Data Warehousing und Data Mining

Indexierung

Ulf Leser

Wissensmanagement in der Bioinformatik

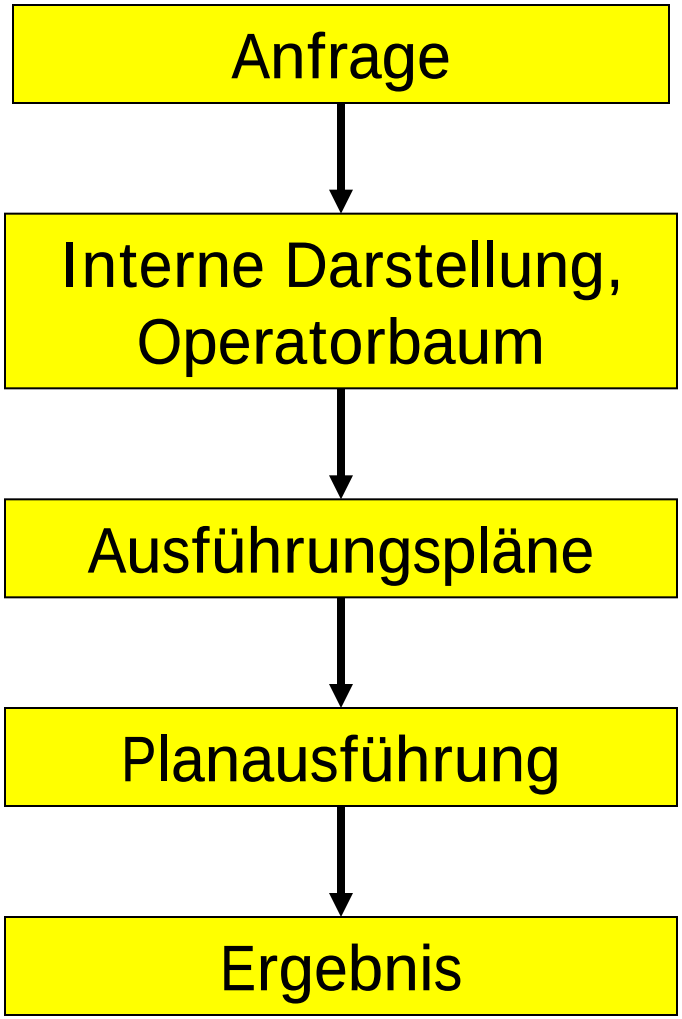
Inhalt dieser Vorlesung

- Indexierung
- Tricks mit B*-Bäumen
- Indexierung mit Bitmaps
- Join-Indexe

Wo sind wir?

- Bisher: Konzepte und Modelle eines DWH
  - Multidimensionales Datenmodell
  - OLAP Anfragen
  - ETL Prozesse
- Ab jetzt: Performanz
  - Physische Implementierung
  - Ein- und multidimensionale Indexierung
  - Spezielle Joinmethoden
  - Berechnung großer CUBEs
  - Partitionierung
  - Materialisierte Sichten
- Später: Data Mining

Anfragebearbeitung in RDBMS



Parsing; Syntaktischer und semantischer Check

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

Dynamische Programmierung; Regel- oder kostenbasierte Optimierung

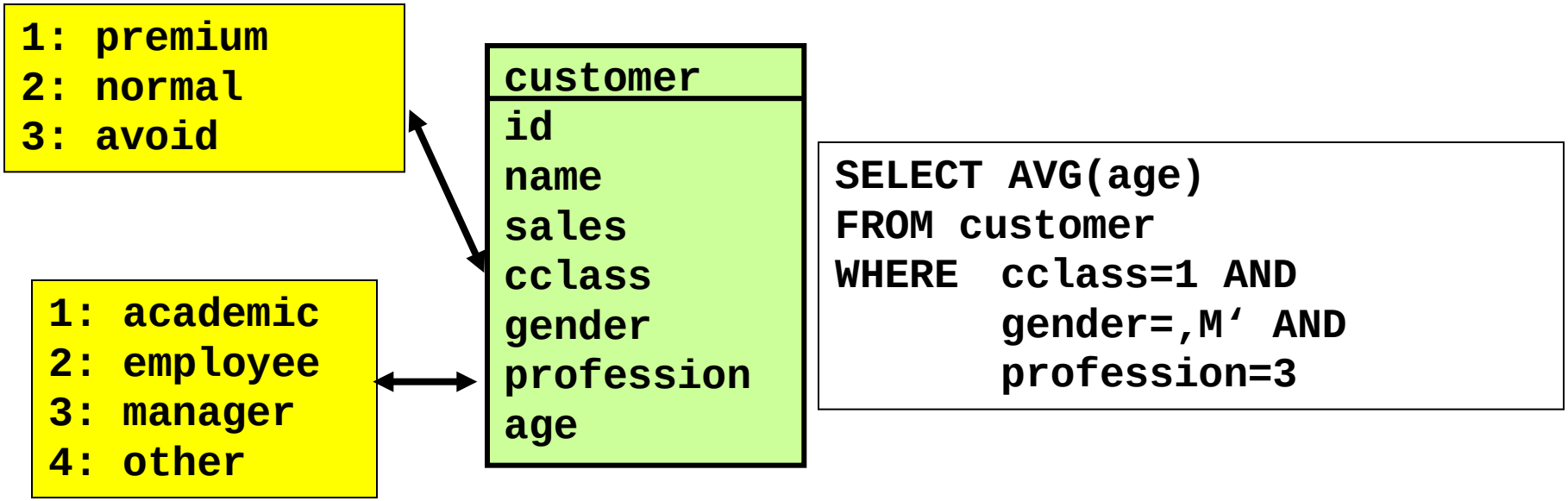
Synchronisierung, Buffering, Pipelining, Caching, ...

Anfragemuster

- Punktanfragen: Primärschlüssel, Ergebnis ist ein Tupel
  - In OLAP eher untypisch, in OLTP vorherrschendes Muster
- Bereichsanfragen (range queries): **`day>10 AND day<20`**
  - In DWH sehr häufig, oft in Kombination mit Aggregation
- Aggregation und Gruppierung
  - Meistens Zugriff auf sehr viele Fakten
  - Mehrere (abhängige, hierarchische) Aggregate in einer Anfrage
- Multidimensionale Anfragen
  - Gruppierungen / Ranges über mehrere Attribute
- Joinanfragen
- Kombinationen aus allem

Zugriff auf Faktentabelle

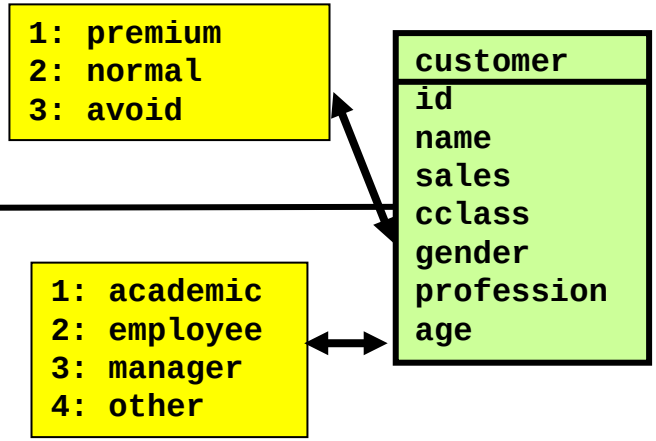
- Faktentabelle sehr, sehr groß – kritischer Engpass
- Typischer Zugriff
  - Bedingungen/Gruppierung auf Dimensionsattributen
    - Erfordert (meistens) einen Join pro Dimension
  - Aggregation von Fakten (hierarchisch, mehrdimensional)



