Data Warehousing

Indexierung

Ulf Leser

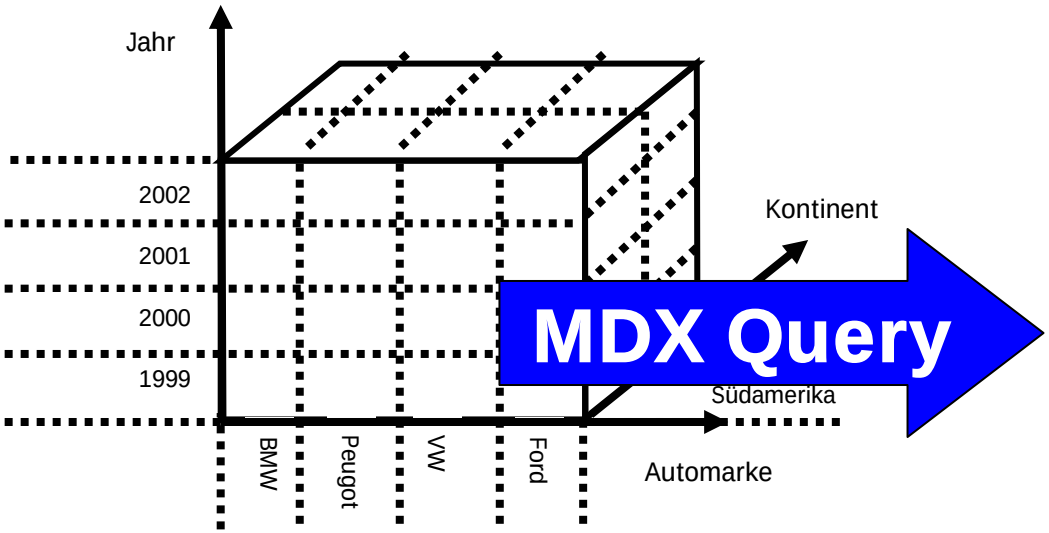
Wissensmanagement in der Bioinformatik

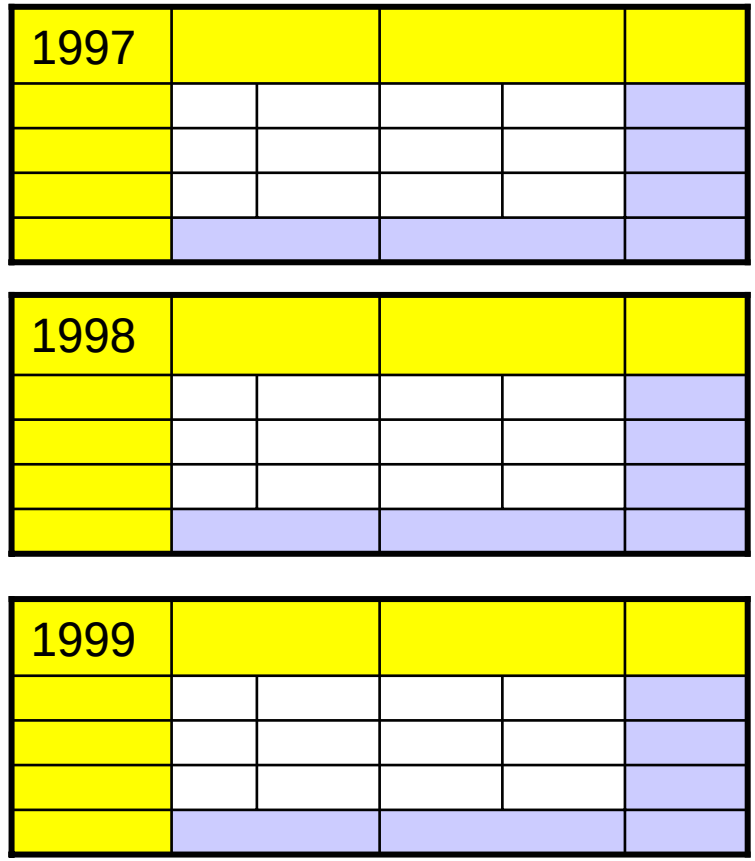
- Nächsten Montag
  - OLAP und Data Mining Plattform Analysis Services und die Data Warehouse Technologie
  - SQL Server 2005
  - Erfahrungen aus Kundenprojekten
  - Vorteile von ROLAP (Echtzeit-Support und flexibleres Datenbankschema) mit den Vorteilen vonMOLAP (Performance und optimierte Storage) verbinden
  - Mit spezielle Dimensionstypen klassische BI-Aufgaben lösen
- Morgen
  - Vortrag von Microstrategy entfällt

MDX

- Microsoft‘s Vorschlag
- Teil von „OLE DB for OLAP“
- MDX: Multidimensional Expressions
- Neue Sprache
  - Standard ohne feste Semantik (by example)
  - MDDM Konzepte als First-Class Elemente
  - SQL-artige Syntax, aber gänzlich andere Bedeutung
  - DDL und DML (hier nur letzteres)
  - Sehr mächtig und komplex

Grundprinzip





Knotenbezeichnung: Unique Names oder Navigation (s.u.)

Verschiedene Klassifikationsstufen möglich

```
SELECT {Wein, Bier, Limo, Saft} ON COLUMNS
       {[2000], [1999], [1998].[1], [1998].[2]} on ROWS
FROM Sales
WHERE  (Measures.Menge, Country.Germany)
```

Slicer: Auswahl des Fakt „Menge“, Beschränkung auf Werte in BRD

	Wein	Bier	Limo	Saft
2000				
1999				
1998.1				
1998.2				

Implizite Summierung von Menge, nur BRD, nach Produktgruppe / Zeitangabe

MDX - Fazit

- Hohe Komplexität
- Mächtige Sprache
- Direkte Anlehnung an MDDM
- Könnte sich als Standard durchsetzen
  - Schnittstelle zwischen OLAP GUI und DB-Server
  - Unterstützt von MS, Microstrategy, Cognos, BusinessObjects, ...

SQL und OLAP

- Übersetzung eines MDDM in ein Star-Schema
- Triviale Operationen
  - Auswahl (slice, dice): Joins und Selects
  - Verfeinerung (drill-down): Joins und Selects
- Einfache Operationen
  - Aggregation um eine Stufe: Group-by
- Schwieriger
  - Hierarchische, multidimensionale Aggregation

ROLLUP Beispiel

```
SELECT T.year_id, T.month_id, T.day_id, sum(...)
FROM   Sales S, Time T
WHERE  T.day_id = S.Day_id
GROUP BY ROLLUP(T.year_id, T.month_id, T.day_id)
```

1997	Jan	1	200
1997	Jan	...	
1997	Jan	31	300
1997	Jan	ALL	31.000
1997	Feb	...	
1997	Feb	ALL	450
1997	...	...	
1997	ALL	ALL	1.456.400
1998	Jan	1	100
1998	...	...	
1998	ALL	ALL	45.000
...	...	...	
ALL	ALL	ALL	12.445.750

