Hasso Plattner Institut
IT Systems Engineering | Universität Potsdam

Datenbanksysteme II

Physische Repräsentation von Daten

(Kapitel 12)

25.4.2007

Felix Naumann

5-Schichten Architektur

Aufbau

Motivation: Beziehung zwischen Block-Modell der Festplatte und Tupelmodell des DBMS

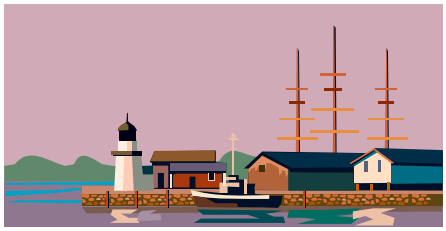
- Attributwerte -> Bytelisten fester oder variabler Länge: „Felder“
- Tupel -> Feldlisten fester oder variabler Länge: „Datensätze“
- Physische Blöcke speichern Datensatzmengen
- Relation -> Mengen von Blöcken: „Datei“
  - Plus Indexstrukturen

Felder

- Kleinste Dateneinheit: Attributwerte
- Repräsentiert durch „Felder“ (*fields*)
- ```
  CREATE TABLE Schauspieler (
      Name CHAR(30),
      Adresse VARCHAR(255),
      Geschlecht CHAR(1),
      Geburtstag DATE );
  ```
- Wie werden Datentypen als Felder repräsentiert?
- Wie werden Tupel als Datensätze repräsentiert?
- Wie werden Mengen von Datensätzen oder Tupeln in Blöcken repräsentiert?
- Wie werden Relationen als Mengen von Blöcken repräsentiert?
- Was passiert bei variablen Feld- oder Tupellängen?
- Was passiert wenn ich einen Block nicht einheitlich in Tupel einteilen kann?
- Was passiert wenn sich die Größe eines Datensatzes ändert, insbesondere vergrößert?
  ```

Überblick

- Datenelemente
- Datensätze
- Adressierung
- Daten variabler Länge
- Datensatzänderungen



Datentypen

- Irgendwann werden alle Daten als Bitsequenzen dargestellt.
- Irgendwann werden alle Daten als Bytesequenzen dargestellt.
  - Integer: 2 oder 4 Byte
  - Float: 4 oder 8 Byte
- Strings fester Länge: CHAR(*n*)
  - Feld hat *n* Byte
  - Fehlende Byte-Werte werden mit ⊥ gepadded.
  - ‚Katze' in CHAR(8) wäre also K a t z e ⊥ ⊥ ⊥

Strings variabler Länge

7

- VARCHAR(*n*)
- Es werden n+1 Byte reserviert.
- Variante 1
  - Byte 1 speichert Länge
    - => Länge ist maximal 255 Byte
  - Weitere Bytes speichern Attributwert
  - Fehlende Bytes werden ignoriert
  - 5 K a t z e
    - Eigentlich 00000101 K a t z e
- Variante 2
  - Erste Bytes speichern Attributwert
  - Hintendran ein null-Wert (⊥)
  - K a t z e ⊥

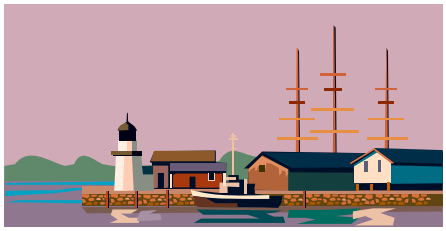
Datum / Bit / Boolean

8

- DATE, TIME
  - i.d.R. repräsentiert als String fester Länge
  - Problem: Zeit kann mit Bruchteilen von Sekunden gespeichert werden (theoretisch beliebig genau)
    - Lösung 1: Speicherung als Sting fester Länge mit maximaler Genauigkeit
    - Lösung 2: Speicherung als String variabler Länge
- BIT(*n*)
  - 8 Bit pro Byte
  - Letzte Bits ignorieren, falls *n* nicht durch 8 teilbar
  - BIT(12): 010111110011 wird zu 01011111 und 0011<u>0000</u>
- BOOLEAN
  - 8 Bit:
    - 00000001 und 00000000
    - 11111111 und 00000000

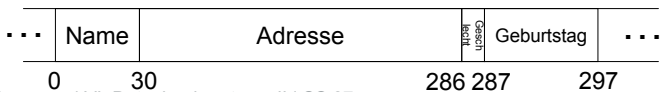
Überblick

- Datenelemente
