Vorlesung Implementierung von Datenbanksystemen

9. Speicherung von Tupeln und Relationen

Prof. Dr. Klaus Meyer-Wegener

Wintersemester 2019/20

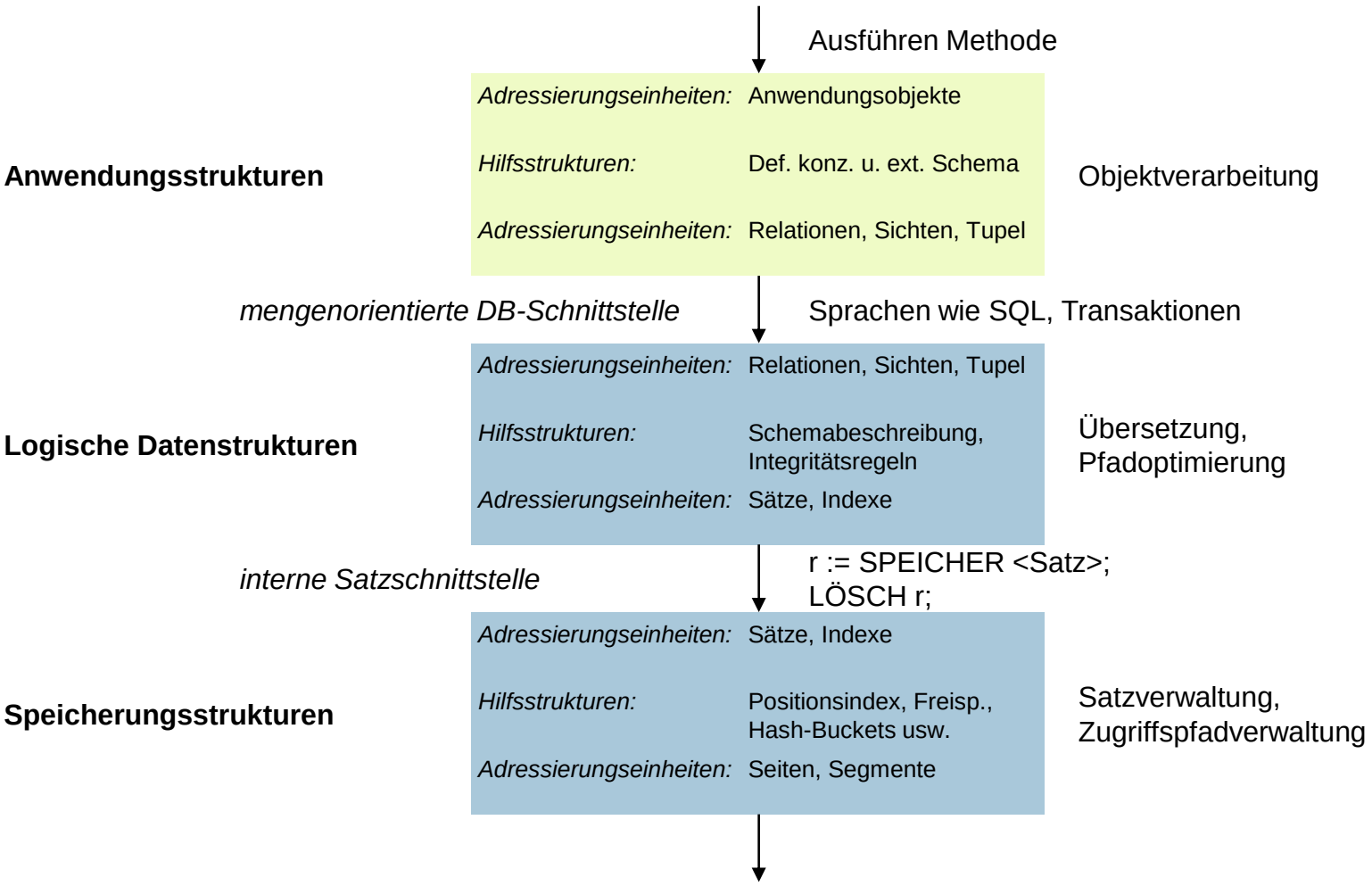
Lehrstuhl für Informatik 6 (Datenmanagement)
Friedrich-Alexander-Universität Erlangen-Nürnberg