Humboldt Universität Informatik

ROLLUP Operator

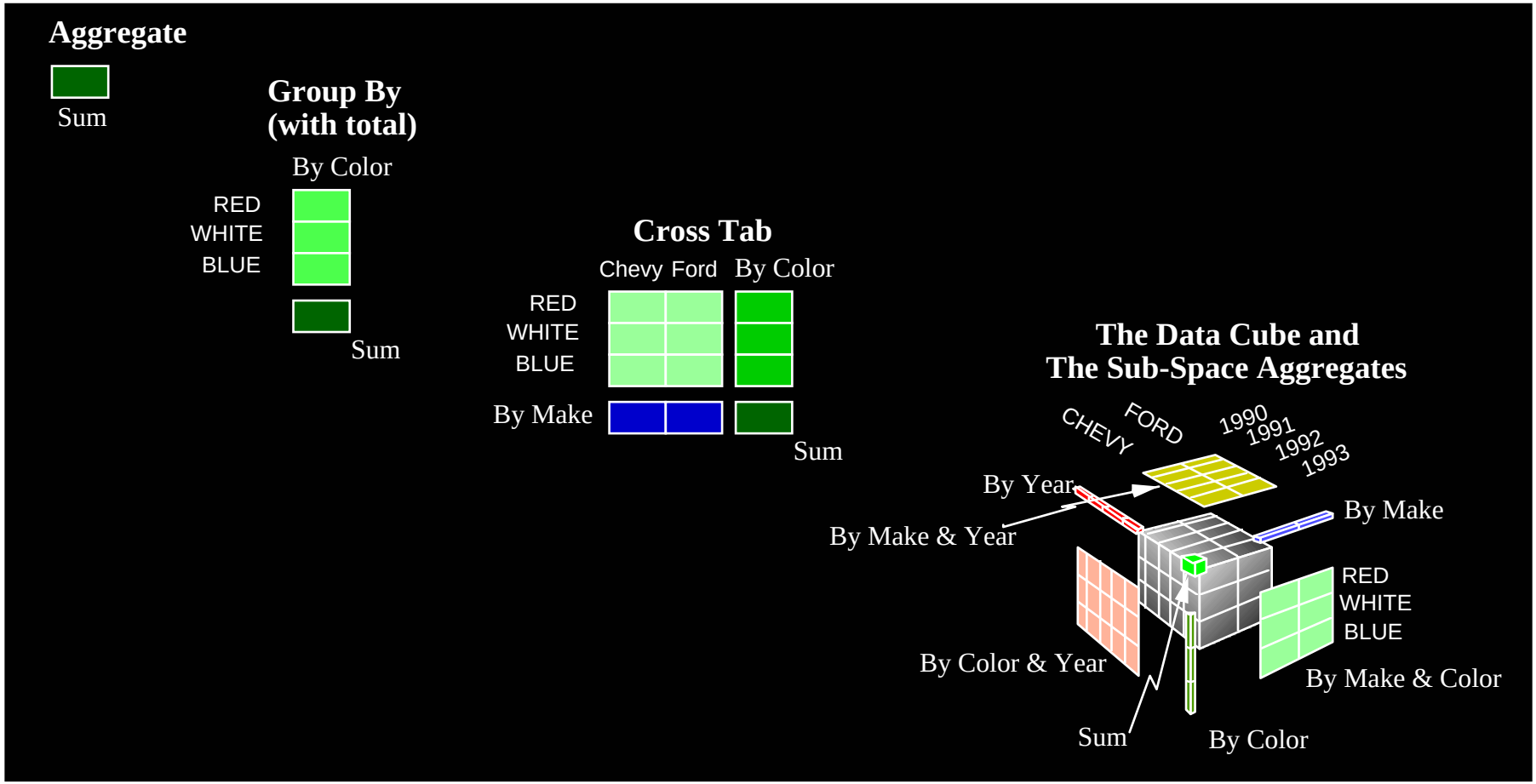
- Herkömmliches SQL
  - Dimension mit k Stufen – Union von k Queries
  - ➢ k Scans der Faktentabelle
    - Keine Optimierung wg. fehlender Multiple-Query Optimierung in kommerziellen RDBMS
  - ➢ Schlechte Ergebnisreihenfolge

- ROLLUP Operator
  - Hierarchische Aggregation mit Zwischensummen
  - Summen werden durch „ALL“ als Wert repräsentiert

Cube Operator

- d Dimensionen, jeweils eine Klassifikationsstufe
  - Jede Dimension kann in Gruppierung enthalten sein oder nicht
  - 2^d Gruppierungsmöglichkeiten
- Herkömmliches SQL
  - Viel Schreibarbeit
  - 2^d Scans der Faktentabelle (wieder keine Optimierung möglich)
- CUBE Operator
  - Berechnung der Summen von sämtlichen Kombinationen der Argumente (Klassifikationsstufen)
  - Summen werden durch „ALL“ repräsentiert
  - Keine Beachtung von Hierarchien
    - Durch Schachtelung mit ROLLUP erreichbar

Cube-Operator: Veranschaulichung



Source: Gray et al., „Datacube“, Microsoft & IBM

Fazit

- Operationen schachtelbar
  - GROUP BY ROLLUP(year, month, day, CUBE(pg_id, shop_id))
- SQL-Standard
  - Implementiert (mindestens) von Oracle, DB2, SQLServer
- Wesentliche Verbesserung
  - Kompaktere Queries
  - Deutliche Beschleunigung durch weniger Scans
  - Aggregierbarkeit beachten: „ROLLUP AVG“?
  - Implementierung von CUBE nicht trivial – sehr viele Zwischensummen
- Weitere Operationen
  - GROUPING – Erkennen von Summenwerten
  - Sliding Windows, sequenzbasierte Operationen
  - Percentile, Rank, TopN – Queries
  - ...
- Aktives Gebiet aller DB-Hersteller

