

# Data Warehousing und Data Mining

Indexierung

Ulf Leser

Wissensmanagement in der  
Bioinformatik



# Datenqualität [Spiegel Online, 30.5.2007]

---

- **Bundesagentur meldet zu geringe Arbeitslosenzahlen**
- Die offiziellen Arbeitsmarktdaten der vergangenen Monate waren falsch. Knapp 40.000 Arbeitslose sind nicht in die Statistik eingeflossen, gab die Bundesagentur für Arbeit heute zu. **Ein ganzer Datensatz war einfach verloren gegangen.**
- Nürnberg - Zwischen der amtlichen Statistik und der tatsächlichen Arbeitslosenzahl gab es eine Differenz, sagte der Vorstandsvorsitzende der Bundesagentur für Arbeit, Frank-Jürgen Weise, heute in Nürnberg bei der [Vorstellung der aktuellen Zahlen. \(mehr...\)](#) Wegen einer Panne bei der Datenübertragung habe die Bundesagentur monatelang zu niedrige Arbeitslosenzahlen gemeldet.
- Angefangen hat alles mit einem Fehler im Dezember 2006. Nach Angaben Weises waren damals knapp 40.000 erfasste Arbeitslose nicht in die Arbeitsmarktstatistik eingeflossen. "Ein ganzer Datensatz ist nicht verarbeitet worden", sagte er. Besonders pikant: Das in solchen Fällen automatisch erstellte **Fehlerprotokoll blieb von den Mitarbeitern der Bundesagentur unbeachtet** liegen.
- Die Statistik vom Januar weist dadurch beispielsweise 37.500 weniger Erwerbslose aus, als es tatsächlich gab. Die Differenz zwischen der amtlichen Statistik und der tatsächlichen Arbeitslosenzahl habe sich allerdings in den darauf folgenden Monaten nach und nach verringert und betrage im Mai nur noch 6000, sagte Weise. Der Grund sei, dass viele der nicht erfassten Jobsucher inzwischen eine Stelle gefunden hätten.
- "Die Differenz in der Arbeitslosenstatistik ändert aber nichts an unserer generellen Arbeitsmarkteinschätzung und dem Trend", sagte Weise. Die amtliche Statistik werde nun nachträglich korrigiert. Außerdem werde künftig bei auftauchenden Fehlermeldungen die Verarbeitung von Arbeitslosendaten sofort gestoppt.
- Weise zufolge deckte erst eine **Plausibilitätskontrolle** im April die Statistik-Lücke auf. Es sei aufgefallen, dass sich erfolgreiche Jobsucher bei den Arbeitsagenturen abgemeldet hätten, die statistisch gar nicht erfasst waren.

# Inhalt dieser Vorlesung

---

- Indexierung
- Wiederholung: B\*-Bäume
- Indexierung mit Bitmaps
- Join-Indexe

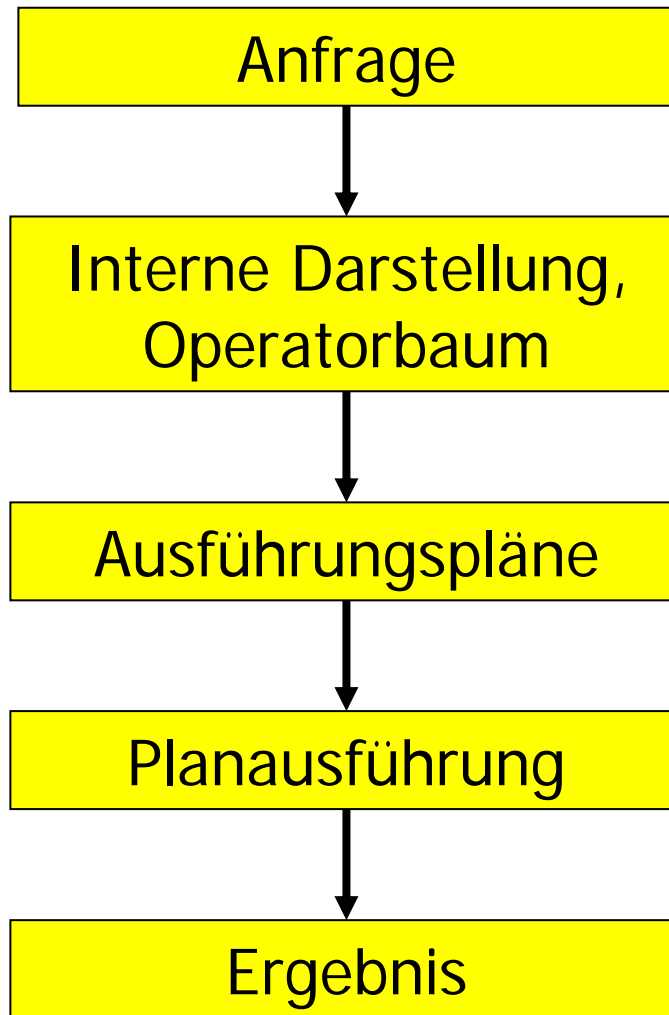
# Wo sind wir?

---

- Bisher: **Benutzerebene** eines DWH
  - Multidimensionales Datenmodell
  - OLAP Anfragen
  - ETL Prozesse
- Für die nächsten Stunden: **Performanz**
  - Wie kann man eine Anfrage schnell ablaufen lassen?
  - **Ein- und multidimensionale Indexierung**
  - Spezielle Joinmethoden
  - Berechnung großer CUBEs
  - Partitionierung
  - Materialisierte Sichten

# Anfragebearbeitung in RDBMS

---



Parsing; Syntaktischer und semantischer Check

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

Dynamische Programmierung; Kostenbasierte Optimierung

Synchronisierung, Pipelining, Caching, ...