Implementierung im engeren Sinne

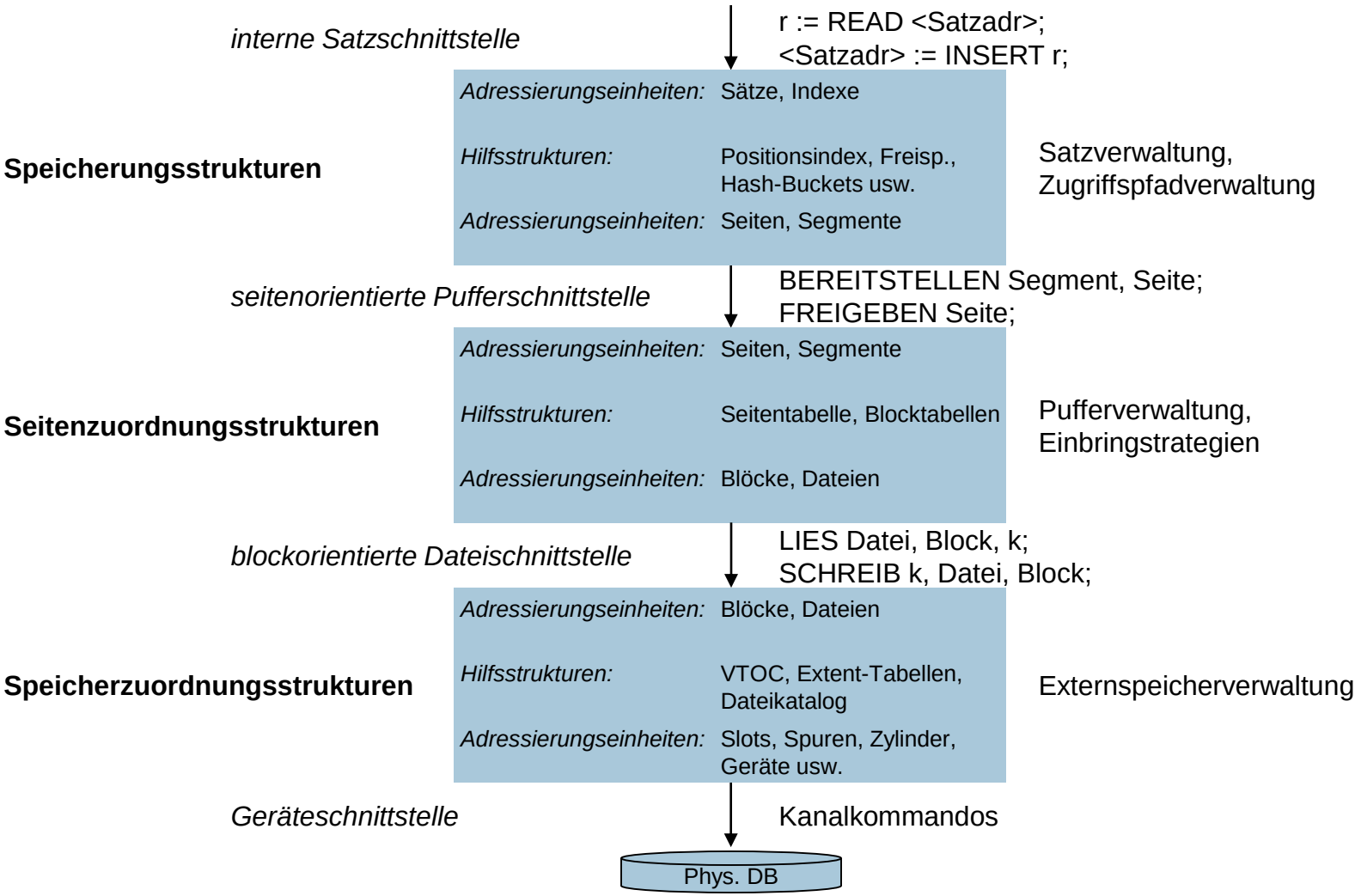
- **Relationen mit den Mitteln der darunter liegenden Schichten abspeichern**
  - Viele Möglichkeiten!
    - Ein Tupel = ein Satz, das geht,
      aber auch: ein Tupel in mehreren Sätzen usw.
- **Anfragen (SQL) möglichst effizient ausführen**
  - Zugriffe auf Indexstrukturen, Seiten, Blöcke, Festplatten, …
  - Ziele: kurze Antwortzeit, guter Durchsatz

- **Von oben nach unten durch das Schichtenmodell**
  - Was passiert mit einer Anfrage?
  - Wie erreicht man transaktionales Verhalten?