- Datensätze
- Adressierung
- Daten variabler Länge
- Datensatzänderungen



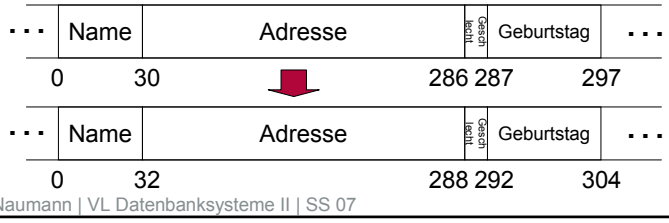
Datensätze fester Länge

- Jeder Datensatz hat ein Schema.
  - Namen und Datentypen der Felder
  - Offset im Datensatz
  - Anmerkung: JEDER Datensatz!
    - Realisiert als Pointer auf Schema
- **CREATE TABLE Schauspieler (**
  Name CHAR(30),
  Adresse VARCHAR(255),
  Geschlecht CHAR(1),
  Geburtstag DATE);
- Alle haben fest Länge
  - 30 Byte + 256 Byte + 1 Byte + 10 Byte = 297 Byte



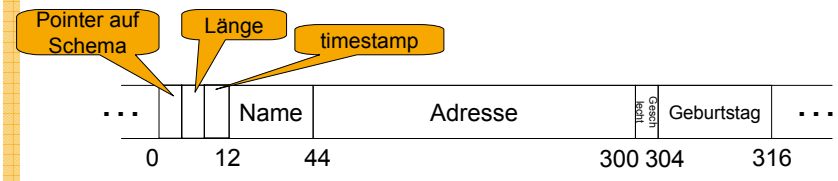
Versatz zur Effizienz

- Felder beginnen am besten bei Hauptspeicheradressen, die ein Vielfaches von 4 (bzw. 8) sind.
  - Manchmal sogar Pflicht
- Für Festplatte eigentlich egal
  - Aber eingelesener Datensatz landet auf einem Speicherplatz mit entsprechender Adresse
    - Vielfaches von 4
    - Vielfaches von 2^n
  - Entsprechend versetzt sind die anderen Felder



Speicherung der Metadaten

- Metadaten eines Datensatzes
  - Schema bzw. Pointer auf ein Schema
  - Länge des Datensatzes
  - Timestamp der letzten Änderung bzw. des letzten Lesens
  - ...
- Lösung: Header vor den Datensatz



Aufteilung in Blöcke

13

- Block header (optional)
  - Links auf andere Blocks (z.B. Index)
  - Rolle dieses Block (im Index)
  - Relation der Tupel
  - Verzeichnis der offsets der Datensätze
  - Block ID (gleich)
  - Timestamp der letzten Änderung / des letzten Lesens
- Einfachster Fall: Alle Datensätze aus gleicher Relation, aller fester Länge

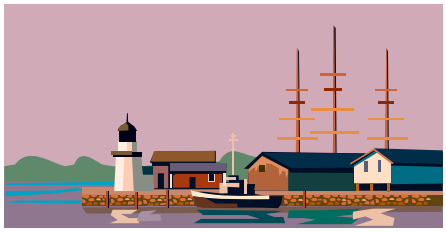
Header	Datensatz 1	Datensatz 2	…	Datensatz *n*	

- Beispiel
  - Datensatz 316 Byte
  - Block 4960 Byte, 12 davon als Header
  - => 12 Datensätze und 292 verlorene Byte

Überblick

14

- Datenelemente
- Datensätze
- Adressierung
- Daten variabler Länge
- Datensatzänderungen



Virtueller Speicher vs. Festplatte

15

- Block im Hauptspeicher
  - Block-Adresse ist im virtuellen Adressraum
  - Zeigt auf erstes Byte des Blocks
  - Datensatzadresse zeigt auf erstes Byte des Datensatzes
- Block auf Festplatte
  - Adresse nicht im virtuellen Adressraum
  - Speicherort im gesamten System des DBMS
    - Disk ID, Zylinder#, Oberfläche…
  - Datensatz: Block + offset des ersten Bytes

Adressraum des Servers

16

- Blocks und Offsets innerhalb von Blocks
- Variante 1: Physische Adressen
  - Rechner ID
  - Disk ID
  - Zylinder#
  - Spur# (bei mehr als einer Oberfläche)
  - Block#
  - (Offset innerhalb des Blocks)

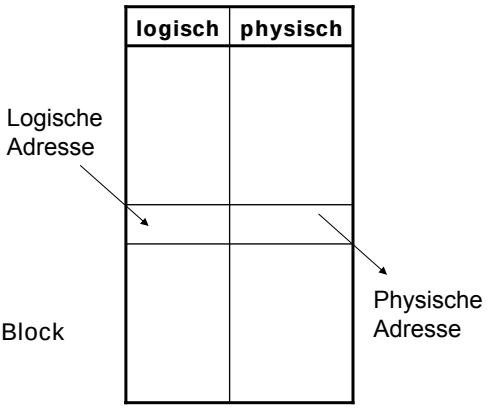
  } 8-16 Byte
- Variante 2: Logische Adressen
  - Beliebiger Byte String
  - *Mapping table* übersetzt diese Adresse in eine physische Adresse.

Logische Adressen

17

Warum die Indirektion?

- Flexibilität bei der Umorganisation von Daten
  - Änderungen nur auf der mapping table