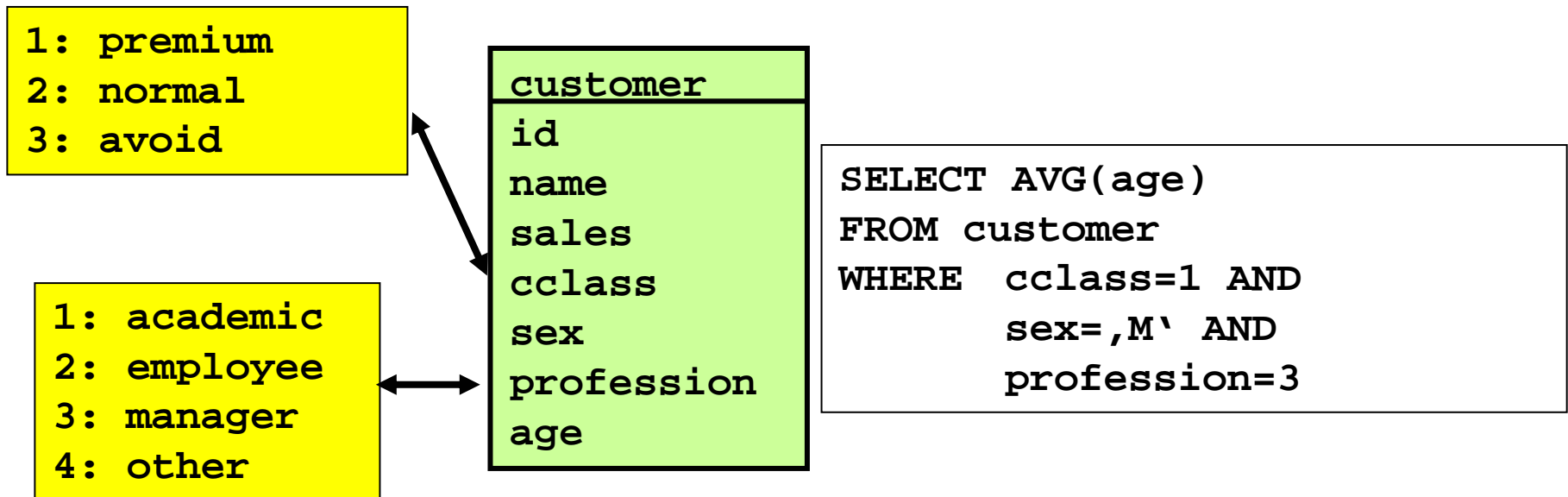
# Anfragemuster

---

- Punktanfragen
  - Selektion mit Primärschlüssel, Ergebnis ist ein einzelnes Tupel
  - In DWH's eher untypisch
- Bereichsanfragen (range queries)
  - ... **WHERE day>10 AND day<20**
  - (Zwischen-)Ergebnis sind viele, viele Tupel
  - In DWH sehr häufig, oft in Kombination mit Aggregation
- Aggregation (und Gruppierung)
  - Erfordern meistens Zugriff auf sehr viele Fakten
  - Oftmals viele voneinander abhängige Aggregate in einer Anfrage
- Multidimensionale Anfragen
  - Gruppierungen /Bedingungen über mehrere Attribute
  - ... **WHERE day>10 AND amount>40 AND shop\_id>100**
- Joinanfragen
  - Kritisch, da potentiell sehr große Zwischenergebnisse
- Normal: Kombinationen aus allem

# Zugriff auf Faktentabelle

- Faktentabelle sehr, sehr groß – kritischer Engpass
- Typischer Zugriff
  - Bedingungen/Gruppierung auf Dimensionsattributen
    - Erfordert (meistens) einen Join pro Dimension
  - Aggregation von Fakten (hierarchisch, mehrdimensional)



# Selektivität von Anfragen

---

- Kein Index
  - 1.000.000 Einträge
  - Daten von 4 Attributen benötigt
  - Also: Alle Datenblöcke lesen, full table scan
- Indexierung
  - Annahme: **Gleichverteilung** der Wert
  - Index auf `sex`: 50% Selektivität
  - Index auf `cclass`: 33% Selektivität
  - Index auf `profession`: 25% Selektivität
- Zusammen: **~4% Selektivität**
  - Unter Annahme der Unabhängigkeit der Attribute
  - **Indexzugriff lohnt sich ab 7%**
- Aber: Kein Einzelindex ergibt ausreichende Selektivität
  - Optimierer wählt Full Table Scan
  - Verschrenktes Potential

# Prinzip eines Index

---

- Daten liegen **ungeordnet** in Datenblöcken
- Index
  - Eigene persistente Datenstruktur
  - Speichert **geordnete Liste aller Werte eines Attributs** zusammen mit Zeigern auf die physikalischen Adressen der Tupel (TID)
    - Es gibt viele Arten von „Ordnung“
    - B\*-Bäume: Einfache Sortierung
  - **Reduktion der Zugriffszeit** auf alle Tupel, bei denen das Attribut einen gegebenen Wert hat
    - Punktanfrage auf Primärschlüssel mit sortiertem Index:  $O(\log(n))$
- Management
  - Aktualisierung des Index mit Aktualisierung der Tabelle
  - Index muss vom Benutzer explizit angelegt werden (Wizards)

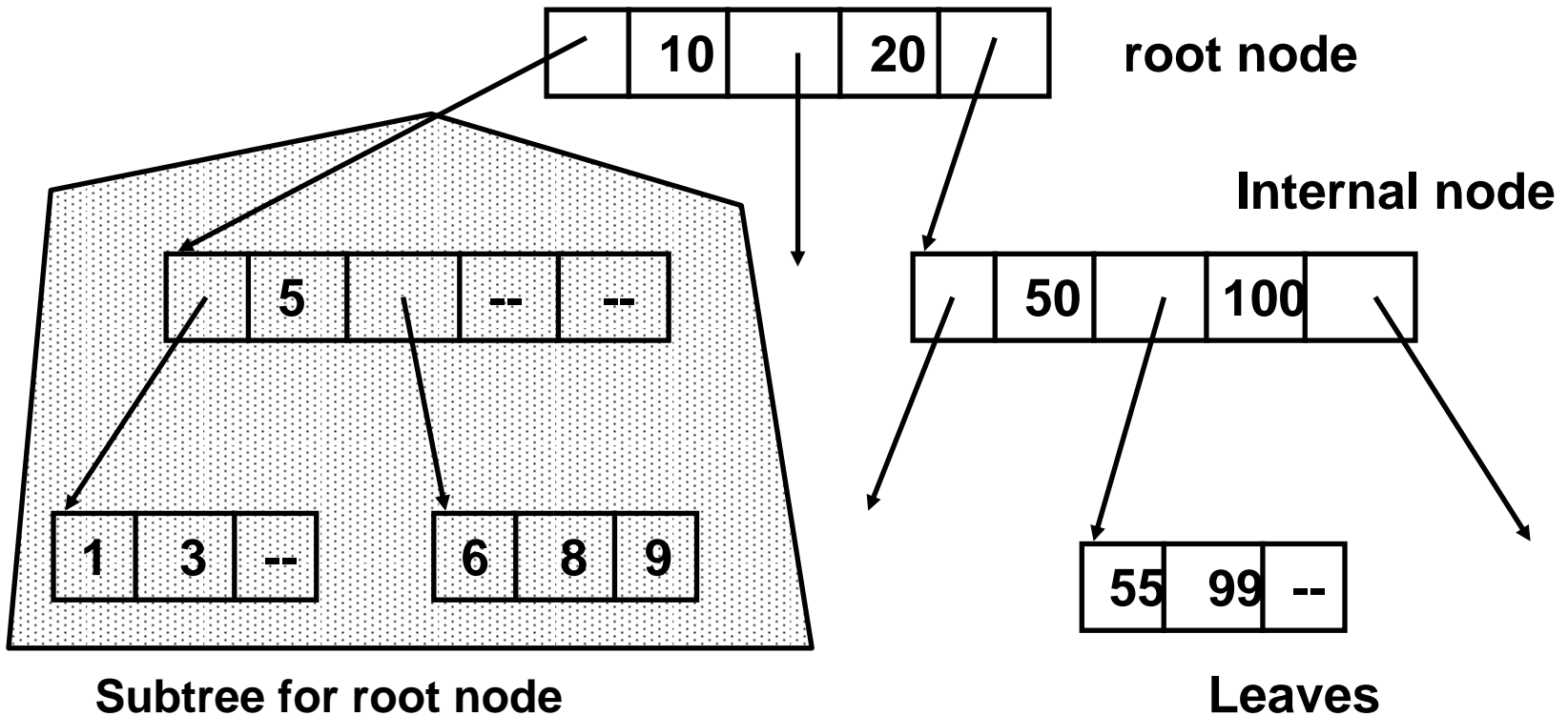
# Inhalt dieser Vorlesung

---

- Indexierung
- Wiederholung: B- und B\*-Bäume
  - Grundidee
  - Bulk-Loading eines B\*-Index
  - Oversized, zusammengesetzte, degenerierte, user-defined, ...
- Indexierung mit Bitmaps
- Join-Indexe