Prinzip eines Index

- Daten liegen ungeordnet in Datenblöcken
- Index
  - Eigene, redundante und persistente Datenstruktur
  - Speichert „geordnete“ Liste aller Werte eines (oder mehrerer) Attribute zusammen mit Zeigern auf Tupel (TID)
    - Es gibt viele Arten von „Ordnung“
  - Reduktion der Zugriffszeit auf alle Tupel, bei denen der indexierte Wert einen gesuchten Wert hat
    - Punktanfrage auf Primärschlüssel: O(log(n))
    - Im B*-Baum ist der Logarithmus typischerweise zur Basis >200
- Management
  - Aktualisierung des Index mit Aktualisierung der Tabelle
  - Index muss vom Benutzer explizit angelegt werden (Wizards)

Selektivität von Anfragen



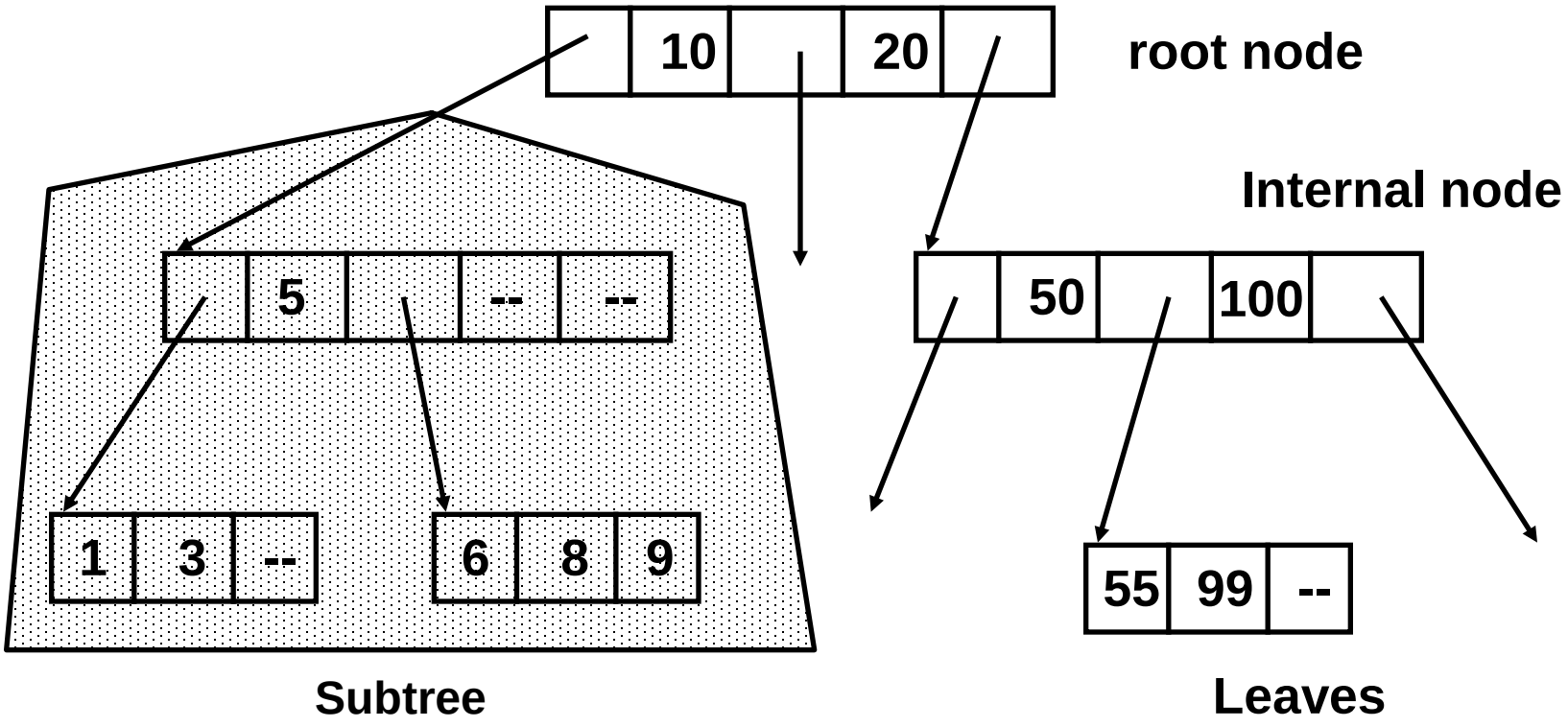
- Kein Index: Full table scan
- Mit attributweisen Indexen (bei Gleichverteilung)
  - Index auf **`gender`**: 50% Selektivität
  - Index auf **`cclass`**: 33% Selektivität
  - Index auf **`profession`**: 25% Selektivität
- Multidimensionale Punktanfrage: ~4% Selektivität
  - Annahme: Unabhängigkeit der Werte
  - Indexzugriff lohnt sich ab ~5%
- Aber: Kein Einzelindex ergibt ausreichende Selektivität
  - Optimierer wählt Full Table Scan
  - Besser: Zusammengesetzte oder Multidimensionale Indexe