Schichtenmodell (Wiederholung)



Schichtenmodell (2)



9.1 Speicherung von Tupeln in Sätzen

- **Sätze sind aus einzelnen Feldern zusammengesetzt.**
  - Die haben einen Namen (Kundennummer, ...),
  - einen Typ (integer, boolean, ...) und
  - eine feste oder variable Länge (in Bytes).
- **Systemkatalog**
  - Informationen über die Felder und ihre Reihenfolge
- **Metadaten (Beispiele):**
  - Name des Feldes
  - Charakteristik (fest, variabel, multipel)
  - Länge
  - Typ (alphanumerisch, numerisch, ...)
  - Besondere Methoden bei der Speicherung
    - z.B. Nullwertunterdrückung, Zeichenverdichtung, Verschlüsselung
  - Symbol für den undefinierten Wert

Speicherung von Tupeln in Sätzen (2)

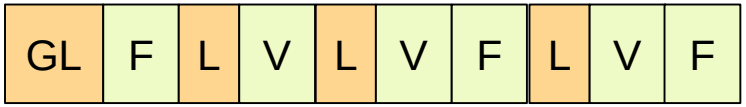
- **Satztyp**
  - Menge von Sätzen mit gleicher Struktur
    - Z.B. Tupel derselben Relation
    - Einmalige Beschreibung im Systemkatalog
  - Jedem Satz wird beim Abspeichern ein Satztyp zugeordnet.
  - Zuordnung zu Segmenten:
    - Typischerweise n:1, manchmal n:m
  - Länge der Sätze eines Satztyps
    - Fest, wenn alle Felder feste Länge haben
      oder bei Feldern variabler Länge immer die Maximallänge reserviert wird
    - Sonst variabel
- **Annahmen**
  - Variable Satzlänge (allgemeinerer Fall)
  - Ein Satz sollte vollständig in einer Seite ablegbar sein.
  - **Reihenfolge der Felder spielt keine Rolle.**

Anforderungen an die Tupelspeicherung

- **Speicherplatzeffizienz**
  - Variable Länge
  - Undefinierte Werte (Nullwerte) gar nicht speichern
  - Möglichst wenig Hilfsstrukturen (Längenfelder, Zeiger)
- **Direkter Zugriff auf Felder**
  - ohne vorher andere Felder lesen zu müssen
  - Direkt zur Anfangs-Byte-Position des Feldes in einem Satz
- **Flexibilität**
  - Hinzufügen von neuen Feldern bei allen Sätzen
    - Undefinierter Wert bei den schon gespeicherten Sätzen
    - Kann am Ende angefügt werden, da Reihenfolge beliebig
  - Löschen eines Feldes aus allen Sätzen
    - Im Systemkatalog als ungültig kennzeichnen
    - Überall undefinierten Wert eintragen?
    - Überall Feld löschen??

Speicherungsstrukturen in Sätzen

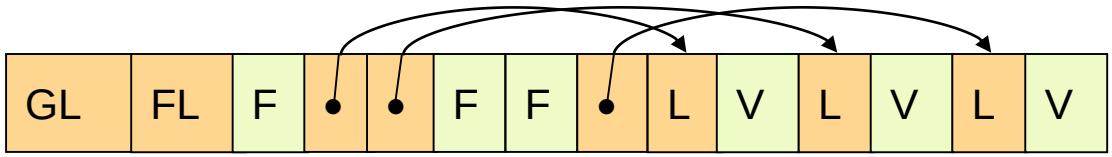
- **Konkatenation von Feldern fester Länge?**
  - Zu speicheraufwändig (immer Maximallänge)
- **Zeiger im Vorspann?**
  - Unflexibel (beim Hinzufügen von Feldern)
- **Eingebettete Längenfelder**
  - Dynamische Erweiterung möglich
  - Keine direkte Berechnung der satzinternen Adresse eines Felds aus Katalogdaten möglich



GL	Gesamtlänge
F	**Inhalt** fester Länge (aus Systemkatalog)
L	Längenangabe für jedes Feld variabler Länge
V	**Inhalt** variabler Länge

Speicherungsstrukturen in Sätzen (2)

- **Eingebettete Längenfelder mit Zeigern**
  - Fester Strukturteil: Felder fester Länge und Zeiger
  - Variabel lange Felder ans Ende des festen Strukturteils legen
  - Satzinterne Adresse aus Katalogdaten berechenbar



GL	Gesamtlänge
FL	Länge des festen Strukturteils
F	**Inhalt** fester Länge (aus Systemkatalog)
L	Längenangabe für jedes Feld variabler Länge
V	**Inhalt** variabler Länge

9.2 Speicherung von Relationen

- **Noch viel mehr Möglichkeiten!**
  - Tupel über mehrere Sätze verteilen (fragmentieren)
  - **Spaltenweise** abspeichern ("Column-Store")
    - Gut für analytische Auswertungen (Data Warehouse)
    - Nicht so gut für Änderungen und Verbünde