Hybride Adressierung

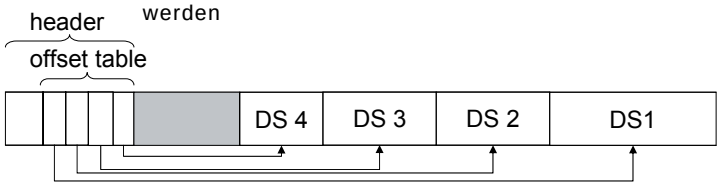
- Physische Adresse für einen Block
- Logische Adresse für einen Datensatz in dem Block
  - Z.B. Schlüsselwert

Hybride Adressierung

18

Idee: Physische Adressen zu einem Block. Block speichert *offset table*.

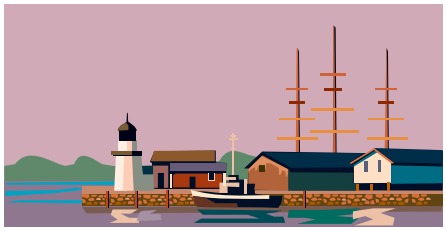
- Auffüllen des Blocks von hinten bei Datensätzen variabler Länge
  - Anzahl der Datensätze nicht fix
    => Größe des headers kann offen gelassen werden
- Vorteile der Flexibilität auch ohne Mapping table
  - Innerhalb eines Blocks kann umorganisiert werden
  - Datensatz kann sogar Blöcke wechseln
    - Speicherung der neuen Adresse in der offset table
    - Bei Löschung kann ein Grabstein hinterlassen werden
      - » Alte Pointer auf das gelöscht Objekt können anmgepasst werden



Überblick

19

- Datenelemente
- Datensätze
- Adressierung
- Daten variabler Länge
- Datensatzänderungen



Variable Länge

20

Bisher: Feste Länge.

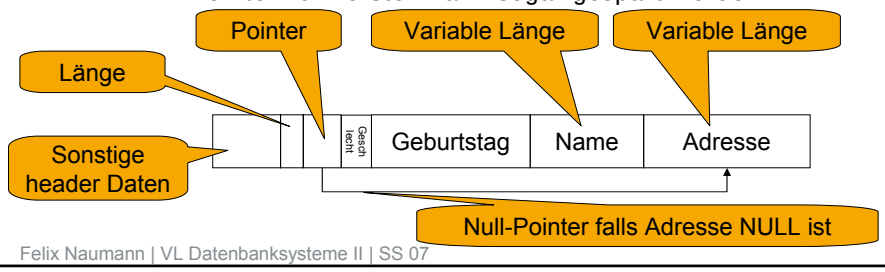
Aber:

- Felder variabler Länge
  - **`Adresse VARCHAR(255)`** wird selten voll ausgeschöpft
- Datensätze variabler Länge
  - Ergänzung von Datensätze um Felder
  - Schauspieler, die auch Regie führen
- Riesige Felder
  - GIF, MPEG
  - Passen nicht mehr auf einen Block

Finden von Feldern

21

- Datensatz muss Informationen speichern, um jedes Feld aufzufinden.
- Idee: Felder fester Länge an den Anfang des Datensatzes
- Header speichert
  - Länge des Datensatzes
  - Pointer (offsets) zu den Anfängen aller Felder variabler Länge
    - Pointer zum ersten kann sogar gespart werden.

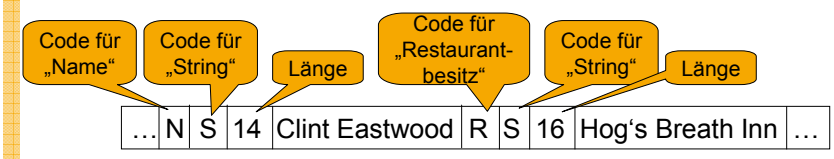


Datensätze variabler Länge

22

Es ist unbekannt welche und wieviele Felder der Datensatz haben wird.

- *Tagged fields* (getaggte Felder)
  - Feldname (Attributname)
  - Feldtyp
  - Feldlänge
  - Feldwert
- Nützlich bei
  - Informationsintegration: Es ist noch unbekannt welche Felder von Quellen hinzukommen.
  - Dünn besetzte Datensätze: Tausende Attribute, nur wenige haben Werte



Zu große Datensätze

Idee: *Spanned records* überspannen mehr als einen Block.

- Für übergroße Felder
  - „Riesige“ Felder (Mega- Gigabyte) gleich
- Für Datensatzgrößen, die viel Platz verschwenden
  - Z.B. 51% eines Blocks => 49% verschwendet
- Datensatzfragment
  - Falls zu einem Datensatz mehr als ein Fragment gehört, ist er „spanned“.
- Zusätzliche Informationen im Header
  - Bit sagt ob, Fragment oder nicht
  - Bits sagen ob erstes oder letztes Fragment
  - Zeiger zum nächsten und/oder vorigen Fragment
    - Doppelt verkettet Liste

BLOBs

BLOB = Binary Large Object

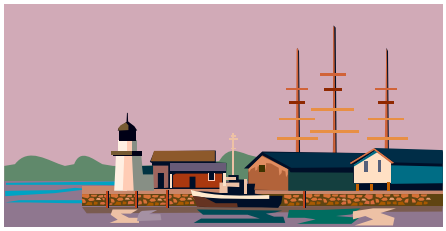
- Bilder/Grafiken: JPEG, GIF
- Audio: mp3, ..
- Filme: MPEG, ...
- Probleme