Inhalt dieser Vorlesung

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

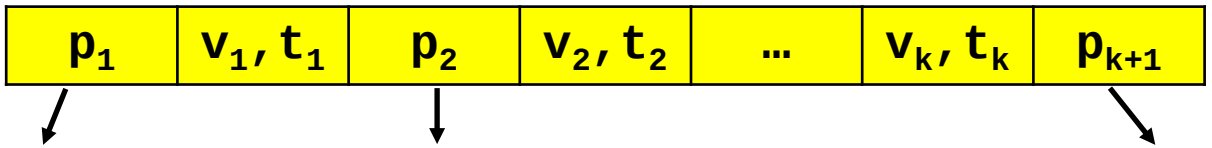
B-Trees

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



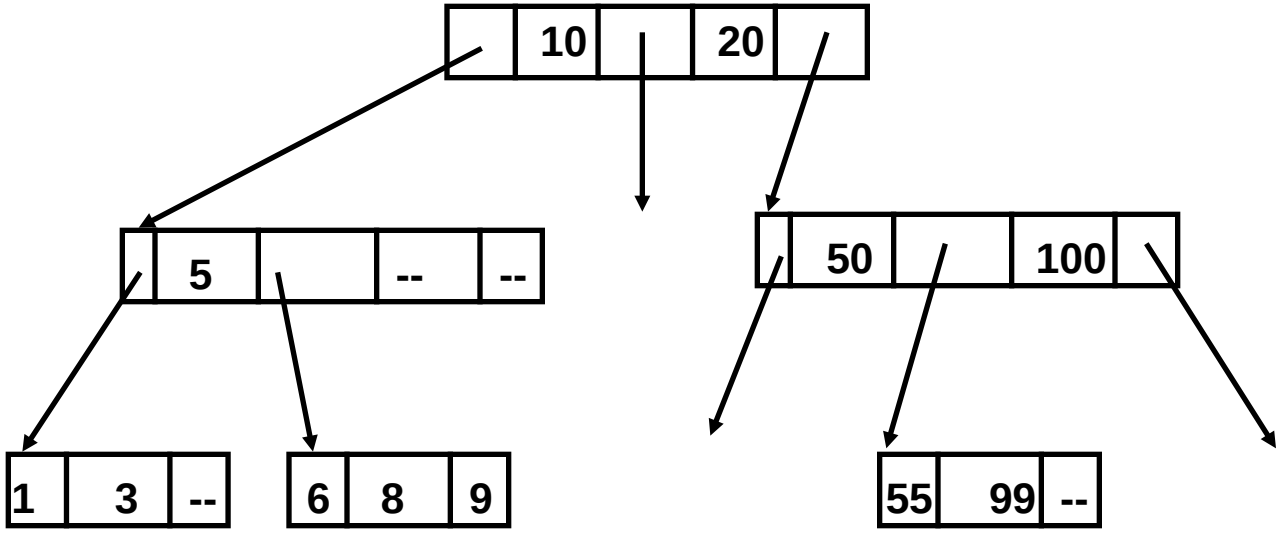
B-Trees: Formally

- Internal nodes contain pairs (val, TID) + ptrs to children
- Leaf nodes only contain pairs (val, TID)
- Assume we can fit 2k combinations of (ptr, val, TID) plus one pointer into one block on disk
  - B-Tree: Each internal node contains between k and 2k tuples (ptr, value, TID) (plus 1 ptr)
    - Exception: Root node
  - Subtree left of tuple (p, v_m, t) contains only values $v \leq v_m$
  - Subtree right of (p, v_m, t) contains only values $v > v_m$



- B-trees use always at least 50% of allocated space

Searching B-Trees



Find 8

1. Start with root node
2. Follow left-most pointer
3. Follow second-left ptr
4. Found

Find 60

1. Start with root node
2. Follow right-most pointer
3. Follow middle ptr.
4. Not found

Complexity

- B-trees are always kept balanced (depth)
- Assume n keys; let r=|value|+|TID|+|pointer|
- By definition, a node contains between k and 2k records
- Best case: Suppose all nodes are full
  - Requires b~n*r/2k blocks
    - Actually somewhat less, since leaves contain no pointers
  - The height of the tree is $h \sim \log_{2k}(b)$
  - 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*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

- |value|=20, |TID|=16, |pointer|=8, block size=4096
- Assume n=1.000.000.000 (1E9)
- We have r=44
- We may fit between k=46 and 92 index records per block
- Blocks: b = n*r/k ~ 1E9 (worst) or b ~ 5E8 (best)
- We need between $\log_{92}(5E8) \sim 4$ and $\log_{46}(1E9) \sim 6$ IO
- By caching the first two levels (between 1+46 and 1+92 blocks), this reduces to 2 to 4 IO
- Blocks on modern disks usually are much larger
- Point queries have essentially constant runtime

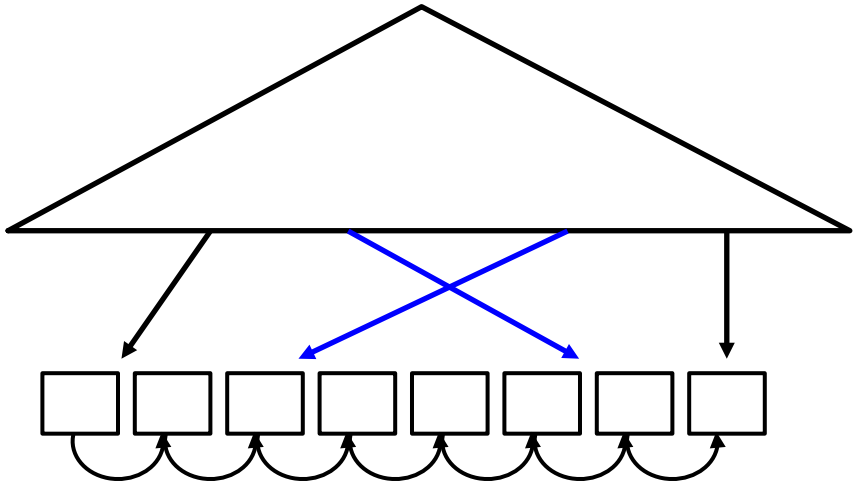
B-trees on Non-Unique Attributes

- B-trees store pairs (value, TID)
- If duplicates exist, there are two options

- Compact representation
  - Store (value, TID_1, TID_2, ... TID_n)
  - Difficult – no fixed number of pairs per block
  - Requires internal overflow blocks – degradation
- Verbose representation
  - Treat duplicates as different values
  - Generates a tree although a list would suffice (waste of space)
- Better: B* trees

B*-Trees

- Records in leaves are pairs (value, TID)
- Internal nodes store only pairs (border value, pointer)
  - No TIDs
  - Only borders between ranges of values in subtrees
  - Border values need not exist in the database - only signposts
- Plus: Leaves are connected – faster range queries
  - Note: Order of leaves on disk is difficult to guarantee



Advantages

- Simpler operations
- Disadvantage: Every search ends in a leaf, never earlier
- More records per page in internal nodes
  - No TIDs in internal nodes
  - Increased fan-out, reduced average height
  - This is the main advantage of B*trees
- Not necessarily “real” values in internal nodes
  - Can be used to save further space – Prefix Trees

Inhalt dieser Vorlesung

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

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 $O(\log_k(b))$ IO
  - Altogether: $O(n*\log_k(b))$
- 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?

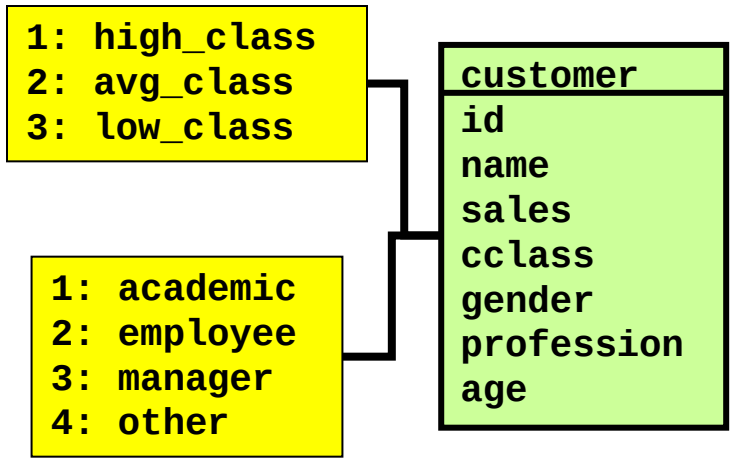
Bulk-Loading a B*-Tree

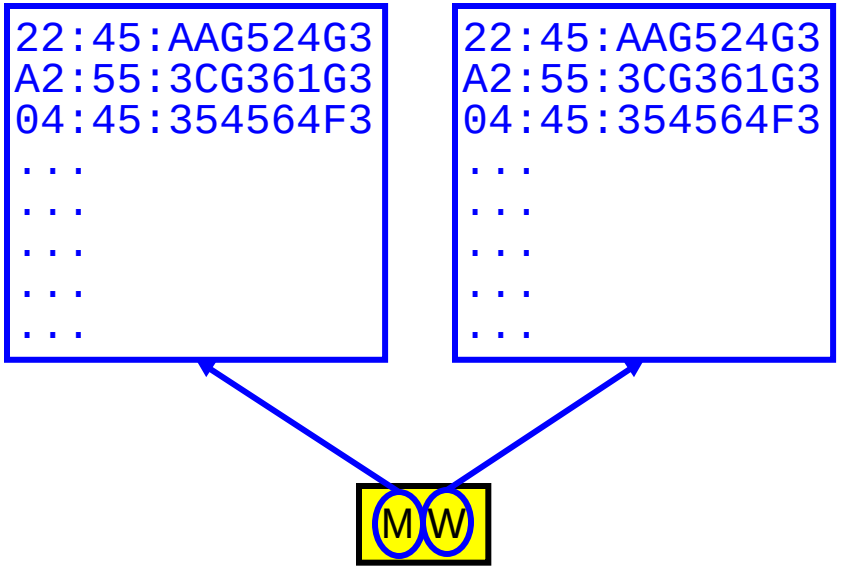
- First sort records
  - $O(n*\log_m(n))$, (m number of records fitting into memory)
  - Clearly, m (mem-buffer size) >> k (one-block size)
- Insert in sorted order using normal insertion
  - Tree builds from lower left to upper right
  - Caching will work very well
  - But space usage will be only around 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 performed in linear time