# B-Trees

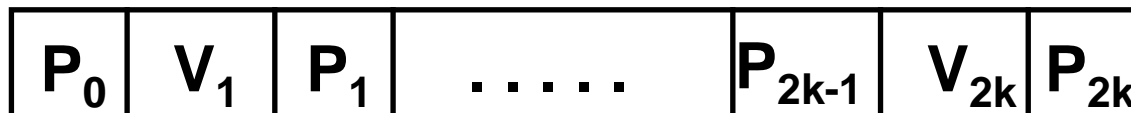
- Balanced index with **variable number of levels**
  - **Adapts** to table growth / shrinkage



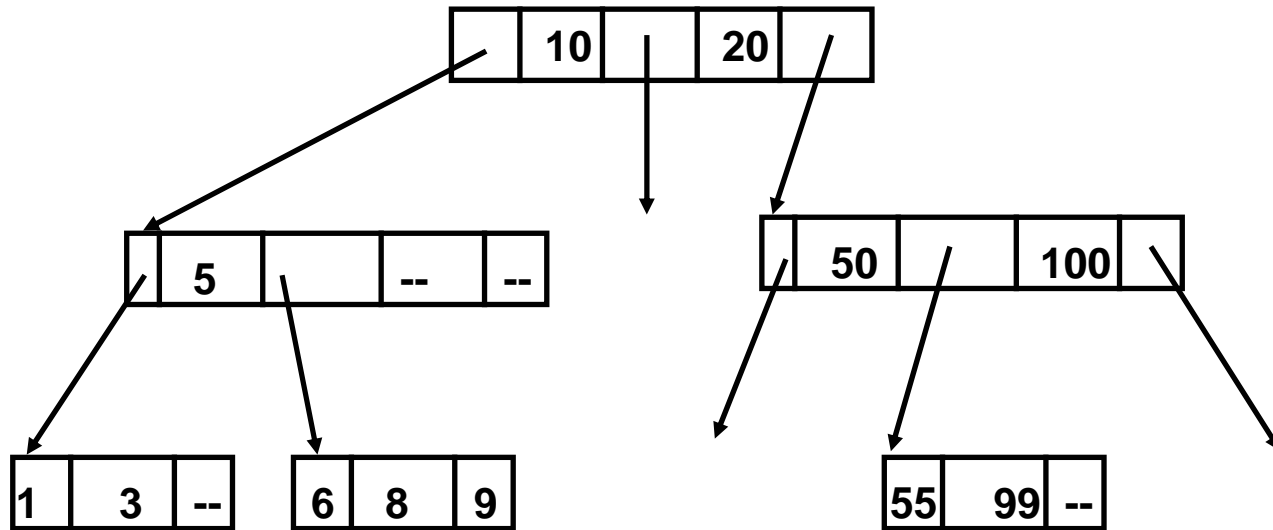
# Formally

---

- We assume an index on a primary key (no duplicates)
- Tree structure
  - Internal nodes contain pairs (value, TID) and pointers to other nodes
  - Leaf nodes only contain (value, TID)
- We assume we can fit  $2k$  combinations of (pointer, value, TID) plus one pointer into one block
- Each internal node contains between
  - $k$  and  $2k$  pairs (value, TID)
  - $k+1$  and  $2k+1$  pointers to subtrees
    - Subtree left of position  $m$  in node contains only and all values  $y < V_m$
    - Subtree right of position  $m$  in node contains only and all values  $V_m$  with  $y > V_m$
    - $V_m < V_{m+1}$
  - Exception: Root node
- Note: B-trees use always at least 50% of allocated space



# Searching B-Trees



Find 8

1. Start with root node
2. Go left
3. Go right
4. Found

Find 60

1. Start with root node
2. Go right
3. Follow middle ptr.
4. Not found

# Complexity

---

- B-trees are **always balanced**
- Assume  $n$  keys; let  $r = |\text{value}| + |\text{TID}| + |\text{pointer}|$
- One block maximally can hold  $2k$  records of size  $r$ 
  - (+ one pointer)
- Best case: Suppose all nodes are full
  - We have  $b = n \cdot r / 2k$  blocks
    - Actually somewhat less, since leaves contain no pointers
  - The height  $h$  of the tree is  $\sim \log_{2k}(b)$
  - Search requires **between 1 and  $\log_{2k}(b)$  IO** ( $O(\log_{2k}(n))$ )
- Worst case: All nodes contain only  $k$  values
  - We need  $b = n \cdot r / k$  blocks
  - The height of the tree is  $h \sim \log_k(b)$
  - Search requires **between 1 and  $\log_k(b)$  IO** ( $O(\log_k(n))$ )

# Example

---

- Imagine

- $|value|=20$ ,  $|TID|=16$ ,  $|pointer|=8$ , block size=4096
- Assume  $n=1.000.000.000$  (1E9)
- We have  $r=44$
- We have between 46 and 92 index records per block
- Hence, we need **between 5 and 6 IO**
  - By caching the first two levels (between  $1+46$  and  $1+92$  blocks), this reduces to **3 to 4 IO**

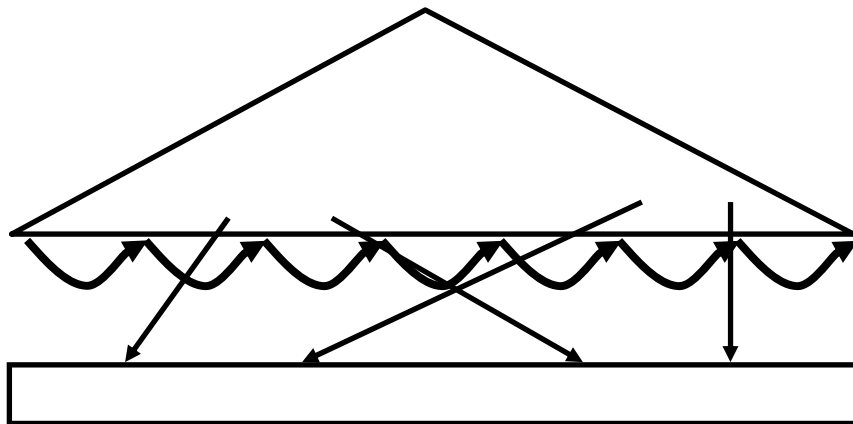
# B-trees on Non-Unique Attributes

---

- B-trees store pairs (value, TID)
- If duplicates exist, we have two options
- Compact representation
  - Store (value, TID<sub>1</sub>, TID<sub>2</sub>, ... TID<sub>n</sub>)
    - Difficult to handle – internal nodes cannot keep fixed number of pairs any more
    - Requires **internal overflow blocks**
- Verbose representation
  - Treat duplicates as different values
  - Generates a tree although a list would suffice
- Better: **B\* trees**

