Prolog – IDB WS 2020/21

Demian E. Vöhringer
Lehrstuhl für Informatik 6, Friedrich-Alexander-Universität Erlangen-Nürnberg
5. November 2020

Übersicht

Intro

Vorlesung

Übung

Prüfung

Intro

Wer für wen?

Betreuung

Demian E. Vöhringer

`cs6.tf.fau.de/dev`

Dozent

Prof. Dr. Klaus Meyer-Wegener

`cs6.tf.fau.de/kmw`

Studiengänge

- Bachelor (Wirtschafts-)Informatik, EEI & Mathematik
- Master IuK & Berufspädagogik Technik
- Lehramt Informatik an Gymnasium, Real- & Mittelschule
- ...

Voraussetzungen

- **Konzeptionelle Modellierung**
  Besonders:
  - Relationenmodell
  - Relationenalgebra
  - SQL
- **Grundlegende Programmierung**
  Besonders:
  - Java o.ä.
  - Algorithmen und Datenstrukturen
    z.B. Sortieren, Suchen, Bäume, Hashing
- **Betriebssysteme**
  Besonders:
  - Kapitel 12. Speicherverwaltung
  - Kapitel 13. Dateisystem
  - Und Prozesse

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

Vorlesung

Ziel der Vorlesung

- Innenleben eines Datenbankverwaltungssystems (Datenbank-Management-Systems, DBMS)
  - Speicherverwaltung
  - Indexstrukturen
  - Pufferverwaltung
  - Anfrageverarbeitung
  - ...
- Datenabstraktion, -kapselung, -unabhängigkeit
  - Schichtenbildung
  - „Virtueller“ Speicher

Vorlesung

ACHTUNG: Nur als Videoaufzeichnung

- Aufzeichnung aus dem WS2019/20 $\rightarrow$ StudOn
- Empfohlene Einteilung per Semesterplan $\rightarrow$ StudOn
- Wöchentliche Fragestunde zur aktuellen Vorlesung (Mittwoch, 9 Uhr)
  Fragen vorher ins Forum.
  Keine Fragen $\rightarrow$ Keine Fragestunde.
- Skript (Folien, PDF) $\rightarrow$ StudOn
- Aktuelle Informationen $\rightarrow$ StudOn