Eigenschaften von B*-Bäumen

- Robuste, generische Datenstruktur
- Unabhängig vom Datentyp
  - Attributwerte müssen vollständig geordnet sein
- Effiziente Aktualisierungsalgorithmen
- Aber …
  - Attribute mit geringer Kardinalität: Degenerierte Bäume
  - Auch zusammengesetzte Indexe sind eindimensional

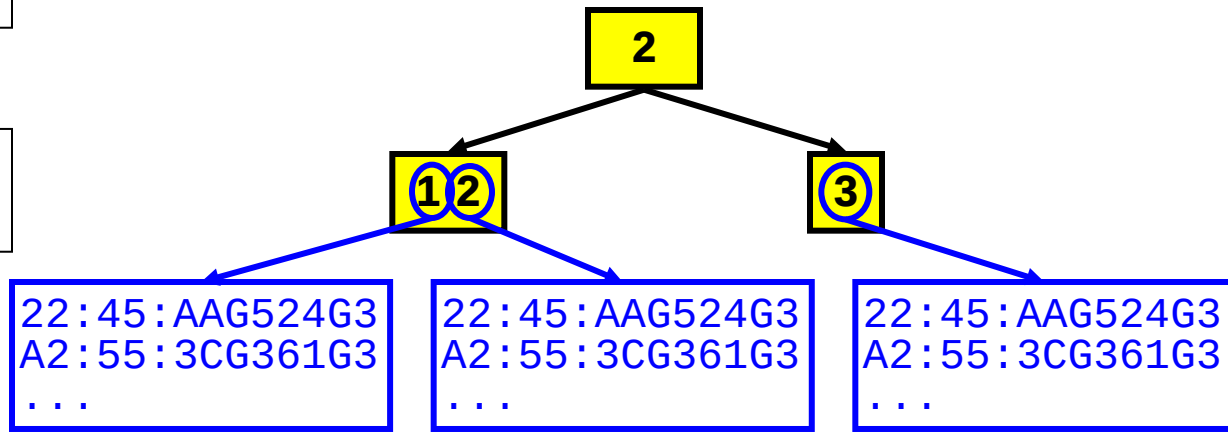
Degenerierte B*-Bäume






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

```
CREATE INDEX i_c ON
customer(cclass)
```



Falsch geordnete Multi-Attribut B*-Bäume

- Zusammengesetzter Index: Sequenz mehrerer Attribute
- Selektivität = Produkt der Selektivitäten
  - Bei Unabhängigkeit der Attributwerte
- Aber jede Anfrage muss ein Präfix der indexierten Attributsequenz enthalten

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

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

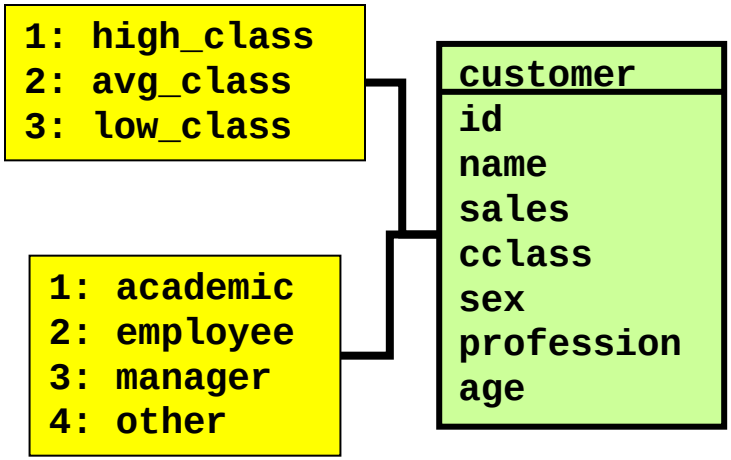
```
SELECT ...
FROM ...
WHERE   cclass=1 AND
        prof=‚other‘ AND
        gender=‚m‘
```

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

Oracle: 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

Trick: Oversized Indexe

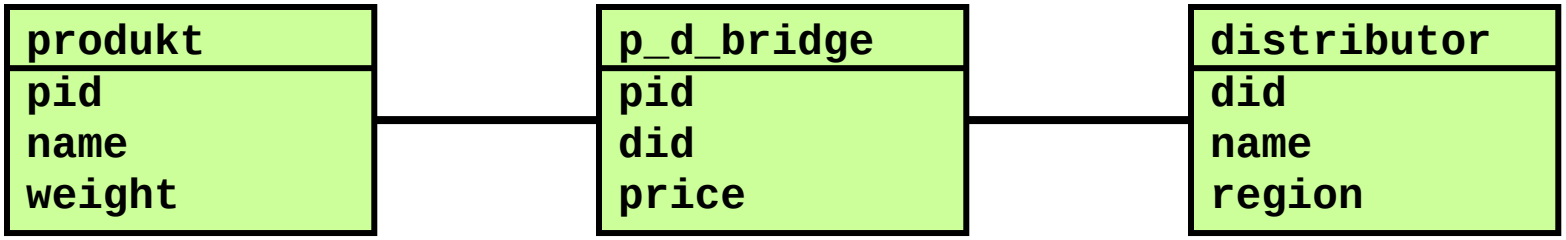



```
SELECT AVG(age)
FROM customer
WHERE   cclass=1 AND
        gender=‚M‘ AND
        profession=3
```

- 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(gender, cclass, profession, age)
```

Index-Organized Tables

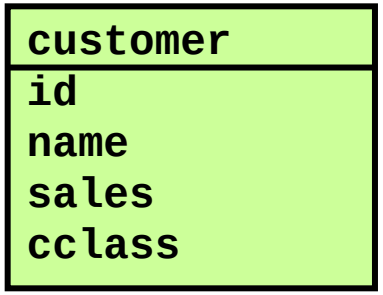



```
SELECT MIN(pd.price)
FROM produkt p, .. pd, d
WHERE   pd.pid=p.pid AND
        pd.did=d.id AND
GROUP BY pd.pid
```

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

- Index **`bridge`** ist eine (geordnete) Kopie der Tabelle
- Besser: Anlegen der Tabelle als „Index-Organized“
  - Halbierung des Speicherbedarfs
  - Schnelles sortiertes sequentielles Lesen

Berechnete Indexe