# B\*-Trees

- Definition of B\*-Trees
  - Only records in leaves are pairs (value, TID)
  - Records in internal nodes are pairs (border value, pointer)
    - Borders between ranges of values in underlying subtrees
    - Border values need not exist in the database - only **signposts**
  - Leaves are connected by pointers – faster range queries



B\*-Tree

Data file organized  
as heap file

# Advantages

---

- Simpler operations
  - But every search must reach a leaf node
- No TIDs in internal nodes
  - More records per page in internal nodes
    - Better space usage
    - Increased fan-out, reduced average height
    - No problem with duplicate values (hidden in heap files)
  - This is the main advantage of B\* trees
- Not necessarily “real” values in internal nodes
  - Can be used to save further space – Prefix B\*-Tree (for text)
- Linked leaves
  - Faster range queries
  - Optimally, leaf blocks (and heap blocks) are in sequential order on disk

# Loading a B\*-Tree

---

- What happens in case of

```
CREATE INDEX myidx ON verylargetable(id);
```

- ?

# Loading a B\*-Tree

---

- What happens in case of

```
CREATE INDEX myidx ON verylargetable(id);
```

- **Record-by-record** insertion
  - Each insertion has  $3h+2 = O(\log_k(b))$  block IO
  - Altogether:  $O(n \cdot \log_k(b))$
- Blocks are read and written in arbitrary order
  - Very likely: bad cache-hit ratio
- Space usage will be anywhere between 50 and 100%
- Can't we do better?

# Bulk-Loading a B\*-Tree

---

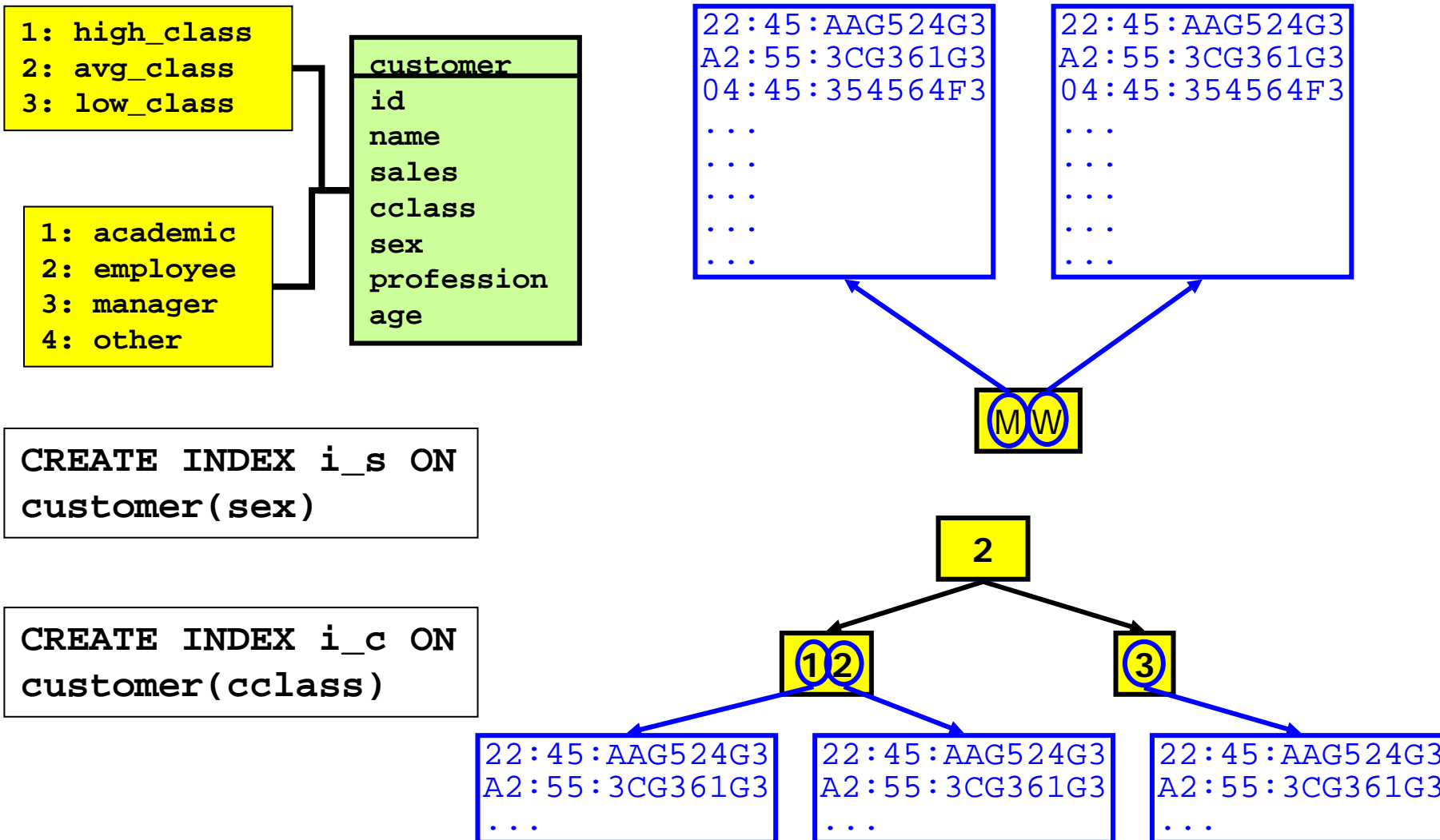
- First **sort records**
  - $O(n \cdot \log_m(n))$ , where  $m$  is number of records fitting into main memory
  - Clearly,  $m \gg k$
- **Insert in sorted order** using normal insertion
  - Tree builds from lower left to upper right
  - **Caching will work very well**
  - But space usage will be only around 50%
- **Alternative**
  - **Compute structure in advance**
    - Every  $2k$ 'th record we need a separating key
    - Every  $2k$ 'th separating key we need a next-level separating key
    - ...
  - Can be generated and written in linear time

# 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: Anfrage muss ein **Präfix der indizierten Attributreihenfolge** enthalten

```
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`
```