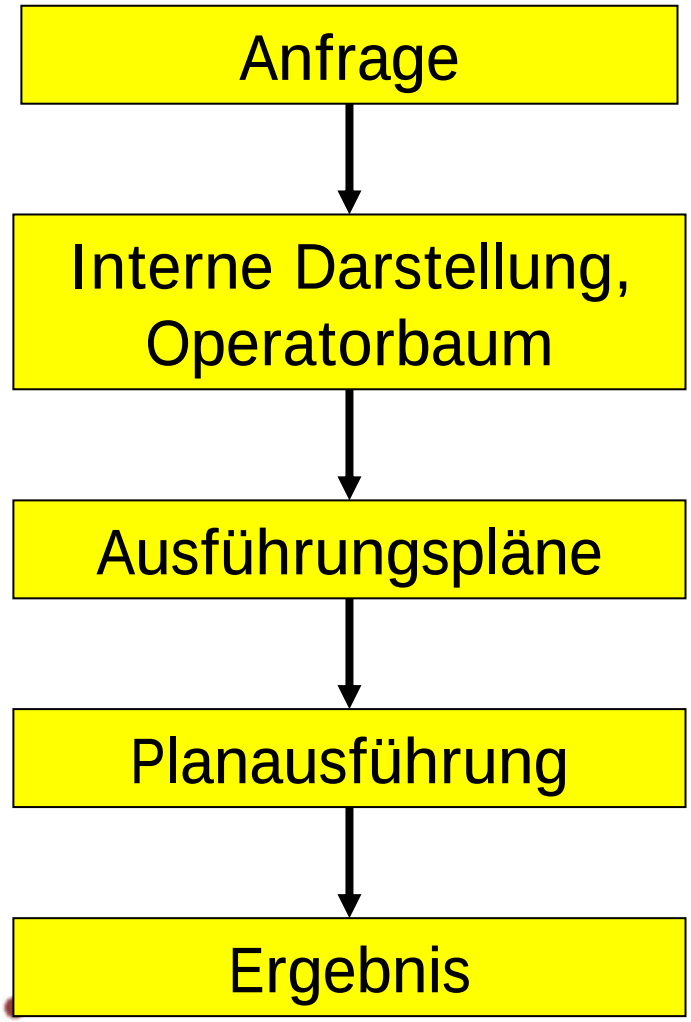
Inhalt dieser Vorlesung

- B-Baum Indexierung
- Bitmap Indexierung
- Multidimensionale Indexierung

Datenbankperformance

- Datenbanken heute fast wichtiger als Betriebssysteme
  - Migration Oracle von SUN nach Windows
  - Migration SAP von Oracle nach DB2
- Performance unternehmenskritisch
- Riesige Unterschiede
  - Schlechte Performance: Lange Wartezeiten, hohe Serverbelastung, lange Sperrzeiten
  - Sehr schlechte Performance: Frage nicht mehr beantwortbar
- Keine Datenbank ist (bisher) selbstoptimierend
- Vielfältige Möglichkeiten zur manuellen Optimierung
  - Speichersubsystem
  - Schema / Vorausberechnung
  - Indexierung

Von der Anfrage zur Antwort



Parsing; Syntaktischer und semantischer Check

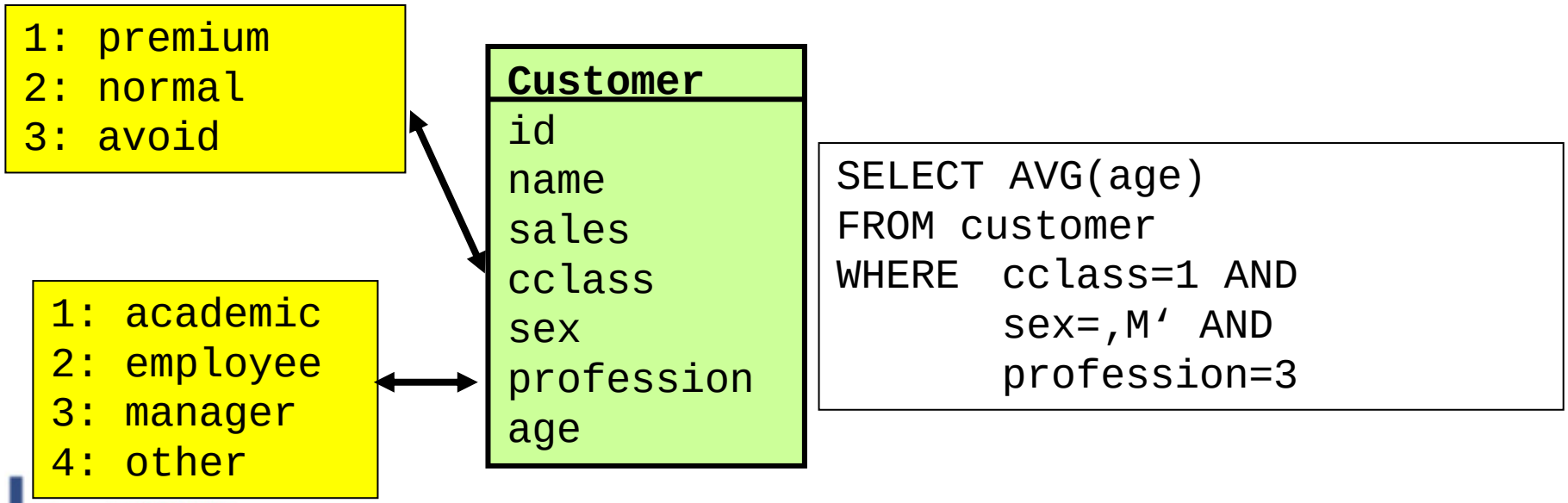
Plangenerierung; Viewauflösung; algebraische Transformationen; Join- und Operatorreihenfolge; Zugriffswege

Dynamische Programmierung; Auswahl des (vermutlich) besten Planes

Sortierung auf Sekundärspeicher, Operatorpipelining, Caching, ...

Zugriff auf eine Tabelle

- Faktentabelle sehr, sehr groß – kritischer Engpass
- Typischer Zugriff
  - Bedingungen/Gruppierung auf Dimensionsattributen
  - Aggregation von Fakten, hierarchisch, mehrdimensional



Motivation

- Full table scan
  - 1.000.000 Einträge
  - Daten von 4 Attributen benötigt
  - Alle Datenblöcke lesen
- Indexierung
  - Annahme: Gleichverteilung der Wert
  - Index auf `sex`: 50% Selektivität
  - Index auf `cclass`: 33% Selektivität
  - Index auf `profession`: 25% Selektivität
- Zusammen: ~4.1% Selektivität
  - Unter Annahme der Unabhängigkeit der Attribute
- Indexzugriff lohnt sich ab 7%