```
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
         ...
```

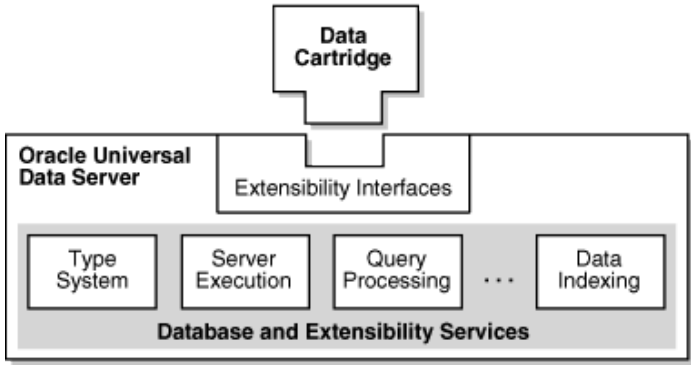
- **c_name** wird nicht benutzt. Besser:

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

- **c_name** wird vielleicht benutzt. Besser:

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

Oracle Data Cartridge Index Interface (ODCII)



- Implementierung eigener Indexstrukturen
- Create, Drop, Insert, Delete, ...
- Anfragen
  - Müssen explizit benutzt werden
  - Keine Überladung der vorhandenen Operatoren
  - Oracle-CBO (Cost Based Optimizer) API
- Hinter dem Interface
  - Zugriff auf Daten via SQL-Schnittstelle (JDBC)
  - Ergebnis sind ROWIDs
- Erfahrung: Flaschenhals Interface RDBMS / Plug-In

Inhalt dieser Vorlesung

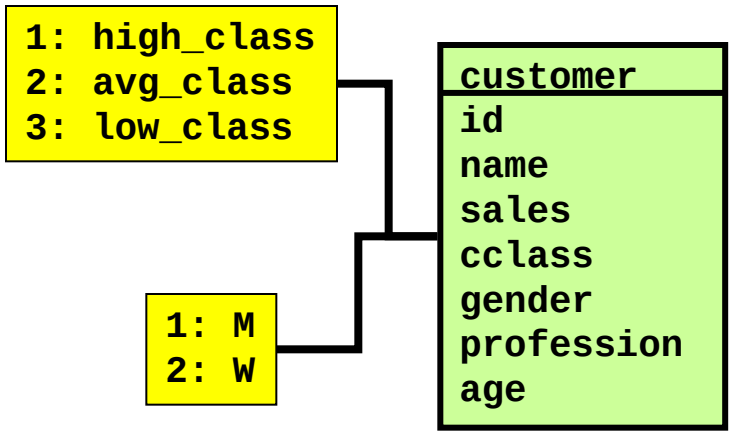
- Indexierung
- Tricks mit B*-Bäumen
- Indexierung mit Bitmaps
  - Grundidee
  - Komprimierung
  - Wechsel der Zahlenbasis
- Join-Indexe

Bitmap-Index

- Abhilfe für
  - Attribute niedriger Kardinalität (keine Entartung)
  - (Partielle) mehrdimensionale Anfragen
- Kernidee
  - Repräsentation von Attributwerten als Bitmatrizen
    - Niedrige Kardinalität – kleine Matrizen
    - Können sehr kompakt gespeichert werden
  - AND/OR Bedingungen durch Bitoperationen
    - Sehr schnell ausführbar
- State-of-the-Art
  - Für Systeme mit viel Hauptspeicher, geringen Update-Raten, und Attribute mit niedriger/mittlerer Kardinalität
  - Also: Dimensionaler Teil eines DWH

Grundaufbau

- T mit Attribut A, |T|=n, |A|=a verschiedene Werte
- Repräsentation der a Werte als je ein Bitarray der Länge n
  - Attribut X, Wert z: X.z[i]=1 gdw T[i].X=z; sonst 0
  - Die Ordnung der Tupel muss feststehen (später mehr)
- Repräsentation von A: Bitmatrix mit n*a Bits




```
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 **gender**?

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

Vorteil 1 – Speicherplatz

- Vergleich Bitmap mit B*-Baum
- Kompakte Repräsentation bei kleinem a
  - Sei n=1.000.000, zwei Attribute mit a_1=5, a_2=3, TID=4 Byte
  - Kernproblem: In jedem Index ist jede TID einmal repräsentiert
    - Platz für Speichern der konkreten Attributwert ignorieren wir
    - Wir ignorieren innere Knoten (nicht schlimm)
    - Wir unterschätzen also die echte Größe (idR nicht schlimm)
    - Wir nehmen Füllgrad 100% an (schlimm – bis zu 50% Fehler)
  - Zwei B*-Bäume: ~2*4*1.000.000 = 8MB
  - Bitmap-Index: 1.000.000 * (3+5)/8 = 1MB
- Vorteile
  - Kleiner Index passt eher in Hauptspeicher
  - Spart das Lesen von (nur logarithmisch vielen) Blöcke im B*

Vorteil 2 – Komplexe Bedingungen

- AND-verknüpfte Bedingungen an mehrere Attribute
- B*-Index
  - Lesen der TID Liste zu jeder Bedingung / Attribut
  - Schnittmengen aller Listen (z.B. durch Sortierung)
  - Auf Tupel in Schnittmenge zugreifen („explizite“ TID Liste)
- Bitmap-Index
  - Lesen der Bitarrays für jede Bedingung / Attribut
  - AND Verknüpfung
    „... cclass=3 AND gender=‚m‘“ ⇒ B[2] ∧ B[4]
  - Auf Tupel mit passenden Bits zugreifen („relative“ TID Liste)
    - Effizient, wenn das x‘te Tupel im Array an der x‘ten Stelle in der Datendatei liegt
    - Also muss die Ordnung der Tupel in Daten erhalten bleiben

Bewertung

- Kein Gewinn bei geringer Selektivität und einzelnen Bedingungen
  - Auch durch Bitmap-Indexe erhält man eine lange Liste von TIDs
  - Laden der Tupel überschattet alles, was man beim Index vielleicht gewinnt
- Großer Gewinn bei hoher Selektivität durch verknüpfte Bedingungen
  - Index wahrscheinlich im Hauptspeicher
  - Bitoperationen sehr schnell (viel schneller als Schnittmengen)
  - Keine langen TID Listen als Zwischenergebnis
- Außerdem: Ordnungen der Attribute im Index/Anfrage sind egal

Nachteile

- Benötigen feste Ordnung der Tupel
  - Keine Bitmap Indexe auf index-Organized Tables
- Nicht geeignet bei Range-Queries mit grossen Ranges
  - Sehr viel implizites OR, idR geringe Selektivität
- 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

Management von Bitmap-Indexen

- Feste Ordnung ist meist gegeben
  - Löschen eines Tuples: Platz wird freigegeben (Grabstein)
    - Ordnung der Tupel ändert sich nicht
    - In allen Bitarrays alle Werte auf 0 setzen
  - Einfügen eines Tuples: Ersetzen Grabstein oder append
    - Letzteres verlängert alle Bitarrays um 1
    - (Bei Komprimierung unter Umständen implizit zu erledigen)
  - Update kann zu Tupelverschiebung führen
    - Behandeln als Delete + Insert
- Schwierig sind Reorganisationen („Vacuum“)
- Spezialfall: Tupel mit neuem Wert für A
  - Anlegen eines neuen Bitarrays mit einer 1, sonst 0