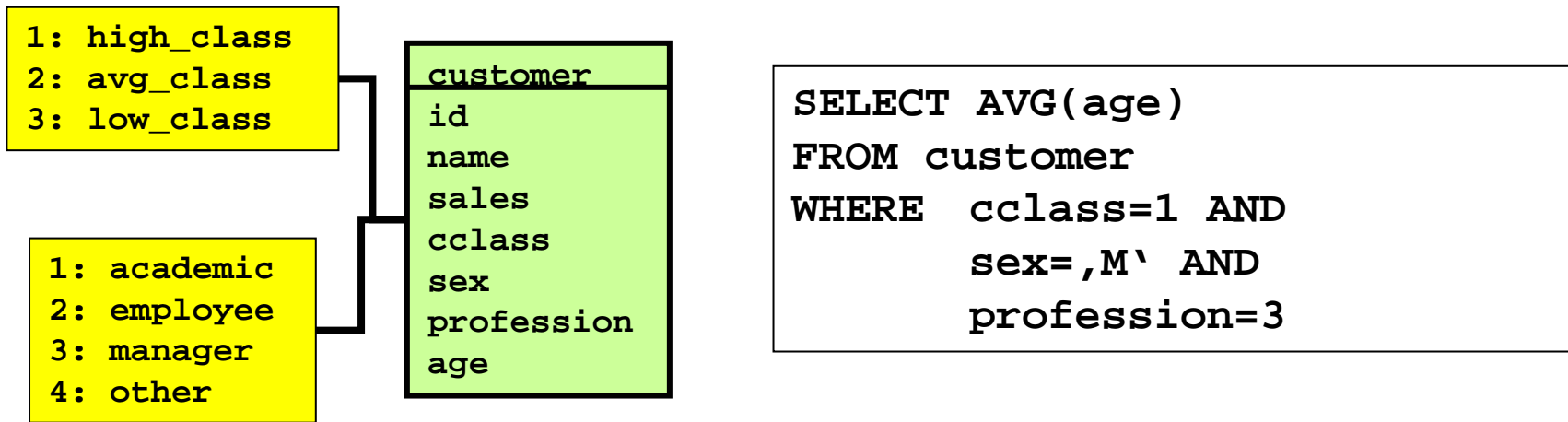
```
SELECT ...
FROM ...
WHERE cclass=1 AND
      prof=',other`
```

# Abhilfe

---

- Attribute mit geringer Kardinalität  
⇒ Bitmap-Indexe
- Queries auf hochdimensionalen Daten  
⇒ Multidimensionale Indexe
- Zuerst aber
  - „Tricks“ mit B\*-Indexen

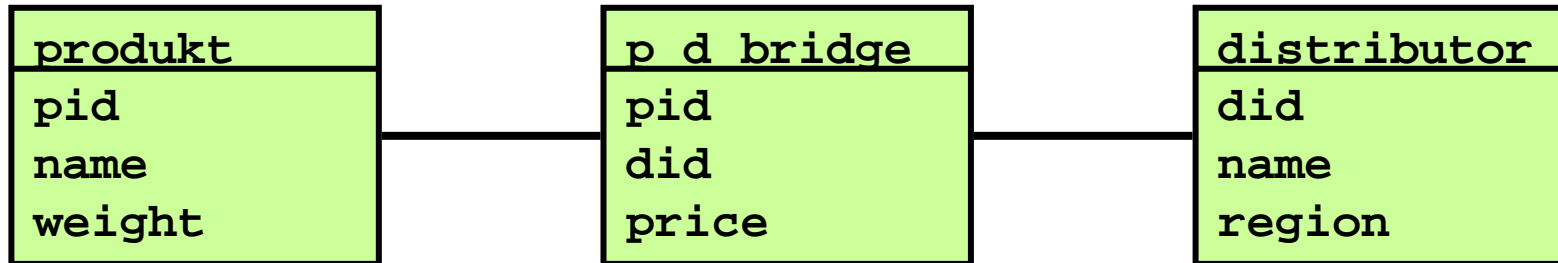
# Oversized Indexe



- 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
  - Schnelles **sortiertes sequentielles Lesen**

# Berechnete Indexe

customer
id
name
sales
cclass

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

```
SELECT id, ...  
FROM customer  
WHERE  
  upper(name) = ,SCHMIDT` ;
```

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

- c\_name wird nicht benutzt
- Besser

- c\_name wird nicht benutzt
- Besser

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

```
CREATE INDEX c_name ON  
customer(soundex(name)) ;  
SELECT id, ...  
FROM customer  
WHERE soundex(name) = soundex(,Meier`);
```

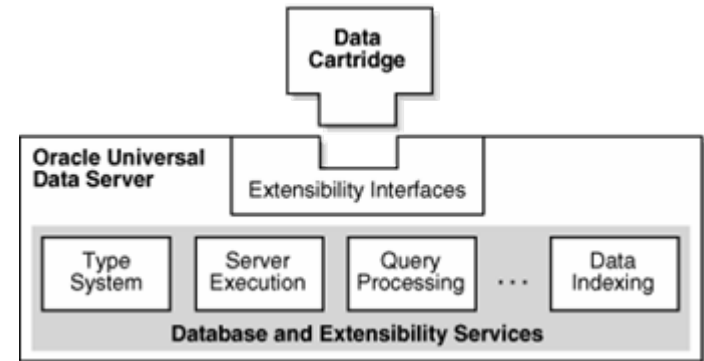
# Mehr zu Indexen

---

- 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: RDBMS 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
  - Erfahrung: Flaschenhals Interface RDBMS / PlugIn

# Oracle Data Cartridge Index Interface (ODCII)

- Index-Definition
  - Create, Drop, Alter ...
- Index-Pflege
  - Insert, Delete, Update
- Index-Anfragen
  - keine Überladung der vorhandenen Operatoren möglich
  - Berechnung der Prädikate in Where-Klausel
- Keine Implementierung eigener Join-Funktionen
- Oracle-CBO (Cost Based Optimizer) API
  - statisch oder dynamisch
- Hinter dem Interface
  - > auf Daten via SQL-Schnittstelle (JDBC)
  - < ROWIDs



# Inhalt dieser Vorlesung

---

- Indexierung
- Wiederholung: B- und B\*-Bäume
- Indexierung mit Bitmaps
  - Grundidee
  - Komprimierung
  - Wechsel der Zahlenbasis
- Join-Indexe

# Bitmap-Index

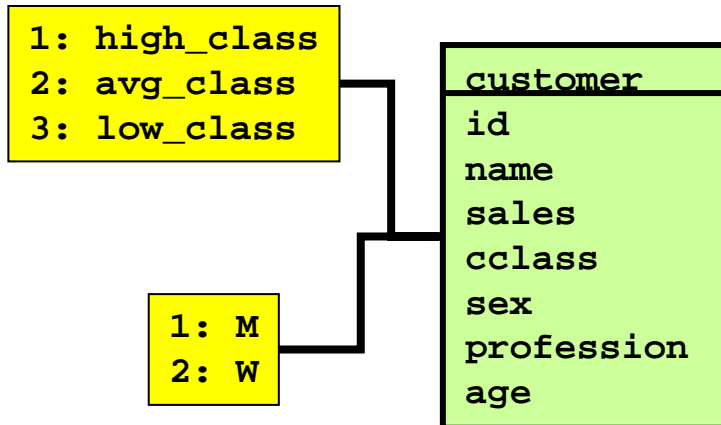
---

- Adressierte Probleme
  - Indexierung von Attributen **niedriger Kardinalität**
    - B\*-Bäume entarten
  - Auswertung von Bedingungen auf mehreren Attributen
    - Zusammengesetzte B\*-Indexe sind ordnungssensitiv
- Kernidee
  - Repräsentation von Attributwerten als **Bitmatrizen**
    - Niedrige Kardinalität – kleine Matrizen
    - Können sehr kompakt gespeichert werden
  - AND/OR Bedingungen durch **Bitoperationen** implementierbar
    - Sehr schnell ausführbar