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

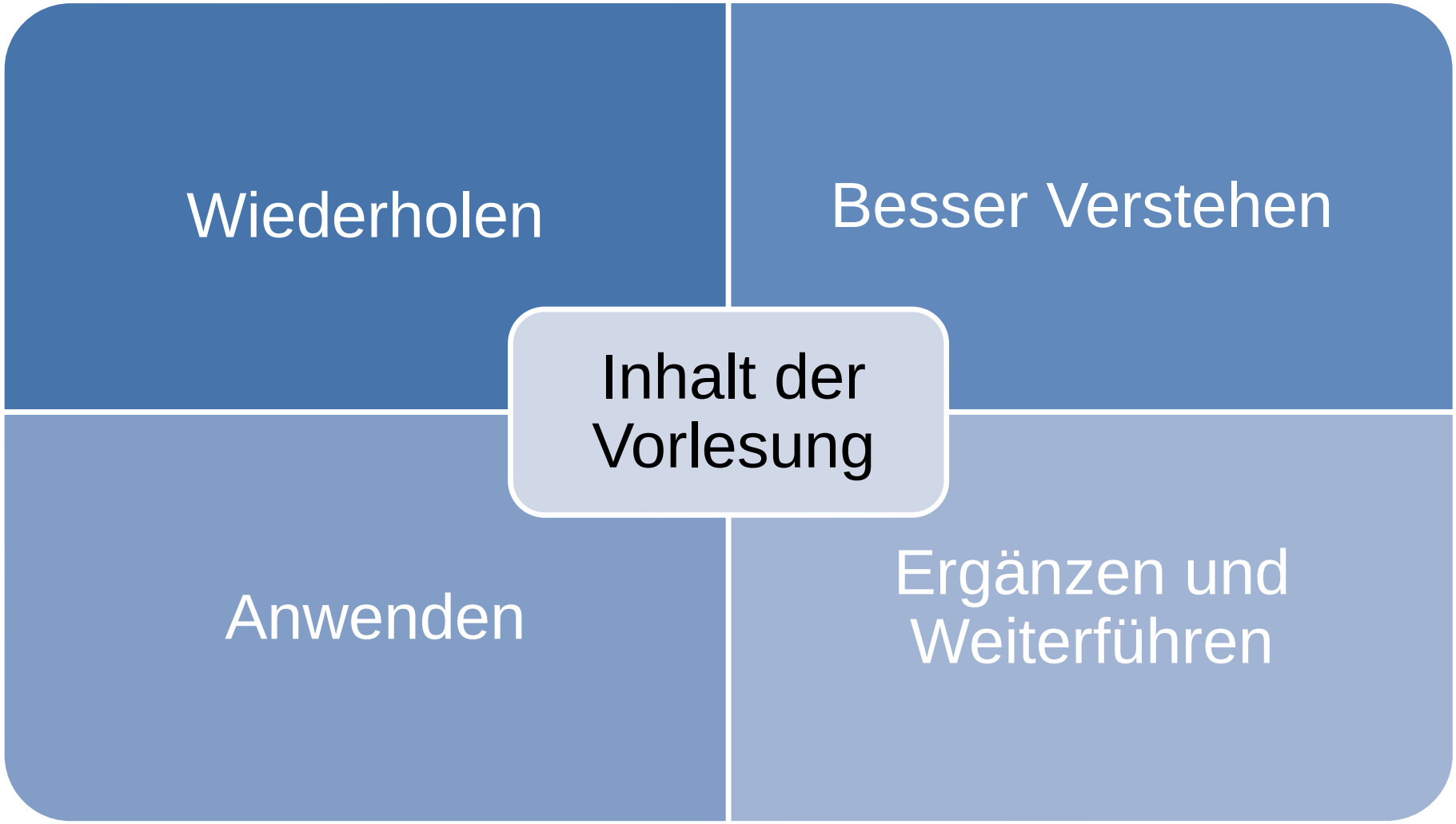
Übung

Man hilft den Menschen nicht, wenn man für sie tut, was sie selbst tun können.

Abraham Lincoln

12.02.1809 – 15.04.1865

Sinn der Übung



Anmeldung

- Über StudOn
- Freischaltung ab jetzt
- Chancengleichheit bis heute, 20 Uhr
- Zunächst begrenzte Plätze
  Gruppen werden iterativ vergrößert
- Erstes Blatt: diese Woche in Eigenarbeit
- Erste Übung: nächste Woche

StudOn – Wozu?

- Zugriff auf Unterlagen
- Möglichkeiten zur freiwilligen Abgabe von Übungsaufgaben
- Forum für Diskussionen und Ankündigungen
- Möglichkeit zur Benachrichtigung aller Teilnehmer per E-Mail
- Gleichmäßige Verteilung auf Übungsgruppen

Das Übungsblatt

- Präsenzaufgaben
  - Eigenständig vorbereiten
  - Besprechung in der Tafelübung
  - Musterlösung wird veröffentlicht
- $*$ Vertiefungsaufgaben
  - Vertiefen oder erweitern den Stoff
  - Abgabe auf StudOn
  - Korrektur durch Tutor
  - Keine Musterlösung
- $*$ Programmieraufgaben
  - Praxiserfahrung
  - Abgabe auf GitLab
  - Korrektur durch Testcases und Tutor

Übungswoche

- Übungsblätter mit Präsenz- und Vertiefungsaufgaben auf StudOn

  Ab Mittwoch der Vorwoche
- Eigenständiges Vorbereiten der Präsenzaufgaben
- Besprechung der Aufgaben in den Übungen

