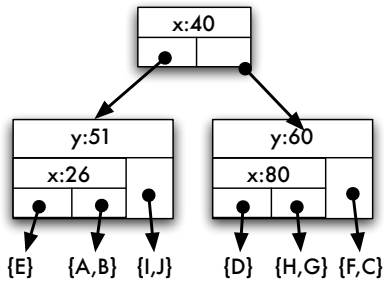
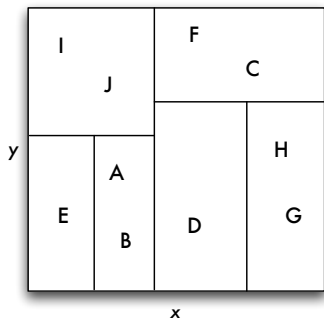
- Idee des kdB-Baums

- ▶ Auf jeder Indexseite einen Teilbaum darstellen, der nach mehreren Attributen hintereinander verzweigt
- ▶ Aufwand: exact-match $O(\log n)$, partial match besser als $O(n)$

kdB-Baum: Struktur

- kdB-Baum vom Typ (b, t)
- Bereichsseiten (innere Knoten): enthalten kd-Baum mit max. b internen Knoten
 - ▶ kd-Baum mit Schnittelementen und zwei Zeigern
 - ▶ Schnittelement: Zugriffsattribut und -wert
 - ▶ Linker Zeiger: kleinere Zugriffsattributwerte
 - ▶ Rechter Zeiger: größere Zugriffsattributwerte
- Satzseiten (Blätter): enthalten bis zu t Tupel der gespeicherten Relation

kdB-Baum: Beispiel

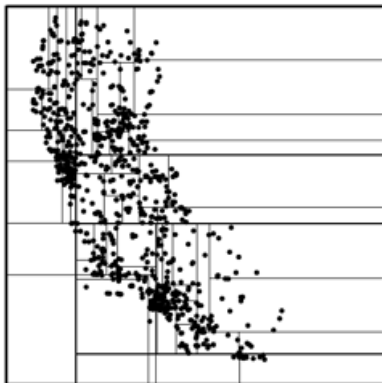


kdB-Baum: Trennattribute

- Reihenfolge
 - ▶ Zyklisch festgelegt
 - ▶ Einbeziehung der Selektivität: Zugriffsattribut mit hoher Selektivität möglichst früh und häufiger als Schnittelement einsetzen
- Trennattributwert
 - ▶ Ermittlung einer geeigneten Mitte des Wertebereichs aufgrund von Verteilungsinformationen

k[dD](B)-Baum: Fazit

- Speichert auch schlecht verteilte Daten
- Schwer handhabbar für mehr als 3 Dimensionen



R-Baum

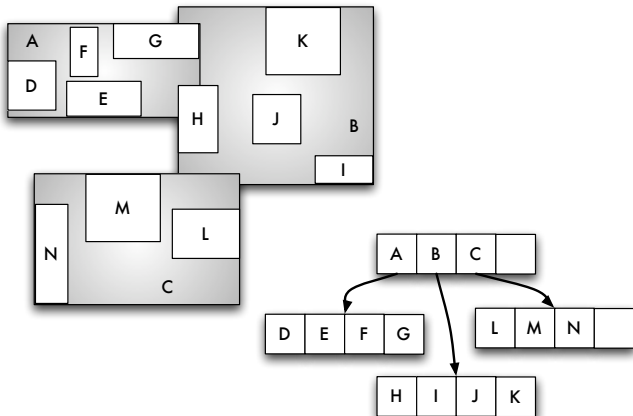
- Jeder Knoten des R-Baums speichert max. m Indexeinträge
($m = 2d + 1$)
- Jeder Knoten (außer der Wurzel) enthält mindestens n Einträge
($n = d + 1$)
- d -dimensionaler R-Baum verwendet d -dimensionale Intervalle (Rechtecke, Quader) zur Indexierung des Datenraums
- Eintrag auf Blattebene: (I, tid) mit I ein d -dimensionales Intervall und tid Tupelidentifikator, der den entsprechenden Datensatz referenziert
- Eintrag in inneren Knoten: (I, cp) mit I dem d -dimensionalen Intervall, das alle Intervalle der Einträge des Kindknotens umfasst (minimum bounding box) und cp Zeiger auf diesen Kindknoten (child pointer)

R-Baum (2)