  - Speicherung: Mehr als ein Block nötig
    - Sequenz von Blöcken/Zylindern
    - Realtime: Lesegeschwindigkeit einer Disk nicht ausreichend => Verteilung auf mehrere Disks
  - Lesen
    - Anweisung, einen (ganzen) Datensatz zu lesen, ist nicht mehr gültig
    - Stattdessen: Kleiner Teil eines Datensatzes lesen
    - Navigation innerhalb des BLOBs (z.B. „Sprung zur 45ten Minute“) => spezielle Indexstrukturen

CLOB = Character Large Object

Überblick

25

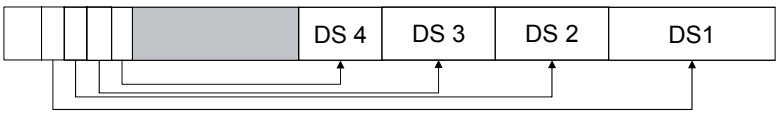
- Datenelemente
- Datensätze
- Adressierung
- Daten variabler Länge
- Datensatzänderungen



Einfügen mit Platz

26

- Einfacher Fall: Keine Ordnung verlangt
  - Suche freien Platz auf einem Block (oder suche freien Block).
  - Füge Datensatz ein.
- Schwierigerer Fall: Ordnung (z.B. nach Primärschlüssel) ist verlangt.
  - Suche entsprechenden Block
  - Falls Platz frei ist, bewege Datensätze auf Block, so dass neuer Datensatz an entsprechende Stelle eingefügt werden kann.



Einfügen ohne Platz

- Variante 1: Suche Block in der Nähe
  - Voriger oder nächste Block
  - Bewege ersten oder letzen Datensatz zu jeweils neuem Block
    - Weiterleitungsadresse in altem Block (Nachsendeauftrag)
    - Bewege gegebenenfalls Datensätze in beiden Blöcken hin und her.
  - Füge neuen Datensatz ein.
- Variante 2: Erzeuge Overflow Block
  - Designierter *overflow block*
  - Adresse im header des ursprünglichen Blocks
  - Overflow Block kann selbst wiederum einen *overflow block* haben.

Löschen

- Nach Löschen
  - Datensätze im Block verschieben um freien Platz zu konsolidieren
  - Oder: Im header eine Liste mit freien Plätzen verwalten
    - Oder: als verkette Liste in den freien Plätzen
- Reorganisation der overflow Blocks möglich.
- Grabsteine (tombstones)
  - Es könnte Pointer auf den zu löschenden Datensatz geben.
  - Grabstein hinterlassen (3 Varianten)
    - Null-Pointer im header
    - Null-Pointer in mapping table
    - Grabsteinbit (= 4Byte?) am Anfang der Datensätze
  - Müssen ewig erhalten bleiben

Update

29

- Bei fester Länge kein Problem
- Bei variabler Länge
  - Gleiche Probleme wie beim Einfügen und Löschen
  - Keine Grabsteinproblematik

Zusammenfassung

30

- Felder
  - Feste oder variable Länge
- Datensätze
  - Header + Felder
- Datensätze mit variabler Länge
  - Pointerliste zu den Feldern
- Blocks
  - Header + Datensätze
- Spanned Records
  - Fragmente
- BLOBs
  - Verteilung (Striping)
  - Auch CLOBs
- Offset tables
  - Im Header eines Blocks
  - Für Löschen, Einfügen oder Längenänderungen
- Overflow Blocks
  - Verkette Liste
  - Bei Einfügen
- Datenbankadressen
  - Disk, Zylinder, Spur, Sektor, Byte im Sektor
  - Oder: Mapping table
- Hybride Adressen
  - Physische Adresse + logische Adresse
- Grabsteine
  - Bei Löschen