➢ Kein Einzelindex ergibt ausreichende Selektivität

➢ Optimierer wählt Full Table Scan

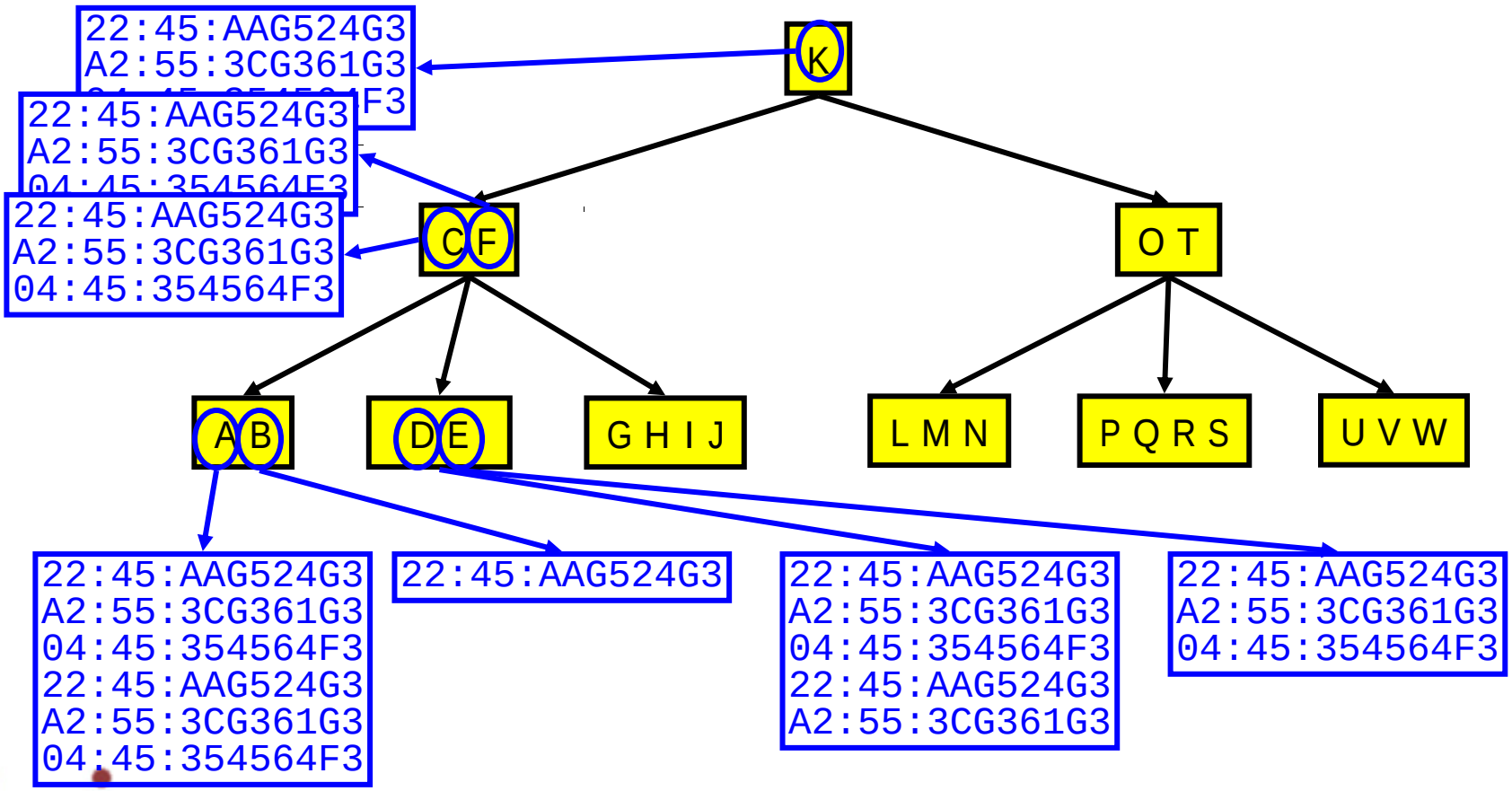
Prinzip eines Index

- Daten liegen ungeordnet in Datenblöcken
- Index
  - Spezielle Datenstruktur
  - Speichert sortierte Werteliste eines Attributs zusammen mit Array von Zeigern auf die physikalischen Adressen der Tupel (TID) in den Datenblöcken
  - Reduktion der Zugriffszeit auf alle Tupel, bei denen das Attribut einen bestimmten Wert hat
    - Punktanfragen auf Primärschlüssel: O(log(n))
- Management
  - Aktualisierung mit Aktualisierung der Tabelle
  - Index muss vom Benutzer explizit angelegt werden (Wizards)

Teil I. B-Bäume

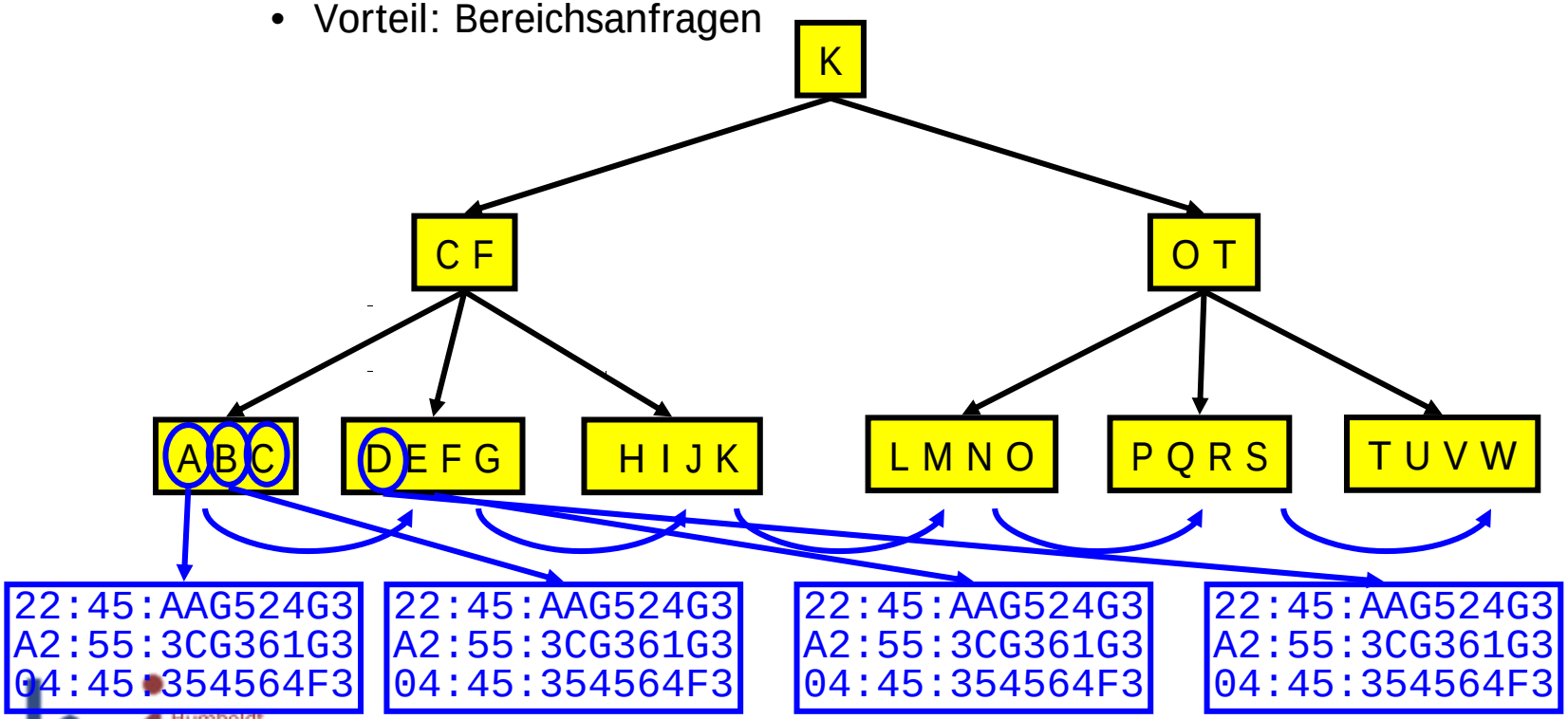
- B-Baum der Ordnung m: Jedes Blatt enthält höchstens 2m und mindestens m Werte (außer Wurzel)
- Die Werte in einem Knoten sind geordnet
- Ein Knoten mit i Werten hat i+1 Kinder
- Zu jedem Wert wird eine TID-Liste gespeichert
- Zugriff auf TID-Liste eines Wertes: $\log_m(n)$ (bei n möglichen Attributwerten)
- B-Baum ist ausgeglichen: Alle Blätter haben dieselbe Höhe
- Einfügen/Löschen/Update: $\log_m(n)$ Algorithmen bekannt
  - Im Worst Case den Baum einmal rauf und einmal runter

Beispiel


22:45:AAG524G3
A2:55:3CG361G3
22:45:AAG524G3
A2:55:3CG361G3
22:45:AAG524G3
A2:55:3CG361G3
04:45:354564F3
K
C F
O T
A B
D E
G H I J
L M N
P Q R S
U V W
22:45:AAG524G3
A2:55:3CG361G3
04:45:354564F3
22:45:AAG524G3
22:45:AAG524G3
A2:55:3CG361G3
04:45:354564F3
22:45:AAG524G3
A2:55:3CG361G3
04:45:354564F3