# Grundaufbau

- Tabelle T mit Attribut A,  $|T|=n$ ,  $|A|=a$  verschiedene Werte
- Repräsentation jedes der a Werte von A als **Bitarray** der Länge n
  - Attribut X, Wert z:  $X.z[i]=1$  gdw i.tes Tupel hat Wert z in Attribut X
  - Die **Ordnung der Tupel** muss feststehen (später mehr)
- Repräsentation von A: **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,...
...
```

Bitmap-Index cclass?

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

Bitmap-Index 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
    - $2 \cdot 4 \cdot 1.000.000 = 8\text{MB}$
    - In jedem Index ist jede TID einmal repräsentiert
    - Speicherplatz unabhängig von Kardinalität der Attribute
  - Bitmap-Index
    - $1.000.000 \cdot (4+2)/8 = 0,75\text{MB}$
- Vorteile
  - Kleiner Index passt eher in **Hauptspeicher**
  - Kleine Indexe können schneller von Platte gelesen werden

# Vorteil 2 – Komplexe Bedingungen

---

- Bedingungen an **mehrere Attribute**
  - B\*-Index
    - Lesen der TID Liste zu jeder Bedingung
    - Schnittmengenbildung aller Listen durch Sortierung oder Hashing
      - **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 geringer Selektivität
  - Auch durch Bitmap-Indexe erhält man nur eine lange Liste von TIDs
- Großer Gewinn bei **hoher Selektivität**
  - **Bitoperationen sehr schnell**
  - Bitmatrizen sehr klein (viel kleiner als B\*-Bäume)
- Außerdem: **Ordnung der Attribute** im Index ist egal

# Nachteile

---

- Lohnt nur wenn
  - „Geringe“ Kardinalität der Attribute
  - Anfragen haben hohe Selektivität
  - Häufige zusammengesetzte Anfragen
- Möglichkeiten
  - `SELECT AVG(*) FROM cube GROUP BY day_id`
  - Tabelle nach `day_id` sortieren, `GROUP BY` ausrechnen
  - Index auf `day_id` – random access Zugriff auf `cube`
    - Oversized Index oder IOT wäre hier gut
  - Bitmap-Index auf `day_id` – random access Zugriff auf `cube`
- Großer Platzbedarf bei hohem  $|A|$ 
  - Beispiel:  $n=1.000.000$ ,  $a_1=50$ ,  $a_2=100$ , TID= 4 Byte
  - B\*-Bäume:  $2*4*1.000.000 = 8\text{MB}$
  - Bitmap?

# Nachteile

---

- Lohnt nur wenn
  - „Geringe“ Kardinalität der Attribute
  - Anfragen haben hohe Selektivität
  - Häufige zusammengesetzte Anfragen
- Möglichkeiten
  - `SELECT AVG(*) FROM cube GROUP BY day_id`
  - Tabelle nach `day_id` sortieren, `GROUP BY` ausrechnen
  - Index auf `day_id` – random access Zugriff auf `cube`
    - Oversized Index oder IOT wäre hier gut
  - Bitmap-Index auf `day_id` – random access Zugriff auf `cube`
- Großer Platzbedarf bei hohem  $|A|$ 
  - Beispiel:  $n=1.000.000$ ,  $a_1=50$ ,  $a_2=100$ , TID= 4 Byte
  - B\*-Bäume:  $2*4*1.000.000 = 8\text{MB}$
  - Bitmap:  $1.000.000 * (50+100)/8 = 18,75\text{MB}$

# Management von Bitmap-Indexen

---

- Benötigen eine  **feste Ordnung**  der Tupel einer Tabelle
- Die ist aber für gewöhnlich so oder so gegeben
  - Löschen eines Tuples:
    - **Platz wird freigegeben**  (Grabstein)
    - Ordnung der Tupel ändert sich nicht
    - In allen Bitarrays alle Werte auf 0 setzen
  - Einfügen eines Tupels
    - Entweder: Ersetzen eines vorherigen Grabsteins
    - Oder: Hintenanfügen
      - Das verlängert alle Bitarrays um 1
      - (Bei Komprimierung unter Umständen implizit zu erledigen)
- Schwieriger sind  **Reorganisierungen** 
  - „Vacuum“ oder wachsende / schrumpfende Tupel
- Spezialfall: Tupel mit neuem Wert für A
  - Anlegen eines neuen Bitarrays mit einer 1, sonst 0

# Komprimierte Bitmap-Indexe

---

- Ziel: Platzreduktion bei Attributen hoher Kardinalität
- Viele Komprimierungsverfahren möglich
  - Hohe Kompressionsraten bei hoher Attributkardinalität möglich durch „leere“ Bitarrays
- Bitmaps müssen für die Anfragebearbeitung dekomprimiert werden
  - Zur Ladezeit (und bleiben dekomprimiert im Cache)
    - Vorteil: [Schnelle Queries](#)
    - Nachteil: [Hoher Speicherverbrauch](#)
  - Zur Anfragezeit (und liegen komprimiert im Cache)
    - Vorteil: [Geringer Platzverbrauch](#)
    - Nachteil: [Performanceverlust](#) bei Queries
- [Sperren bei komprimierten Bitmaps](#)
  - Sehr viele Bits in einer Page
  - Sperrung der Page kann zig-tausend Tupel sperren
  - Geeignet nur für Read-Only Umgebungen

# Run-Length-Encoding (RLE)

---

- RLE1: **Explizites Speichern der 1-Positionen**
  - Beispiel:  $n=1.000.000$ ,  $a=100$ , TID: 4 Byte
  - Pro Bitarray ist nur einer von 100 Werten eine „1“
    - Annahme: Gleichverteilung der Werte von A über T
  - Bitmap ohne RLE:  $1.000.000 * 100/8 = 12,5 \text{ MB}$
  - Bitmap mit RLE
    - $1.000.000$  ist durch 20 Bit adressierbar
    - $1.000.000 * (20/8) / 100 * 100 = 2.5 \text{ MB}$
- **Nachteil**
  - Wir müssen die **Größe des Arrays** von vorneherein festlegen
    - Um die Anzahl der Bits für Kodierung festlegen zu können
  - Wenig adaptiv für schrumpfende / wachsende Tabellen

# Variante 2: RLE2

---

- Weg von festen Adressgrößen
- Wir speichern die **Länge der 0-Blöcke**
  - 0000110100001010100010000000100010000
  - Blöcke 4,0,1,4,1,1,3,7,3
  - Jeder Block repräsentiert also i-mal 0 und eine 1
  - Schließende Nullen sind nur implizit gespeichert
    - Größe des Bitarray muss bekannt sein
  - Jeden Block wollen wir mit der **minimalen Anzahl Bits** speichern
- Problem: **fehlende Eindeutigkeit** bei sukzessiver binärer Kodierung
  - 000101
    - ergibt Blöcke 3,1 ergibt 111
  - 010001
    - ergibt Blöcke 1,3 ergibt 111
  - 010101 ergibt Blöcke 1,1,1 ergibt 111
- Lösungsvorschläge?



# Speicherverbrauch RLE2

---

- Wie viel Speicher brauchen wir nun?
  - Beispiel:  $n=1.000.000$ ,  $a=100$ , TID: 4 Byte
  - Im Bitarray ist jede 100e Position eine 1
  - Die durchschnittliche Blocklänge ist 99
    - Kodierung benötigt 7 Bit
  - Es gibt ca. 10.000 Blöcke
    - Pro Block:  $(7-1)+1+7$  Bit
  - Das ganze Array:  $100 \cdot (10.000 \cdot 14) / 8 = 1.75 \text{ MB}$ 
    - Erinnerung: B\*-Baum: 12,5 MB, RLE1: 2.5 MB
- Sehr grobe Schätzung
  - Blocklängen sind nicht alle gleich
  - Beispiel: 9000 Blöcke Länge 4, 1000 Blöcke Länge  $\sim 1000$ 
    - $100 \cdot ((9000 \cdot 7) / 8 + (1000 \cdot 20) / 8) = 1,03 \text{ MB}$

# Inhalt dieser Vorlesung

---

- Indexierung
- Wiederholung: B- und B\*-Bäume
- Indexierung mit Bitmaps
  - Grundidee
  - Komprimierung
  - Wechsel der Zahlenbasis
- Join-Indexe

# Vertikale Komprimierung

- Attribut A mit  $|A|=a$  verschiedenen Werten

$t_1$   $t_2$   $t_3$