  In der Übungswoche
  - Kein Monolog der Tutoren
  - aktive Mitarbeit und Diskussion
  - Fragen stellen
  - Lösungsalternativen
- Musterlösung der Präsenzaufgaben auf StudOn

  Nach der letzten Übung der Übungswoche
- Abgabe der Zusatzaufgaben

  Bis Donnerstag 14 Uhr der Folgewoche

Betreuung

Demian E. Vöhringer

Fragestunde Mi 8:15

cs6.tf.fau.de/dev

- Tobias Heineken

  Mi 16:15

  Intensivierungsübung

- Jonny Schäfer

  Mo 8:15

  Mo 10:15

- Joshua Mennecke

  Mo 16:15

  Di 16:15

Übungen – Termine

ÜG1	Mo	8:15 – 9:45	Jonny Schäfer
ÜG2	Mo	10:15 – 11:45	Jonny Schäfer
ÜG3	Mo	16:15 – 17:45	Joshua Mennecke
ÜG4	Di	16:15 – 17:45	Joshua Mennecke
ÜG5	Mi	16:15 – 17:45	Heineken Tobias
IntÜ	Do	16:15 – 17:45	Heineken Tobias

Unklarheiten

Wen bzw. wo Fragen?

- Kommilitonen
- Forum in StudOn
- Intensivierungsübung
- Fragestunde
- Sprechstunde Demian E. Vöhringer
  cs6.tf.fau.de/dev
- Sprechstunde Prof. Meyer-Wegener
  cs6.tf.fau.de/kmw

Intensivierungsübung

Fragen zum Vorlesungsstoff?

Offene Fragen nach der Besprechung des Übungsblatts?

Weitergehnde Fragen zum Übungsstoff?

Intensivierungsübung
Do 16:15
Bitte pünktlich zu Beginn besuchen!

Arbeitsaufwand & Zeitmanagement

Eine durchschnittliche Studentin oder ein durchschnittlicher Student sollte in zwei Stunden pro Woche die Aufgaben lösen und vertiefen können.

(Zzgl. Nachbereitung der Vorlesung: mind. 1h/w)