Inhalt dieser Vorlesung

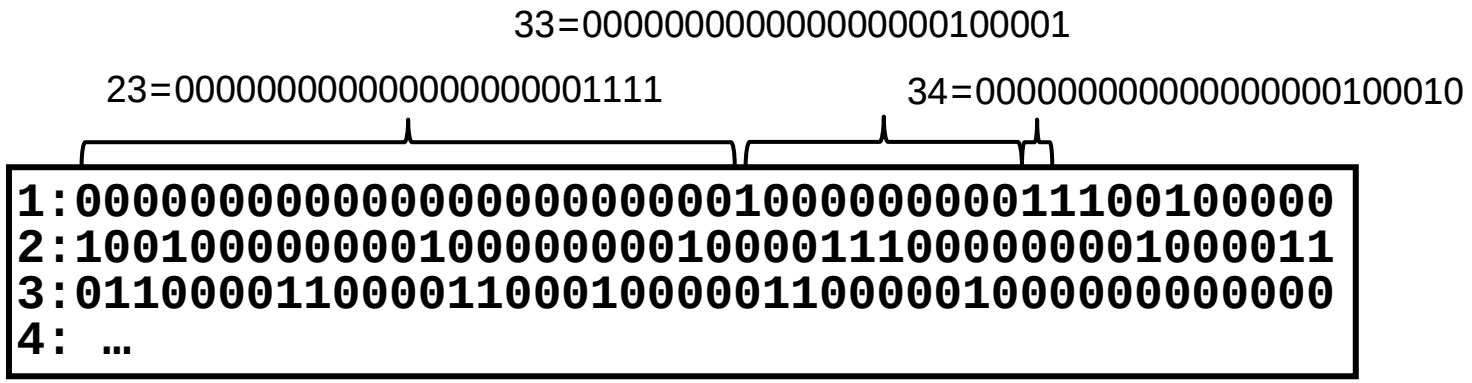
- Indexierung
- Tricks mit B*-Bäumen
- Indexierung mit Bitmaps
  - Grundidee
  - Komprimierung
  - Wechsel der Zahlenbasis
- Join-Indexe

Komprimierte Bitmap-Indexe

- Ziel: Platzreduktion bei Attributen hoher Kardinalität
  - Egal wie hoch die Kardinalität: Nur ein Bit pro Tupel ist gesetzt
  - D.h., dass die Bitmatrix im wesentlichen aus 0‘ern besteht
    - n*a bits, nur n davon sind 1
  - Hohe Kompressionsraten möglich
- Müssen für Anfragebearbeitung dekomprimiert werden
  - Zur Ladezeit
    - Vorteil: Schnelle Queries, wenig IO (da kleine Files)
    - Nachteil: Hoher Speicherbedarf im Hauptspeicher
  - Standard: Zur Anfragezeit (bleiben komprimiert im Speicher)
    - Vorteil: Geringer Speicherverbrauch
    - Nachteil: Performanceverlust bei Queries
    - Trotzdem idR schnell, weil kein IO notwendig

Run-Length-Encoding (RLE)

- Variante 1 (RLE1): Explizites Speichern der 1-Positionen
  - Beispiel: n=1.000.000, a=100, TID: 4 Byte
  - Pro Bitarray ist nur etwa einer von 100 Werten eine „1“
  - 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) = 2.5MB
- Beispiel



Run-Length-Encoding (RLE)

- Platzbedarf: n*log(n) bit
- Unkomprimiertes Array: n*a bit
- Lohnt sich also nur, wenn a>log(n)
- Problem
  - Wir müssen die Größe des Arrays von vorneherein festlegen
    - Um die Anzahl der Bits für Kodierung festlegen zu können
  - Maximal notwendige Zahl von Bits bestimmt also alle Positionen
  - Aber: Die meisten Positionen brauchen weniger Bits
  - Können wir dieselbe Information noch kompakter speichern?

Variante 2: RLE2

- Weg von festen Adressgrößen
  - Dann müssen Grenzen von Adressen erkennbar sein
  - Lohnt sich nur, wenn die meisten Adressen wenige Bits brauchen
- Delat-Encoding: Speichere Länge der 0-Blöcke statt Adressen der 1‘er
  - Wenn 1‘er halbwegs gleichverteilt sind, sind die 0-Blöcke alle halbwegs gleich lang – und die Längen viel kürzer als die Adressen der weit rechts stehenden 1‘er

Variante 2: RLE2

- Beispiel
  - Bitstring: 000011010000101010001000000100010000
  - Blöcke 4,0,1,4,1,1,3,7,3
  - Jeder Block repräsentiert i-mal Null und eine Eins
  - Schließende Nullen nur implizit repräsentiert
    - Größe des Bitarray muss bekannt sein
- Blöcke will man mit der minimalen Anzahl Bits speichern
- Problem: fehlende Eindeutigkeit
  - 000101: Blöcke 3,1; Codierung 111
  - 010001: Blöcke 1,3; Codierung 111
  - 010101: Blöcke 1,1,1; Codierung 111
- Maximaler Abstand: Immer noch n, also erst Mal nichts gewonnen

Kodierung

- Wir merken uns die Länge des Bitstrings für jeden Block
  - Blocklänge benötigt x Bits – (x-1) Einser, dann eine Null
- Damit
  - 000101 ergibt Blöcke 3,1 ergibt 101101
  - 010001 ergibt Blöcke 1,3 ergibt 011011
  - 010101 ergibt Blöcke 1,1,1 ergibt 010101
  - Das sieht nicht nach Komprimierung aus!
- 1-er müssen selten sein
  - 1000000100001000
    - Blöcke 0,7,4 ergibt 00110111110100
    - Blocklänge 0 wird als 00 kodiert
  - 0000000001000010
    - Blöcke 10,4 ergibt 11101010110100
- Noch seltener – 1 aus 100
  - 0000000010000000000000000000000000000000000000100
    Blöcke 8,41 ergibt 11101000111110101001

Speicherverbrauch RLE2

- Wie viel Speicher brauchen wir nun?
  - Beispiel: n=1.000.000, a=100, TID: 4 Byte
  - Im Bitarray ist jede 100‘te Position eine 1
  - Die durchschnittliche Blocklänge ist 99 (7 Bit)
  - Es gibt ca. 10.000 Blöcke
  - Pro Block ungefähr (7-1)+1+7 Bit
  - Das ganze Array: 100*(10.000*14)/8 = 1.75 MB
    - Erinnerung: B*-Baum: 4MB, unkomprimiertes Bitarray: 12,5MB
- Sehr grobe Schätzung
  - Blocklängen sind nicht alle gleich
  - Lange Blöcke brauchen nur logarithmisch mehr Platz und führen zu weniger Blöcken (Gleichverteilung ist Worst-Case)
  - Beispiel: 9000 Blöcke Länge 4, 1000 Blöcke Länge ~1000
    - 100*((9000*6)/8 + (1000*20)/8) = 0,88 MB

Extreme Scale

- Beispiel: n=10.000.000.000, a_1=...a_5=100, TID: 4 Byte
- Je BitArray: Blocklänge 99 (7 Bit), ca. 1E8 Blöcke
- Pro Block ungefähr (7-1)+1+7 Bit
- Das ganze Array: 100*5*(1E8*14)/8 ~ 1E6 MB = 1TB

- Man braucht einen grossen Server
- Aber dann prozessiert man sehr viele Queries ohne IO und trotzdem mit Indexunterstützung