Besonderheiten (im Vergleich zum B^* -Baum)

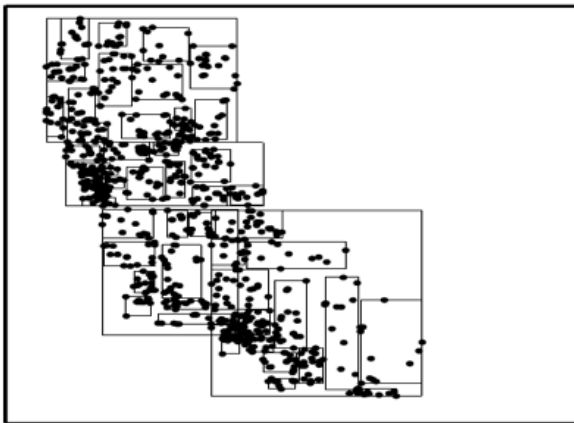
- Suchen: falls verschiedene Regionen von Knoten gleicher Ebene sich überlappen, müssen selbst bei Punktanfragen evtl. mehrere Nachfolger traversiert werden
- Einfügen: es wird versucht, ein Intervall zu finden, das nicht erweitert werden muss; ansonsten das, das am wenigsten erweitert werden muss
- Löschen von Daten spielt für Data Warehouses i.d.R. keine Rolle; Einfügen nur in größeren Abständen (dann aber oft mit vielen neuen Tupeln)
 - ▶ Effiziente Möglichkeit des Bottom-up-Aufbaus der R-Baum-Struktur wichtig

R-Baum: Beispiel



R-Baum: Fazit

- Bessere Anpassung der Bereiche an Daten

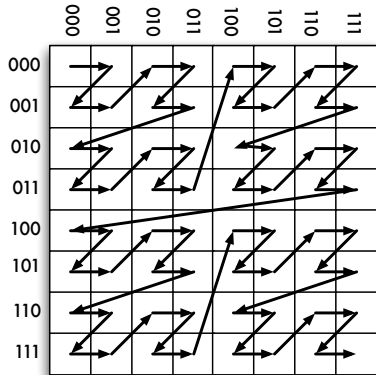


UB-Baum

- Datenraum wird mittels einer raumfüllenden Kurve in disjunkte Teilräume aufgeteilt (meist die sogenannte Z-Kurve)
- Jeder Punkt des von den zu indizierenden Attributen aufgespannten mehrdimensionalen Raums wird auf einen skalaren Wert, den Z-Wert, abgebildet
- Z-Werte werden als Schlüssel in einem herkömmlichen B^+ -Baum verwendet

UB-Baum (2)

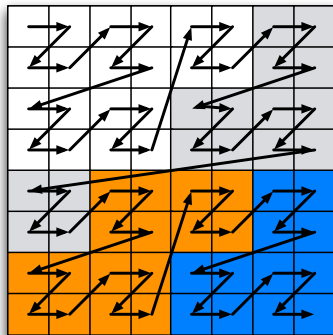
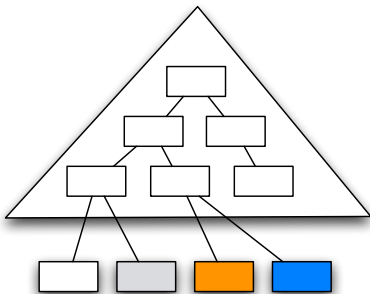
- Aufteilung eines 2-dimensionalen Raums mit der Z-Kurve:



UB-Baum (3)

- **Z-Werte** lassen sich effizient (in linearer Zeit) berechnen:
 - ▶ Je Dimension werden die Basisintervalle binär durchnummeriert;
 - ▶ Durch Verschränken der Bits (bit interleaving) ergibt sich dann der jeweilige Z-Wert.
- **Z-Region**: wird durch ein Intervall $[a, b]$ von Z-Werten bestimmt
 - ▶ Z-Regionen eines UB-Baums werden dynamisch so angepasst, dass die innerhalb einer Z-Region liegenden Objekte genau auf eine B^+ -Baum-Seite passen
 - ▶ Damit kann ein B^+ -Baum als Basisstruktur verwendet werden

UB-Baum (4)