B*-Bäume

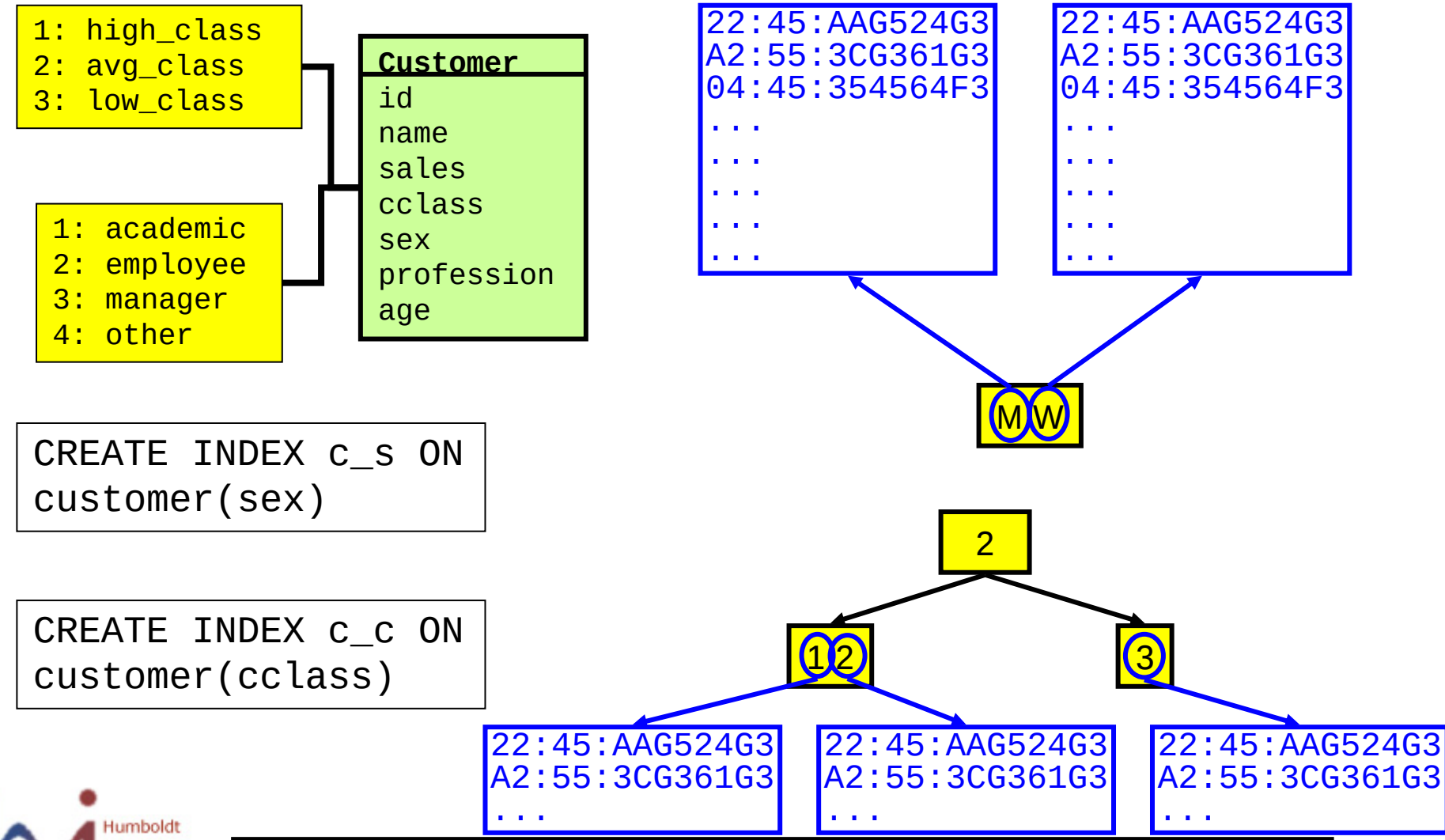
- Wie B-Bäume, aber
  - TID Listen nur in Blättern
    - Viel einfachere Verwaltung, einfachere Operationen, höherer Verzweigungsgrad im Baum
  - Verkettung aller Blätter untereinander
    - Vorteil: Bereichsanfragen



Eigenschaften von B*-Bäumen

- Robuste, generische Datenstruktur
  - Unabhängig vom Datentyp
    - Attributwerte müssen nur vollständig geordnet sein
  - Effiziente Aktualisierungsalgorithmen
  - Kompakt
- Arbeitspferd aller RDBMS
- Aber …
  - Attribute mit geringer Kardinalität
    ⇒ Degenerierte Bäume
  - Zusammengesetzte Indexe
    ⇒ Ordnungssensitiv

Degenerierte B*-Bäume



Falsch geordnete B*-Bäume

- Zusammengesetzter Index
  - Indexierung der Konkatenation mehrerer Attribute
  - Selektivität des zusammengesetzten Index = Produkt der Selektivitäten der Einzelindizes
    - Bei Unabhängigkeit der Attributwerte
- Problem: Ordnungsabhängig

```
CREATE INDEX c_scp ON
customer(sex, cclass,
profession)
```

```
SELECT ...
FROM ...
WHERE   sex=‚m‘ AND
        cclass=1 AND
        prof=‚other‘
```

```
SELECT ...
FROM ...
WHERE   cclass=1 AND
        prof=‚other‘ AND
        sex=‚m‘
```

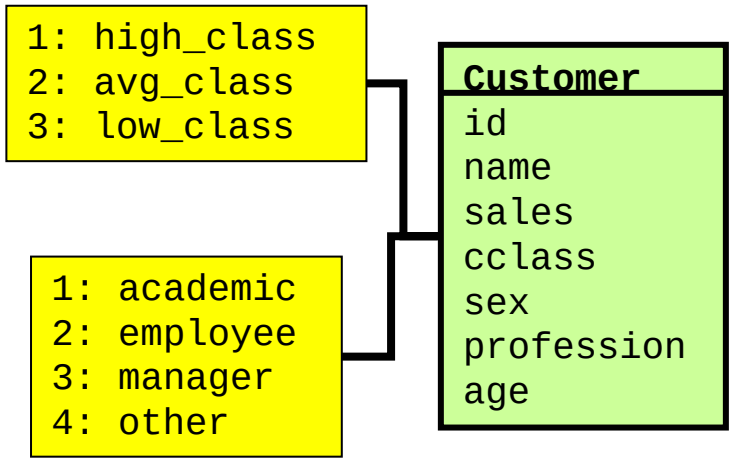
```
SELECT ...
FROM ...
WHERE   cclass=1 AND
        prof=‚other‘
```

Abhilfe

- Attribute mit geringer Kardinalität
  $\Rightarrow$ Bitmap Indexe
- Queries auf hochdimensionalen Daten
  $\Rightarrow$ Multidimensionale Indexe

- Zuerst aber
  – „Tricks“ mit B*-Indexen

Oversized Indexe

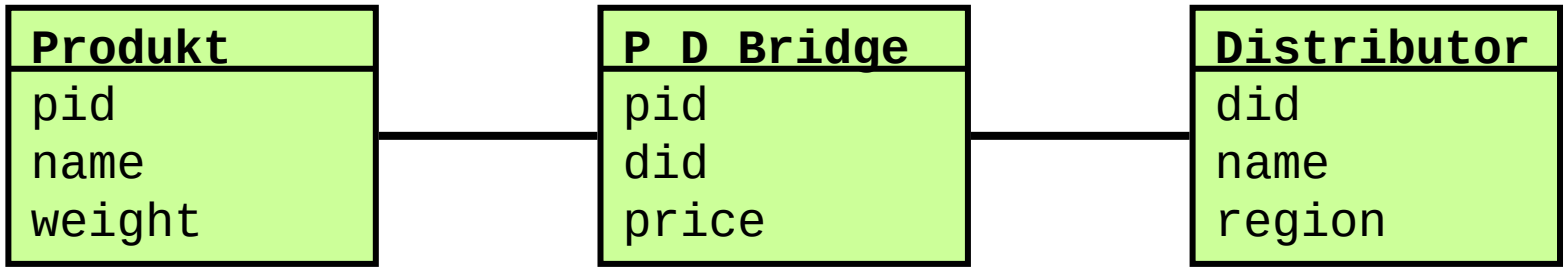



```
SELECT AVG(age)
FROM customer
WHERE  cclass=1 AND
       sex=‚M‘ AND
       profession=3
```