Komprimierung und Sperren

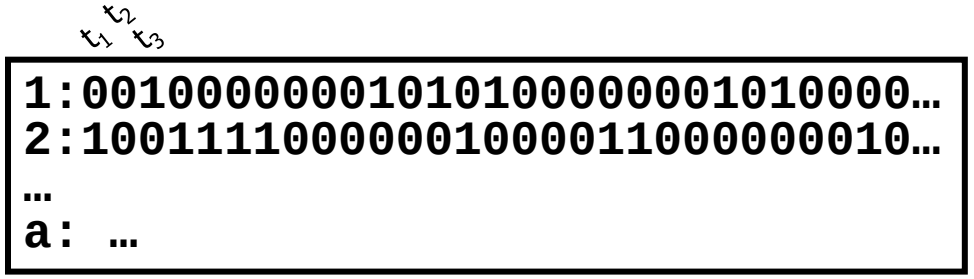
- Vorsicht bei transaktionalen Updates auf komprimierten Bitmaps
- Führen zum Sperren sehr vieler Tupel
  - Sehr viele Bits in einem Diskblock
  - Sperrung des Blocks kann zig-tausend Tupel sperren
  - Eher für Read-Only Umgebungen geeignet

Inhalt dieser Vorlesung

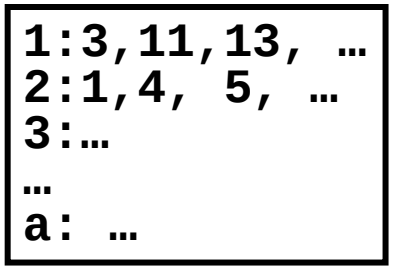
- Indexierung
- Tricks mit B*-Bäumen
- Indexierung mit Bitmaps
  - Grundidee
  - Komprimierung
  - Wechsel der Zahlenbasis
- Join-Indexe

Vertikale Komprimierung

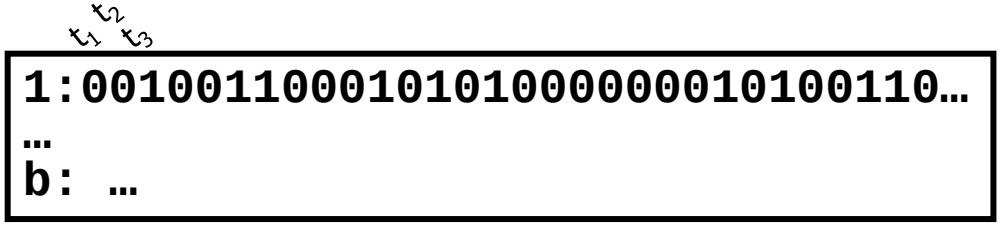
- Attribut A mit |A|=a verschiedenen Werten



- Komprimierung RLE1



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



Beispiel

- Beispiel: a=20, t_1[a]=1, t_2[a]=18, t_3[a]=12, t_4[a]=11, ...
  - Bitmap
    - Wir brauchen 20 Bitarrays
    - Alle Tupel mit einem Wert: Ein Bitarray lesen
  - Binärcodierung
    - Wir brauchen log(20)=5 Bitarrays
    - Alle Tupel mit einem Wert: Alle fünf Bitarrays lesen
  - Codierung zur Basis <5,5>
    - Idee: Bitmap Darstellung pro Ziffer
    - Wir brauchen 5+5=10 Bitarrays
    - Alle Tupel mit einem Wert: Zwei Bitarrays lesen

```
01:1,0,0,0,…
…
11:0,0,0,1,…
12:0,0,1,0,…
…
20:0,0,0,0,…
```

```
0:1,0,0,1,…
1:0,1,0,1,…
2:0,0,1,0,…
3:0,0,1,1,…
4:0,1,0,0,…
```

```
0:0,0,0,0,…
1:1,0,0,1,…
2:0,0,1,0,…
3:0,1,0,0,…
4:0,0,0,0,…
0:0,0,0,0,…
1:0,0,0,0,…
2:0,0,1,1,…
3:0,1,0,0,…
4:0,0,0,0,…
```

Zahlenbasis

- Wir stellen eine Zahlenbasis dar als $<A_1,A_2,A_3,...>$
  - Erste Stelle mit A_1 verschiedenen Werten, zweite Stelle mit A_2 verschiedenen Werten ...
  - Typisch, aber nicht notwendig: $A_1=A_2=...$
- Sei $<x_1,x_2,x_3>$ eine Zahl zur Basis <a,b,c>. Dann ergibt sich die Dezimalzahl: $<x_1,x_2,x_3> = x_1*(b*c) + x_2*c + x_3$
  - Beispiel: <2,4,1> zur Basis <4,4,2>: 2*4*2+4*2+1=25
- Wie stellt man die einzelnen Ziffern dar?
- Wir verwenden dazu Bitarrays per Wert pro Ziffer
  - Also: Zahlen zur Basis <4,4,2> brauchen 10 Bit
  - Prüfen einer Ziffer auf konkreten Wert benötigt nur 1 Bitarray

Darstellung von Zahlen

Darstellung	Werte-bereich	Speicherbedarf Bitmaps	Beispiel: 37
<10,10,10>	10^3	30 Bit	<0,3,7>=0000000000,...
<5,5,5>	125	15 Bit	<1,2,2>=00010,00100,00100
<2,5,4>	40	11 Bit	<1,4,1>=10,10000,0010
<a,b,c>	a*b*c	a+b+c Bit	
<2,2,2,2,2,2>	2^6	12 Bit	<1,0,0,1,1,1>=10,00,00,10,10,10

- Die <2,2,2,2,2,2> speichert man natürlich besser in 6 Bit
- Das geht nur bei 2 als Zahlenbasis, da 1 Bit zwei Werten ausdrücken kann

Noch ein Beispiel

- 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,…
…
11:0,0,0,1,…
12:0,0,1,0,…
…
18:0,1,0,0,…
19:0,0,0,0,…
20:0,0,0,0,…
```

  - Zur Basis <2,4,3>

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