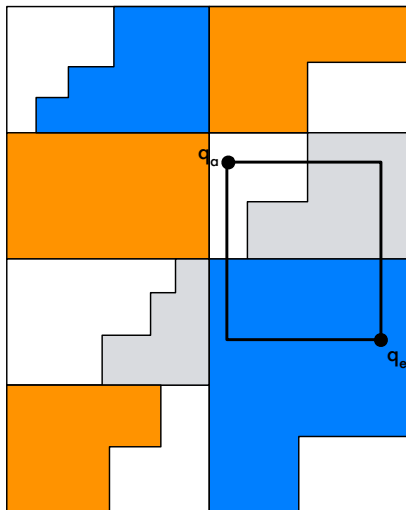
UB-Baum: Bereichssuche (RQ-Algorithmus)

- Jede Bereichsanfrage wird durch 2 Datensätze q_a und q_e bestimmt, die (bildlich) die linke obere und die rechte untere Ecke des Anfragebereichs spezifizieren
 - 1 Beginne mit q_a und berechne die zugehörige Z-Region
 - 2 Lade die entsprechende Seite und wende Anfrageprädikat auf alle Datensätze darin an
 - 3 Berechne den nächsten Bereich der Z-Kurze, der innerhalb des Anfragebereichs liegt
 - 4 Wiederhole 2. und 3. bis die Endadresse der bearbeiteten Z-Region größer ist als q_e
(also den Endpunkt des Anfragebereichs enthält)

UB-Baum: Bereichssuche

- Schritt 3. (Berechnung der Schnittpunkte der Z-Kurve mit dem Anfragebereich)
 - ▶ Erscheint auf den ersten Blick kritisch;
 - ▶ Tatsächlich: effizient durch „ein paar“ Bitoperationen (und ohne Plattenzugriffe) in linearer Zeit lösen (linear in der Länge der Z-Werte)

UB-Baum: Bereichssuche graphisch



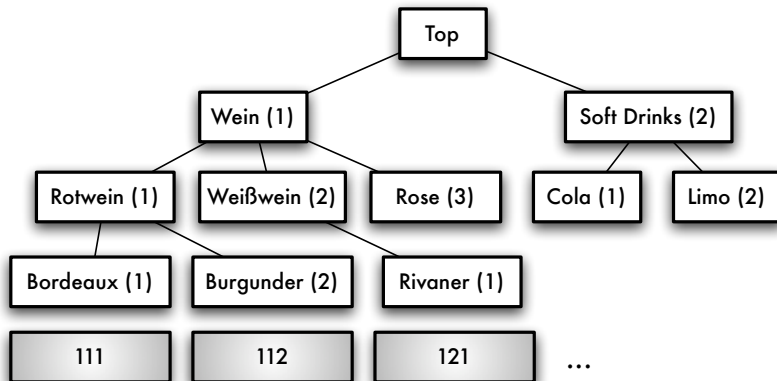
UB-Baum und MHC

- Erweiterungen des UB-Baums für Data Warehouses:
 - ▶ **Mehrdimensionales Hierarchisches Clustering (MHC)**
 - ▶ Unterstützt hierarchisch organisierte Dimensionen, so daß alle Vorteile des UB-Baums erhalten bleiben

MHC: Prinzip

- Totale Ordnung für Hierarchie
- Zuweisung einer eindeutigen Zahl zu jedem Blattelement der Hierarchie
- Elemente des gleichen Unterbaums erhalten zusammenliegende Zahlen (Clustering)
- Berechnung:
 - ▶ Jedes Element einer Hierarchiestufe erhält Zahl (Surrogat)
 - ▶ Für Blattelement: Aneinanderhängen der Surrogate (Binärdarstellung) → Mehrkomponentensurrogat

MHC Beispiel



MHC Nutzung

- Mehrkomponentensurrogate als
 - ▶ Schlüssel für Tupel der Faktentabelle
 - ▶ Indexattribut für UB-Baum
- Beispiel: Bereichsanfrage
 - ▶ Minimales und maximales Mehrkomponentensurrogat als Intervall zur Einschränkung

Zusammenfassung

- Indexstrukturen erlauben verbesserte mehrdimensionale Anfragen
- Eindimensionale Indexstrukturen nicht ausreichend
- B-Baum, Hashverfahren und Erweiterungen eindimensional
- Baum- und Hashverfahren für Mehrdimensionalität anpassen
 - ▶ kdB-Baum, UB-Baum, R-Baum
 - ▶ Grid-Files, MDH