- Normaler Ablauf bei zusammengesetztem Index
  - Suche Werte „1||M||3“ in Baum
  - Ablauf TID Liste, Datenblockzugriff für age Werte
- Besser

```
CREATE INDEX c_scp ON
customer(sex, cclass, profession, age)
```

Index-Organized Tables

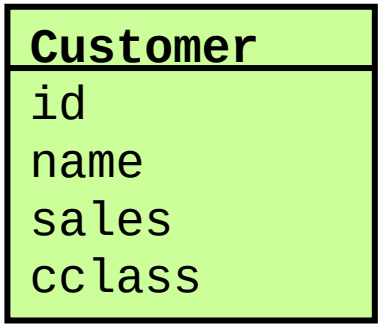



```
SELECT MIN(pd.price)
FROM produkt p, .. pd, d
WHERE  pd.pid=p.pid AND
       pd.did=d.id AND
GROUP BY pd.pid
```

```
CREATE INDEX bridge
ON p_d_bridge( pid, did, price)
```

- Index ist eine (geordnete) Kopie der Tabelle
- ➢ Anlegen der Tabelle als „Index-Organized“
- ➢ Halbierung des Speicherbedarfs

Berechnete Indexe




```
CREATE INDEX c_name ON
customer(name)
```

```
SELECT id, ...
FROM customer
WHERE  name=‚Meier‘ OR
       name=‚meyer‘ OR
       name=‚meier‘ OR
       ...
```

- Keine Benutzung von c_name möglich
- Besser

```
CREATE INDEX c_name ON
customer(upper(name))
```

```
CREATE INDEX c_name ON
customer(soundex(name))
```

Mehr zu Indexen

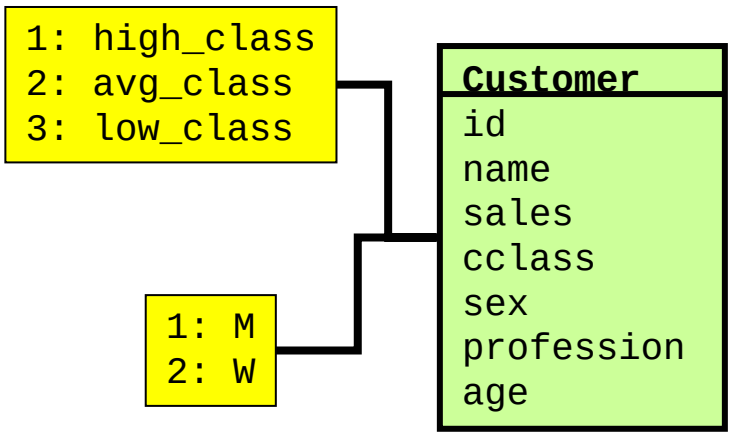
- Oracle 9i: Index Skip Scan
  - Benutzt zusammengesetzte Indexe auch, wenn erstes Attribut nicht in Bedingung enthalten
  - Prinzip: Durchsuchung des sekundären Index für alle DISTINCT Werte des ersten Attributs
  - Verdacht: Oracle schreibt Query in UNION um mit allen möglichen Werten für erstes Attribut
  - Lohnt nur, wenn erstes Attribut sehr niedrige Kardinalität hat
- User-Defined Index
  - Implementierung eigener Indexstrukturen
  - Angabe eigener Kostenabschätzungen
  - Verbindung mit User-Defined Types
  - Transparente Benutzung

Teil 2. Bitmapindex

- Adressierte Probleme
  - Indexierung von Attributen niedriger Kardinalität
    - B*-Bäume entarten
  - Auswertung von Bedingungen auf mehreren Attributen
    - Zusammengesetzte B*-Indexe sind ordnungssensitiv
- Kernidee
  - Repräsentation von Attributen als Bitmatrizen
    - Niedrige Kardinalität – kleine Matrizen
  - Kombinierte Bedingungen durch Bitoperationen implementierbar
    - Sehr schnell ausführbar

Bitmapindex Grundaufbau

- Tabelle T mit Attribut A; |T|=n Tupel, |A|=a verschiedene Werte
- Repräsentation eines Attributwerts: Bitarray der Länge n
  - Attribut X, Wert z: X.z[i]=1 gdw i.tes Tupel hat Wert z in Attribut X
- Repräsentation eines Attributes: Bitmatrix mit n*a Bits



Tabelle

```
22:45,1,Meier,20.000,2,M,...
A2:55,2,Müller,15.000,3,W,...
33:D1,3,Schmidt,25.000,1,M,...
1A:0E,4,Dehnert,22.000,2,M,...
...
```

Bitmapindex cclass

```
1:0010...
2:1001...
3:0100...
```

Bitmapindex sex

```
M:1011...
W:0100...
```

Vorteil 1 - Speicherplatz

- Vergleich Bitmap mit B*-Baum
- Kompakte Repräsentation bei kleinem a
  - Annahmen
    - n=1.000.000, zwei Attribute mit a_1=4, a_2=2, TID=4 Byte
  - 2 B*-Bäume: 2*4*1.000.000 = 8MB
    - In jedem Index ist jede TID einmal repräsentiert
    - Speicherplatz unabhängig von Kardinalität der Attribute
  - Bitmapindex: 1.000.000 * (4+2)/8 = 0,75MB
- Kleiner Index passt eher in Hauptspeicher
- Bedingung
  - Ordnung und Zahl der TIDs muss konstant bleiben
  - Bitmap Indexe nicht geeignet für Tabellen mit vielen Änderungen

Vorteil 2 – Komplexe Bedingungen

- Zugriff mit Gleichheitsbedingungen an mehrere Attribute
  - B*-Index
    - Lesen der TID Liste zu jeder Bedingung
    - Schnittmengenbildung aller Listen durch Sortierung
      - Sehr teuer bei geringer Selektivität
    - Auf Tupel in Schnittmenge zugreifen
  - Bitmap Index
    - Lesen der Bitarrays für jede Bedingung
    - OR/AND Verknüpfung
      „... cclass=3 AND sex=‚m‘ ...“ $\Rightarrow$ B[2] $\wedge$ B[4]
    - Auf Tupel mit passenden Bits zugreifen
- Kein wesentlicher Gewinn bei einfachen Bedingungen
  - Auch durch Bitmapindexe erhält man nur eine lange Liste von TIDs
- Großer Gewinn bei zusammengesetzten Bedingungen
  - Bitoperationen sehr schnell
  - Bitmatrizen sehr klein
- Außerdem: Index ist ordnungsunabhängig