```
0:1,0,0,1,…
1:0,1,1,0,…
0:1,0,1,0,…
1:0,0,0,0,…
2:0,1,0,0,…
3:0,0,0,1,…
0:0,1,1,0,…
1:1,0,0,0,…
2:0,0,0,1, …
```

Ergebnis

- Alle Bitarrays sind n bits lang (im Unterschied zu RLE)
  - Schnellere Dekodierung, einfachere Updates
- Braucht weniger Bitarrays als RLE1/2
  - Im Beispiel (<2,4,3>): Platzverbrauch 9*n statt 20*n Bits
- Das kostet bei Anfragen
  - Finden aller Tupel mit A=x benötigt Laden mehrerer Bitarrays
  - Beispiel: Alle Tupel mit A=15 zur Basis <2,4,3>: 3 Bitarrays
- Implementierung
  - Lesen aller notwendigen Bitarrays
  - Logisches AND ergibt Bitarray mit allen Treffen
  - Kann für komplexe Bedingungen mit anderen Bitarrays kombiniert werden

Beispiel 1: 20 Werte

	Speicherverbrauch	Punktabfrage
Bitmap	20 Bit	1 Bitarray
Bitmapped zur Basis <4,5>	9 Bit	2 Bitarrays
Bitmapped zur Basis <2,4,3>	9 Bit	3 Bitarrays
Binärdarstellung	5 Bit	5 Bitarrays

Beispiel 1: 40 Werte

	Speicherverbrauch	Punktabfrage
Bitmap	40 Bit	1 Bitarray
Bitmapped zur Basis <3,4,4>		
Bitmapped zur Basis <7,7>		
Binärdarstellung		

Beispiel 1: 40 Werte

	Speicherverbrauch	Punktabfrage
Bitmap	40 Bit	1 Bitarray
Bitmapped zur Basis <3,4,4>	11 Bit	3 Bitarrays
Bitmapped zur Basis <7,7>	14 Bit	2 Bitarrays
Binärdarstellung	6 Bit	6 Bitarrays

Fazit Bitmap-Indexe

- Geeignete Indexstruktur für DWH Anwendungen
  - Attribute mit geringen Kardinalitäten
  - Häufige zusammengesetzte Bedingungen
- Komprimierung bietet zusätzliche Vorteile
  - Verschiedene Schemata möglich
  - Viele Queryteile im Hauptspeicher behandelbar
  - Wichtig: Austarieren von Speicherbedarf und Zugriffsaufwand
- Kommerzielle RDBMS bieten heute alle komprimierte Bitmap-Indexe
  - PostGresQL: Bitmap Index mit Variante von RLE1
  - MySQL: Keine Bitmap Indexe
  - Kommerzielle Systems: Alle „komprimiert“, aber wie?

Inhalt dieser Vorlesung

- Indexierung
- Tricks mit Bäumen
- Indexierung mit Bitmaps
- Join-Indexe

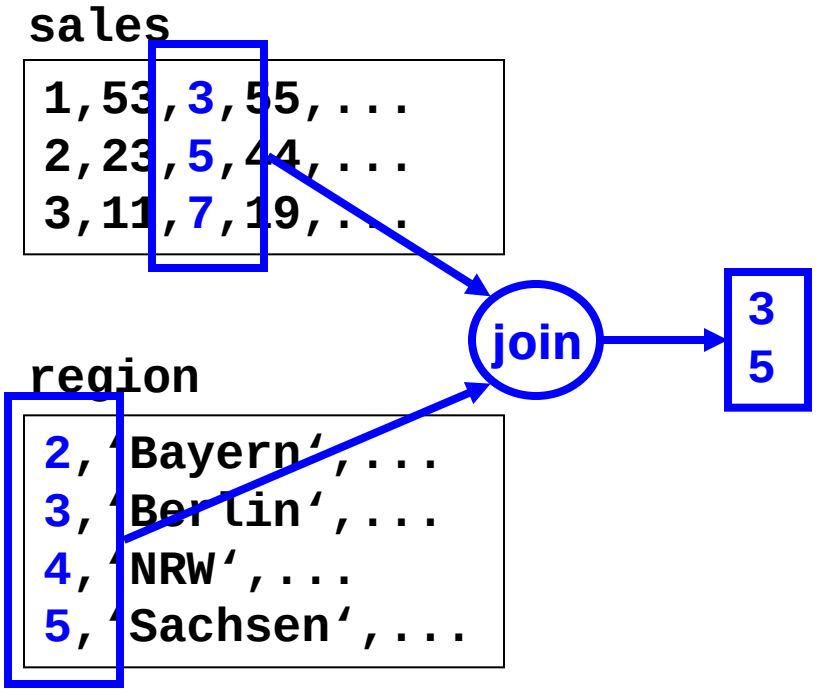
Join-Indexe

- Joins sind teure Operationen
  - Bewegung vieler Daten
  - Viele Alternativen für Optimierer
  - U.U. große Zwischenergebnisse trotz kleiner Endergebnisse
- Beobachtung bei DWH
  - Es werden immer dieselben Joins ausgeführt
    - Faktentabelle mit Dimensionstabellen
    - Dimensionstabellen mit Dimensionstabellen im Snowflake-Schema
- Idee Join-Index: „Materialisierung“ eines Joins

Index-Join (≠ Join-Index)

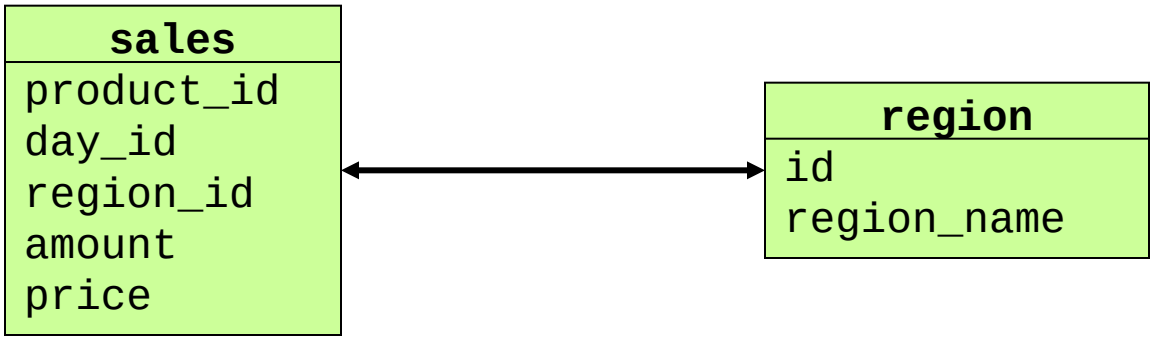
- Joins: Viele Algorithmen
  - Nested Loop, Sort-Merge, Hash-based, ...
- Index-Join
  - Lesen zweier sortierter TID Listen aus dem Index
  - Berechnen der Schnittmenge
  - „Nachladen“ weiterer Attribute aus beiden Tabellen nur für Treffer
  - Benötigt Index auf beiden Joinattributen

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



Join-Index (≠ Index-Join)

- Indexierung von Spalten einer Tabelle A mit Werten einer (häufig gejointen) Tabelle B




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

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

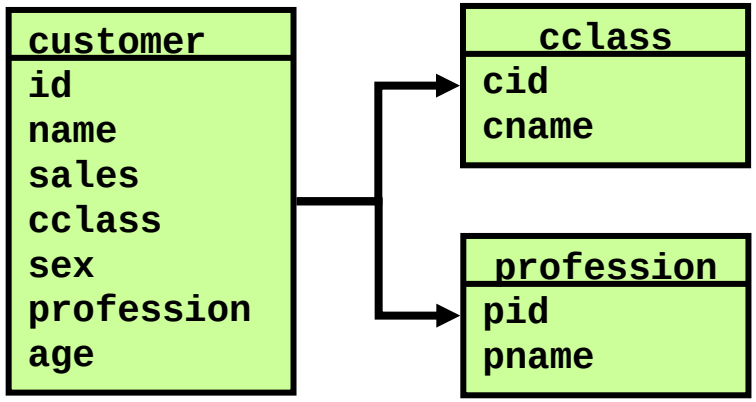
Join Index

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

- Speicherung von **`region.name`** in Index auf **`sales`**
  - Join mit **`region`** und Bedingung an **`region.name`** kann aus dem Index beantwortet werden
- Besonders lohnend, wenn ein kombinierter Index verwendet wird (mit mehreren gejointen Werten)

Bitmapped Join Index

- Bisherige Bitmap-Indexe decken nicht alle Probleme ab
  - Bedingungen nicht an FK, sondern an Attribute der Dimensionstabellen gerichtet
  - Bitmaps auf FK verhindern dann nicht den Join zu Dimensionstabellen
- Lösung: Bitmapped Join Index



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)

Selbsttest

- Welche Methoden gibt es, einen B* Index für viele Werte auf einmal zu bauen?
- Was ist die garantierte Höhe eines B* Index?
- Unterschied B und B* Index?
- Wie funktioniert Bitmap-Komprimierung?
- Stellen Sie XYZ in RLE2 dar; dekodieren Sie XYZ
- Was ist ein Join-Index?
- Was muss für die Kardinalität einer Beziehung gelten, damit man Join-Indexe anlegen kann?