```
1:00100000001010100000001010000...
2:10011110000001000011000000010...
3:01000000001000000000000010000...
...
a: ...
```

- Komprimierung bisher

```
1:14,33,45...
2:3,55, ...
3:17,34,41,...
...
a: ...
```

- Können wir auch **vertikal komprimieren** ( $b < a$ )?

$t_1$   $t_2$   $t_3$

```
1:001001100010101000000010100110...
...
b: ...
```

# Verwendung anderer Zahlenbasen

---

- Beobachtung:  $a$  Werte
  - Binärcodierung:  $\log_2(a)$  Bit
  - Bitmap:  $a$  Bit
- Gesucht seien alle Tupel mit  $A=x$  (oder  $A$  in  $\{\dots\}$ )
- Vorteile Bitmapdarstellung
  - Jeder Wert „ $x$ “ entspricht genau einem Bitarray
  - Finden aller Tupel verlangt das Lesen **eines Bitarrays**
  - Zusammengesetzte Bedingungen durch logische Operationen auf Bitarrays
- Nachteil: **Hoher Speicherbrauch** (bei großem  $a$ )
- Kann man **Kompromisse** finden?
  - Ziel: Weniger Speicherverbrauch ohne viel mehr Bitarrays lesen zu müssen
  - Idee: **Verwendung unterschiedlicher Zahlenbasen**

# Darstellung von Zahlen

	Darstellung	Wertebereich	Speicherbedarf, binär pro Ziffer	Speicherbedarf, Bitmap pro Ziffern
Dezimal	$\langle 10, 10, 10 \rangle$	$10^3$	$3 \cdot 4 = 12$ Bit	30 Bit
Binär	$\langle 2, 2, 2 \rangle$	$2^3$	3 Bit	6 Bit
Allgemein	$\langle a, b, c \rangle$	$a \cdot b \cdot c$	$\text{ceil}(\log(a)) + \text{ceil}(\log(b)) + \text{ceil}(\log(c))$ Bit	$a + b + c$ Bit

- Darstellung einer **Zahl zur Zahlenbasis  $\langle a, b, c \rangle$** 
  - $x = \langle x_1, x_2, x_3 \rangle$ 
    - Mit  $x_1 = x \text{ div } (b \cdot c)$ ,  $x_2 = (x - b \cdot c \cdot x_1) \text{ div } c$ ,  $x_3 = x - b \cdot c \cdot x_1 - c \cdot x_2$
  - $\langle x_1, x_2, x_3 \rangle = x_1 \cdot (b \cdot c) + x_2 \cdot c + x_3$

# Idee

- Repräsentation des Wertes  $x$  eines Tupels  $t$  nicht mehr durch  $a$  Bit, sondern in **Bitmapdarstellung zu einer anderen Basis**
- Beispiel:  $a=20$ ,  $t_1[a]=1$ ,  $t_2[a]=18$ ,  $t_3[a]=12$ ,  $t_4[a]=11$ , ...
  - Bitmap

01:	1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
...	
11:	0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
12:	0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
...	
18:	0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
19:	0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
20:	0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

- Zur Basis  $\langle 2,4,3 \rangle$

1	=	001	=	0*12	+	0*3	+	1
11	=	032	=	0*12	+	3*3	+	2
12	=	100	=	1*12	+	0*3	+	0
18	=	120	=	1*12	+	2*3	+	0

0:	1,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
1:	0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
0:	1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
1:	0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
2:	0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
3:	0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
0:	0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
1:	1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
2:	0,1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

# Ergebnis

---

- Bitarrays sind gleich lang (im Unterschied zu RLE1/2)
- Aber man braucht **weniger** davon
- Das kostet bei Anfragen
  - Finden aller Tupel mit  $A=x$  benötigt Laden **mehrerer Bitarrays**
- Im Beispiel ( $\langle 2,4,3 \rangle$ ):
  - Platzverbrauch  $9*n$  statt  $20*8$
  - Suche nach allen Tupel mit  $A=15$ : Lesen von 3 Bitarrays statt einem
- Implementierung
  - Lesen aller notwendigen Bitarrays
  - Logisches AND ergibt **Bitarray mit allen Treffen**
  - Kann für komplexe Bedingungen mit anderen Bitarrays kombiniert werden