Nachteile

- Lohnt nur unter diversen Annahmen
  - Geringe Kardinalität der Attribute
  - Vor allem Punktanfragen
  - Wenig Veränderungen in der Datenbank
    - INSERT, DELETE in Faktentabelle – Reorganisation der Indexe
    - Neue (UPDATE) Attributwerte – Reorganisation der Indexe
- Großer Platzbedarf bei hohem |A|
  - Beispiel: n=1.000.000, a_1=50, a_2=100, TID= 4 Byte
  - B*-Bäume: 2*4*1.000.000 = 8MB
  - Bitmap: 1.000.000 * (50+100)/8 = 18,75MB

Variante 1: Compressed Bitmap Index

- Ziel: Platzreduktion bei Attributen hoher Kardinalität
- Viele Komprimierungsverfahren möglich
  - Hohe Kompressionsraten bei hoher Attributkardinalität möglich durch „leere“ Bitarrays
- Arten der Verwendung von Kompression
  - Dekomprimierung beim Laden
    ⇒ Vorteil: Schnelle Queries
    ⇒ Nachteil: Hoher Speicherverbrauch, Swapping
  - Dekomprimieren bei Queryverarbeitung
    ⇒ Vorteil: Geringer Platzverbrauch
    ⇒ Nachteil: Performanceverlust bei Queries

RLE Beispiel

- Beispiel: Run-Length Encoding (RLE)
  - n=1.000.000, a=100, TRID: 4 Byte
  - Pro Bitarray ist nur einer von 100 Werten eine „1“
  - Explizites Speichern der Positionen p_i mit 1: $p_1,p_2,p_3,p_4,\ldots$
  - Bitmap ohne RLE: 1.000.000 * 100/8 = 12,5 MB
  - Bitmap mit RLE
    - 1.000.000 ist durch 20 Bit adressierbar

      1.000.000 * (20/8) /100*100 = 2.5MB
- Vorsicht vor Sperren bei komprimierten Bitmaps
  - Sehr viele Bits in einer Page
  - Sperrung der Page kann zig-tausend Tupel sperren
  - Geeignet nur für Read-Only Umgebungen
- Nachteil RLE: Teure Dekomprimierung
  - Sonst keine Bitoperationen möglich

Variante 2: Verwendung anderer Zahlenbasen

- Beobachtung
  - Zahlen zwischen 1-15, Binärkodierung: 4 Bit
  - Zahlen zwischen 1-15, Bitmap: 15 Bit
- Vorteile Bitmapdarstellung
  - Direkter Test auf Gleichheit möglich (1 Bit)
  - Zusammengesetzte Bedingungen durch schnelle Bitoperationen berechenbar
- Vorteile Binärdarstellung
  - Geringer Speicherverbrauch
- Kann man beides vereinen? Kompromisse finden?
  - Idee: Verwendung unterschiedlicher Zahlenbasen

Verwendung anderer Zahlenbasen

	Darstel-lung	Werte-bereich	Speicherbedarf (binär pro Ziffer)	Speicherbedarf (Bitmap pro Ziffern)
Dezimal	<10,10,10>	10^3	3*4=12 Bit	30 Bit
Binär	<2,2,2>	2^3	3 Bit	6 Bit
Allge-mein	<a,b,c>	a*b*c	ceil(log(a))+ ceil(log(b))+ ceil(log(c)) Bit	a+b+c Bit

- Darstellung einer Zahl zur Zahlenbasis <a,b,c>
  - $x = <x_1,x_2,x_3>$
    - Mit x_1=x div (b*c), x_2=(x-b*c*x_1) div c, x_3=x-b*c*x_1-c*x_2
  - $<x_1,x_2,x_3> = x_1*(b*c) + x_2*c + x_3$

Beispiel

- Attribut mit 20 möglichen Werten
- Bitmap <1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1>

1	1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
18	0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0
12	0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0
11	0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0

- Bitmaps zur Basis <2,4,3>
  - 2*4*3 = 24 Werte

```
 1 = 0*12 + 0*3 + 1
18 = 1*12 + 2*3 + 0
12 = 1*12 + 0*3 + 0
11 = 0*12 + 3*3 + 2
```

```
0 1 0 1 2 3 0 1 2
1,0,1,0,0,0,0,1,0
0,1,0,0,1,0,1,0,0
0,1,1,0,0,0,1,0,0
1,0,0,0,0,1,0,0,1
```

2 4 3

Beispiel – Zugriff

- Darstellung von 20 unterschiedlichen Werten
  - $2^5 < 20 < 2^6$
- Original Bitmap
  - Speicherverbrauch — 20 Bit
  - Vergleich mit Konstanter — 1 Bitzugriff
- Bitmap zur Basis <2,4,3>
  - Speicherverbrauch — 9 Bit
  - Vergleich mit Konstanter — 3 Bitzugriffe + 2 AND
- Bitmap zur Basis <2,2,2,2,2,2>
  - Speicherverbrauch — 12 Bit
  - Vergleich mit Konstanter — 6 Bitzugriffe + 5 AND
- Binäre Kodierung
  - Speicherplatzoptimal — 6 Bit
  - Suche eines Wertes — 6 Bitzugriffe + 5 AND

Fazit Bitmapindexe

- Mächtige Struktur für DWH Anwendungen
  - Attribute mit geringen Kardinalitäten
  - Häufige zusammengesetzte Bedingungen
  - „Read-Only“ Umgebung
- Durch Wahl der Zahlenbasis ist (anwendungs-) optimales Austarieren von Speicherverbrauch und Zugriffsaufwand möglich
- Kommerzielle RDBMS bieten nur einfache und komprimierte Bitmapindexe

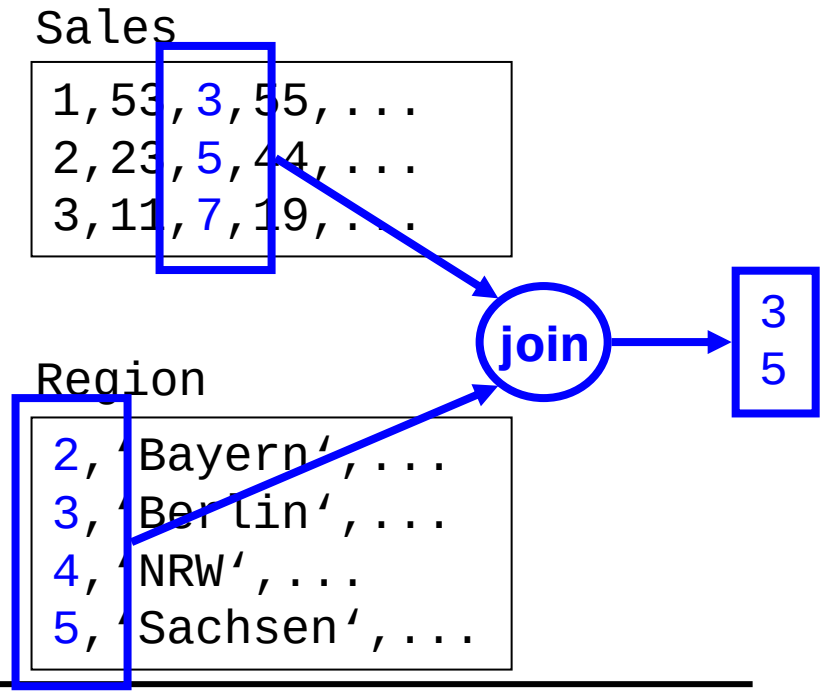
Teil 3. Join Index

- Joins sind teure Operationen
  - Bewegung vieler Daten
  - Viele Alternativen für Optimierer
  - U.U. große Zwischenergebnisse trotz kleiner Endergebnisse
- Beobachtung bei DWH
  - Es werden immer und immer wieder ähnliche Joins ausgeführt: Fakten mit Dimensionstabellen
- Idee
  - „Materialisierung“ von Joins