Spaltenweise Speicherung

- **Auf das Lesen hin optimiert**
  - Nicht auf das Schreiben hin wie bei den üblichen Relationalen DBVS
  - Also Ad-hoc-Anfragen, Auswertung großer Datenmengen, Data Warehouse
- **Nur die Attribute einlesen, die gebraucht werden**
- **Außerdem dichte Speicherung der Attributwerte**
  - Nicht auf Wortgrenzen etc. ausrichten
  - Kostet CPU-Zeit (Umspeichern), spart aber Speicherplatz und E/A-Zeit
- **Verwendet eine beliebige, aber feste Reihenfolge der Tupel**
  - Kann definiert werden durch Reihenfolge der Satzverweise (s. oben), also entweder der Primärschlüsselwerte oder der Satzadressen

Prototyp C-Store

[Ston05a]

- **Speichert Sammlung von Attributgruppen**
  - Jede nach einem anderen Attribut geordnet
- **Attributgruppe wird Projektion genannt**
  - Alle Spalten der Gruppe haben dieselbe Tupelreihenfolge
- **Speicherung:**
  - **Schreibspeicher** (writable store, WS)
    - für schnelles Einfügen und Ändern von Tupeln
  - **Lese-optimierter Speicher** (read-optimized store, RS)
    - für umfangreiche Analysen
  - Tuple Mover dazwischen, im Hintergrund asynchron von WS zu RS
  - Änderungen (Update) durch Löschen und Einfügen realisieren

C-Store: Projektionen

- **Datenmodell**
  - Relational wie gehabt, SQL ohne Änderungen
- **Projektion**
  - An einer Relation verankert
  - Ein Attribut oder mehrere Attribute dieser Relation
  - Ggf. auch Attribute anderer Relationen,
    falls diese über eine Kette von Fremdschlüsseln erreichbar sind
  - Duplikate bleiben erhalten
  - *i*-te Projektion der Tabelle *t*: ***ti***

```
EMP1 (name, age)
EMP2 (dept, age, DEPT.floor)
EMP3 (name, salary)
DEPT (dname, floor)
```

C-Store: Spalten und Segmente

- **Spaltenweise Speicherung der Projektionen**
  - Jedes Attribut mit allen seinen Werten separat in einer Art Array gespeichert
    - Ergibt auch einen Satz …
  - Gleiche Tupelreihenfolge bei allen in derselben Projektion
    - Sortierschlüssel: eines der Attribute
  - Im Beispiel:

```
EMP1 (name, age | age)
EMP2 (dept, age, DEPT.floor | DEPT.floor)
EMP3 (name, salary | salary)
DEPT (dname, floor | floor)
```

  - Zusätzlich auch noch horizontale Partitionierung möglich:
    - Mehrere **Segmente** mit *Sid* > 0
    - Werte-basiert, Intervalle des Sortierschlüssels

C-Store: Speicherschlüssel

- **Für jede Relation: *überdeckende* Menge von Projektionen**
  - Rekonstruktion vollständiger Tupel muss möglich sein
- **Speicherschlüssel (Storage Keys, SK):**
  - In jedem Segment mit jedem Attributwert verbunden
  - Gleicher SK bei verschiedenen Attributen = gleiches Tupel
  - Im Lesespeicher (RS):
    - Durchnummeriert: 1, 2, 3, …
    - Nicht gespeichert, sondern aus der physischen Position des Tupels (bzw. seiner Attributwerte) ableitbar
  - Im Schreibspeicher dagegen:
    - Explizit gespeichert als Festpunktzahlen
    - Größer als der größte im RS vorkommende Wert

C-Store: Verbund-Index, Rekonstruktion

- **Verbund-Indexe (Join Indices):**
  - Seien T1 und T2 Projektionen der Relation T
  - Join Index von den M Segmenten in T1
    zu den N Segmenten in T2:
    - Sammlung von M Tabellen, je eine pro Segment von T1, mit Zeilen:

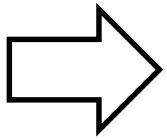
    `(s: SegmentID in T2, k: Storage Key in Segment s)`
  - Immer 1:1
  - Quasi Umsortierung der Tupel in T1
  - **Rekonstruktion von T** ist möglich, wenn es immer einen Pfad von Verbund-indexen gibt, der von einer Projektion (und ihrer Sortierordnung) aus zu allen anderen Attributen führt (und die in diese Sortierordnung bringt).