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

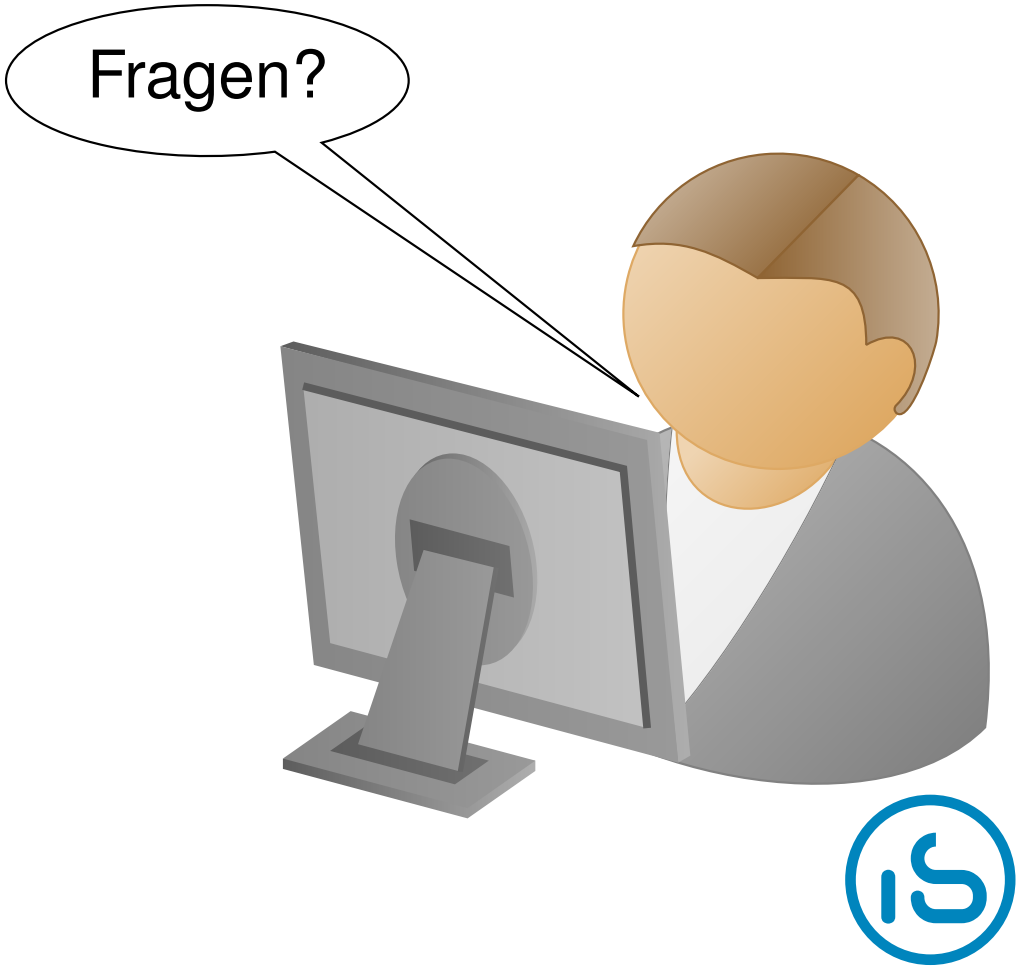
Prüfung

Prüfung

- Schriftliche Prüfung
  - Im Prüfungszeitraum am Ende des Wintersemesters
  - 90 Min
  - Vmtl. am Freitag, den 19. Februar 2021
- Prüfungsstoff
  - Vorlesung **und** Übung
  - Weder Übungsstoff alleine noch Vorlesungsstoff alleine reichen zum Bestehen!
- Prüfungsvorbereitung:
  - Nachbereiten der Vorlesung
  - Selbstständiges bearbeiten der Aufgaben
  - Übungsteilnahme
- Anmeldung auf meinCampus nicht vergessen!

ToDo

- Freitag 16:00 Uhr: Fragestunde zur Vorlesung 1 und 2
- Blatt 1 in Eigenarbeit
- Blatt 2 Vorbereiten
- Übungsgruppen Beitritt auf StudOn



Vorlesung Implementierung von Datenbanksystemen

1. Einführung

Prof. Dr. Klaus Meyer-Wegener

Wintersemester 2019/20

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

Datenabstraktion

- **Auch: Datenkapselung, Datenunabhängigkeit (der Anwendung), Information Hiding, …**

- **Speichern und Wiedergewinnen von Daten**
- **ohne Kenntnis der Details der Speicherung**
  - Keine Adressen, sondern Namen
  - Keine Bytefolgen, sondern Typen (einer Programmiersprache)
- **Persistente (dauerhafte) Speicherung**
  - Daten bleiben über Programmende oder Rechnerabschaltung hinaus erhalten
- **Wiedergewinnen =**
  Auffinden (Suchen, Lokalisieren) und
  Aushändigen (Übergeben – im richtigen Format!)

Generische Software

- **Altes Prinzip der Informatik bzw. der Programmierung:**
  - Was man in verschiedenen Anwendungen immer wieder tun muss
    - Ausgabe auf Drucker oder Bildschirm,
      Kommunikation über ein Netz,
      persistente Speicherung von Daten usw.
  - verallgemeinern (Abstraktion)
  - und in Unterprogramme auslagern.
- **Generische (anwendungsübergreifende) Lösungen**
  - **Konfigurierbar** für die Zwecke einer bestimmten Anwendung
  - durch Parameter oder **Schema**
- **Operationen, Schnittstelle**
  - Was braucht man?
  - Vollständigkeit …