Joins

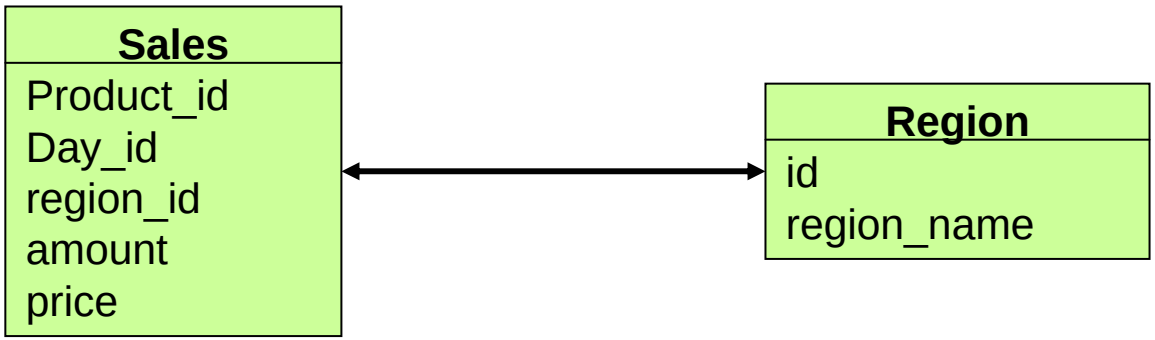
- Viele Algorithmen
  - Nested Loop
  - Blocked Nested Loop
  - Sort-Merge
  - Hash
  - ...
- Index-Join
  - Lesen zweier TID Listen
  - Berechnen des Schnittmenge
  - „Nachladen“ weiterer Attribute aus beiden Tabellen nur für Treffer

```
SELECT R.name, ...
FROM sales S, region R
WHERE  S.region_id=R.id AND
       ...
```



Join Index

- Indexierung von Spalten einer Tabelle A mit Werten einer Tabelle B




```
CREATE INDEX myIndex ON sales (region.region_name)
USING sales.region_id = region.id
```

(Redbrick - nicht in Oracle)

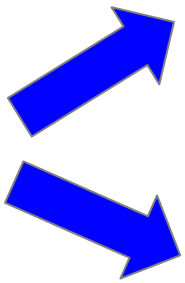
Beispiel

Sales

ROWID	id	region_id	amount
0x001	101	11	200
0x002	101	12	210
0x003	102	11	190
0x004	102	12	195
0x005	103	13	95

Region

ROWID	id	name	…
0x100	10	Berlin	
0x101	11	Bayern	
0x102	12	Sachsen	
0x103	13	NRW	



0x001: 0x101
0x002: 0x102
0x003: 0x101
0x004: 0x102
0x005: 0x103

… oder …

0x100: {}
0x101: {0x001, 0x003}
0x102: {0x003, 0x004}
0x103: {0x005}

Beispiel 2

0x001: 0x101
0x002: 0x102
0x003: 0x101
0x004: 0x102
0x005: 0x103

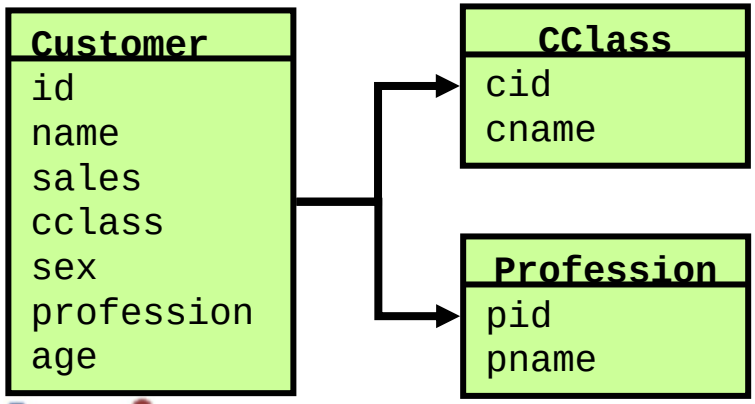
- Speicherung in SALES
- „N“ Seite der 1:n Beziehung
- Bei Join mit REGION und Bedingung an REGION.NAME
  - Table Scan von SALES reicht

0x100: {}
0x101: {0x001, 0x003}
0x102: {0x003, 0x004}
0x103: {0x005}

- Speicherung in REGION
- „1“ Seite der 1:n Beziehung
- Bei Join mit SALES und Bedingung an REGION.NAME
  - Schnittmengenbildung ohne Zugriff auf SALES

Bitmapped Join Index

- Unsere bisherigen Beispiele für Bitmapindexe waren nur zum Teil direkt anwendbar
  - Bedingungen nicht an FK, sondern an „semantische“ Attribute der Dimensionstabellen gerichtet
  - Bitmaps auf FK verhindern dann nicht den Join zu Dimensionstabellen
- Lösung: Bitmap Join Index



- Bitmaps erstellen für Tupel in Customer für alle Werte in `cclass.cname` und in `profession.pname`
- Nur auf der „N“ Seite sinnvoll

Bitmap Join Indexe in Oracle

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

- Macht Joins (manchmal) unnötig
- Verknüpfung mit anderen Bitmapindexen auf `sales` möglich
- Indexierung mehrerer Spalten, mehrerer Joins möglich

```
SELECT SUM(sales.amount)
FROM sales, region
WHERE  sales.region_id = region.id AND
       region.region_name = ‚Berlin‘
```

Fazit

- Vorberechnung von Joins in Form von Indexen
- Speziell Bitmap Join Indexe sehr populär
  - Haben die meisten DB Hersteller
- Stellen einige Anforderungen an den Optimierer
- Unterhalt kostet Ressourcen

Teil 4. Multidimensionale Indexstrukturen

- Eine eigene Vorlesung wert
- Teilweise in DBS-II besprochen
- Kommerziellen Implementierungen
  - Nur R-Trees
  - Anwendung: Spatial Data Structures

➢ Thema wird übersprungen

Zusammenfassung

- Indexierung sehr wichtig für hohe Performance
- Immer eine Abwägung
  - Positiv: Schnellerer Zugriff
  - Negativ: Platz, Zeitbedarf für Aktualisierung
- Auswahl der besten Indexe (Art und Attribute) schwierig
  - Abhängig von Workload
  - RDBMS bieten oftmals Indexassistenten
- Weitere Themen
  - Indexaktualisierung (z.B. bei BULK Loads)
  - Algorithmen zur Indexauswahl

Literatur

- [Leh03]: Kapitel 8.4, 8.5.1
- Ming-Chuan Wu, Alejandro P. Buchmann: Encoded Bitmap Indexing for Data Warehouses. ICDE 1998: 220-230
- Chee Yong Chan, Yannis E. Ioannidis: An Efficient Bitmap Encoding Scheme for Selection Queries. SIGMOD Conference 1999: 215-226
- Marcus Jürgens, Hans-Joachim Lenz: Tree Based Indexes Versus Bitmap Indexes: A Performance Study. IJCIS 10(3): 355-376 (2001)
- Gaede, V. and Günther, O. (1998). "Multidimensional Access Methods." *ACM Computing Surveys* **30**(2): 170-231.