# Beispiel 1

---

- Darstellung von 20 Werten
- Bitmap
  - Speicherverbrauch (pro Tupel) 20 Bit
  - Vergleich mit Konstanter 1 Bitarray lesen
- Bitmapped zur Basis  $\langle 4, 4 \rangle$ 
  - Speicherverbrauch 8 Bit
  - Vergleich mit Konstanter 2 Bitarrays
- Bitmapped zur Basis  $\langle 2, 4, 3 \rangle$ 
  - Speicherverbrauch 9 Bit
  - Vergleich mit Konstanter 3 Bitarrays
- Bitmapped zur Basis  $\langle 2, 2, 2, 2, 2 \rangle$ 
  - Speicherverbrauch 10 Bit
  - Vergleich mit Konstanter 5 Bitarrays
- Binärdarstellung
  - Speicherverbrauch 5 Bit
  - Vergleich mit Konstanter 5 „Bitarrays“

# Beispiel 2

---

- Darstellung von 40 Werten
- Bitmap
  - Speicherverbrauch 40 Bit
  - Vergleich mit Konstanter 1 Bitarray
- Bitmapped zur Basis  $\langle 3, 4, 4 \rangle$ 
  - Speicherverbrauch 11 Bit
  - Vergleich mit Konstanter 3 Bitarrays
- Bitmapped zur Basis  $\langle 7, 7 \rangle$ 
  - Speicherverbrauch 14 Bit
  - Vergleich mit Konstanter 2 Bitarrays
- Binäre Kodierung
  - Speicherplatzoptimal 6 Bit
  - Suche eines Wertes 6 „Bitarrays“

# Fazit Bitmap-Indexe

---

- Mächtige Struktur für DWH Anwendungen
  - Attribute mit geringen Kardinalitäten
  - Häufige zusammengesetzte Bedingungen
- Komprimierung bietet zusätzliche Vorteile
  - Verschiedene Schemata möglich
  - Durch Wahl der Zahlenbasis ist (anwendungs-) optimales Austarieren von Speicherverbrauch und Zugriffsaufwand möglich
- Kommerzielle RDBMS bieten alle komprimierte Bitmap-Indexe (aber wie?)

# Inhalt dieser Vorlesung

---

- Indexierung
- Wiederholung: B- und B\*-Bäume
- Indexierung mit Bitmaps
- **Join-Indexe**

# Join-Indexe

---

- 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 die selben Joins ausgeführt: Fakten mit Dimensionstabellen
- Idee Join-Index
  - „Materialisierung“ von Joins

# Index-Join ( $\neq$ Join-Index)

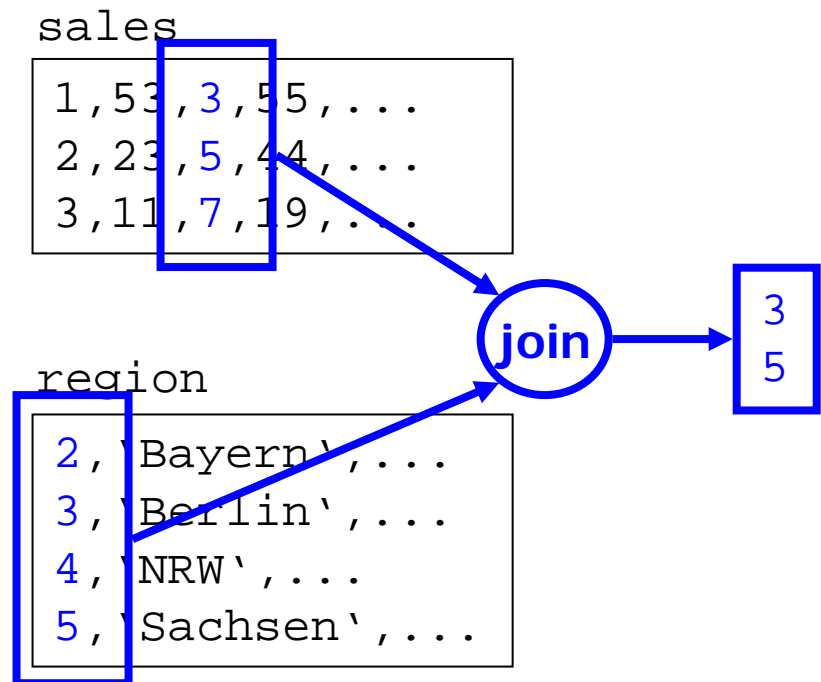
- Joins: Viele Algorithmen

- Nested Loop
- Blocked Nested Loop
- Sort-Merge
- Hash-based

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

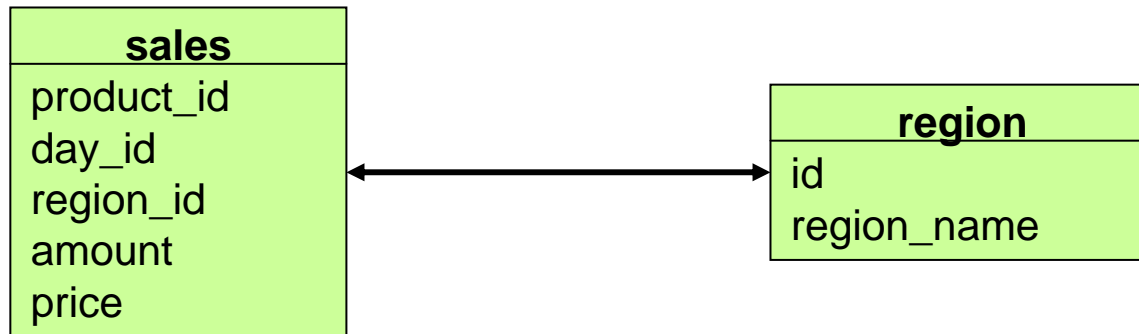
- Index-Join

- Lesen zweier sortierter TID Listen aus dem Index
- Berechnen des Schnittmenge
  - Für inner Joins
- „Nachladen“ weiterer Attribute aus beiden Tabellen nur für Treffer
- Benötigt Index auf beiden Joinattributen



# Join-Index ( $\neq$ Index-Join)

- 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)

# 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

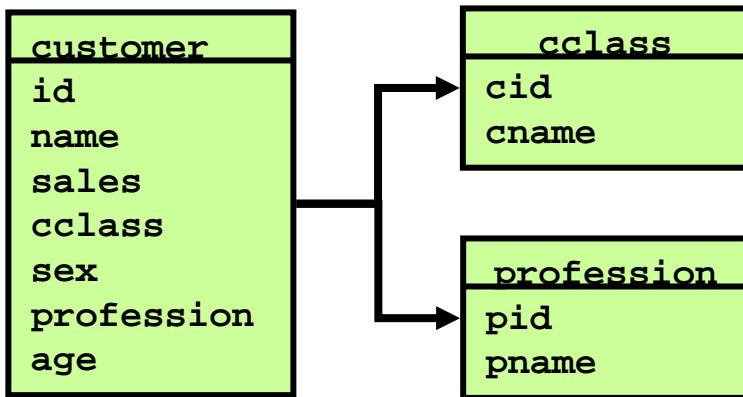


0x001: Bayern
0x002: Sachsen
0x003: Bayern
0x004: Sachsen
0x005: NRW

- Speicherung von `region.name` in Index auf `sales`
- Bei Join mit `region` und Bedingung an `region.name`
  - Table Scan von `sales` reicht
  - Kein Zugriff auf `region` notwendig
  - Kein Join notwendig

# Bitmapped Join Index

- Unsere bisherigen Beispiele für Bitmap-Indexe decken manche Probleme nicht ab
  - Bedingungen nicht an FK, sondern an „semantische“ Attribute der Dimensionstabellen gerichtet
  - Bitmaps auf FK verhindern dann nicht den Join zu Dimensionstabellen
- Lösung: **Bitmapped Join Index**



- Bitmaps erstellen für Tupel in **customer** für Werte in **cclass.cname** und in **profession.pname**

# Bitmapped 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 Bitmap-Indexen auf **sales** möglich
- Indexierung mehrerer Joins möglich

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

# Literatur

---

- Lehner: Kapitel 8.4, 8.5
- Garcia-Molina, Ullman, Widom: Kapitel 5.4
- 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)