C-Store: Verbund-Index, Rekonstruktion (2)

EMP

name	*age*	*dept*	*salary*
Müller	43	E	50
Meyer	27	V	60
Schulze	36	E	40
Schmidt	45	P	30

(Nur ein Segment, deshalb *s* weggelassen)



EMP1

name	***age***	*k*
Meyer	27	4
Schulze	36	2
Müller	43	3
Schmidt	45	1

EMP3

name	***salary***
Schmidt	30
Schulze	40
Müller	50
Meyer	60

C-Store: Physische Strukturen

- **Einzelne Spalten**
  - **Komprimierung**, abhängig von zwei Eigenschaften der Spalte:
    - Sortierung (nach der Spalte selbst): ja/nein
    - Anzahl der verschiedenen Werte: wenige/viele
  - **1. Sortiert mit wenigen verschiedenen Werten**
    - **Tripel (v, f, n)**
      - v – Wert (value)
      - f – Position des ersten Auftretens
      - n – Anzahl der gleichen Werte
    - Organisiert in einem B-Baum (Primär-Organisation)
      - mit dichter Packung (alle Seiten voll)
        - Keine Änderungen!
      - und großen Seiten
        - Geringe Höhe!

C-Store: Physische Strukturen (2)

- **Einzelne Spalten (Forts.)**
  - **2. Unsortiert mit wenigen verschiedenen Werten**
    - Paare (v, b)
      - v – Wert (value)
      - b – **Bitmap**
    - Lauflängencodierung für die Bitmaps
    - Offset-Indexe: B-Baum für Abbildung von Positionen in einer Spalte auf die Werte in dieser Spalte
      - Auffinden des i-ten Werts einer Spalte
      - B-Baum über (Index der ersten 1, Wert)
        - Erspart das Prüfen aller Bitlisten an einer bestimmten Position, um die Bitliste zu finden, in der die 1 gesetzt ist.
  - **3. Sortiert mit vielen verschiedenen Werten**
    - **Delta-Codierung**: Differenzen zum Vorgänger-Wert speichern
    - In jeder Seite zuerst absoluten Wert (und Speicherschlüssel)
    - Wieder B-Baum (Primär-Org.) mit dichter Packung

C-Store: Physische Strukturen (3)

- **Einzelne Spalten (Forts.)**
  - **4. Unsortiert mit vielen verschiedenen Werten**
    - Unkomprimiert
    - B-Baum mit dichter Packung als Sekundär-Organisation möglich
  - **5. Bei Zeichenketten zusätzlich noch Wörterbuch (Dictionary)**
    - (Sortierte) Liste aller vorkommenden Werte
    - In der Spalte nur noch Indexposition in dieser Liste
      - Ganze Zahl, also feste Länge!

- **Verbund-Indexe**
  - Zwei Attribute: SegmentID s, Speicherschlüssel k
  - Speicherung wie die anderen Spalten auch
  - Leider Fall 4

C-Store: Schreibspeicher (WS)

- **Keine zwei verschiedenen Optimierer schreiben:**
  - WS hat genau die gleichen Projektionen und Verbund-Indexe wie RS
  - Physische Strukturen aber anders
- **Speicherschlüssel**
  - Explizit gespeichert in jeder Projektion und jedem Segment
  - Festpunktzahl, größer als Zahl der Sätze im größten Segment des RS
- **Gleiche Segmentierung wie RS**
- **Keine Komprimierung**
  - B-Baum zur Sortierung nach Speicherschlüssel

C-Store: Weitere Aufgaben

- **Speicherverwaltung**
  - Vor allem Allokation
- **Änderungen und Transaktionen**
  - Schnappschuss-Isolation
  - Synchronisation mit Sperren
- **Tuple Mover**
  - Von WS in den RS
  - Hintergrund-Aufgabe
- **Anfrageausführung**
  - Operatoren und Pläne
  - Optimierung
- **Siehe [Ston05a]**

C-Store: Ausblick

- **Inzwischen als Produkt auf dem Markt: Vertica**
  - www.vertica.com
  - Gehört inzwischen zu HP
- **Etliche ähnliche Ansätze:**
  - Produkte: Sybase IQ, KDB, EXASOL, SAP HANA
  - Prototypen: Addamark, Bubba, MonetDB
- **Allgemein akzeptierter Ansatz für Anwendungen mit hohem Lese- und Auswertungsanteil**
  - Z.B. Data Warehouse

Quelle

[Ston05a]

STONEBRAKER, Mike ; ABADI, Daniel J. ; BATKIN, Adam ; CHEN, Xuedong ; et al.: C-Store: A Column-Oriented DBMS. In: *Proc. 31st Conf. on VLDB 2005* (Trondheim, Norway), pp. 553-564