1.1 Schichtenbildung

- **Strukturierung komplexer Software-Systeme**
  - Z.B. ISO/OSI für Kommunikationsprotokolle
- **Entwurfsmuster**
- **Schicht**
  - realisiert einen bestimmten **Dienst** (abstrakt)
    den sie an der **Schnittstelle** (konkret) "nach oben"
    der darüber liegenden Schicht
    zur Verfügung stellt
  - nimmt dafür Dienste darunter liegender Schichten in Anspruch
  - verbirgt die darunter liegenden Schichten vollständig
    (Datenkapselung, siehe oben)
  - muss daher *alle* erforderlichen Funktionen anbieten

Schichtenbildung (2)

- **Beobachtung**
  - Entlang der Schichten wird nach oben hin "abstrahiert" (Veredelungshierarchie).
    - Weniger lästige Details, mächtigere Funktionen
  - "Eine Hauptaufgabe der Informatik ist die systematische Abstraktion." (Hartmut Wedekind)

- **Problem: Anzahl *n* der Schichten in einem Software-System**
  - *n* = 1: monolithisches System

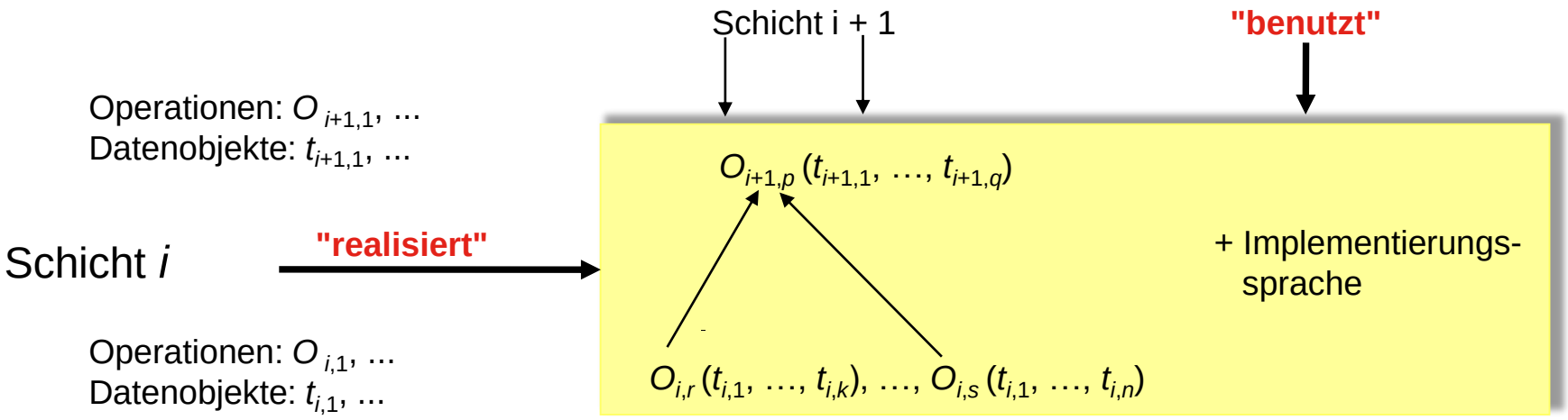
    → keine Schichtenbildung
  - *n* sehr groß:

    → hoher Koordinierungsaufwand, Leistungseinbußen (Schnittstellenüberquerung)
  - Faustregel: *n* typischerweise zwischen 3 und 10

Aufbauprinzip einer Schicht

- **Genauere Begriffsbestimmung "Schicht", "Schnittstelle"**
  - $\{ O_{i,j} \}$: Menge der **Operationen** an der Schnittstelle der Schicht *i*
  - $\{ t_{i,j} \}$: Menge der **Datenobjekte** (Adressierungseinheiten) der Schnittstelle *i*

- **Aufbauprinzip:**



Vorteile der Schichtenbildung

- **Konsequenzen der Nutzung hierarchischer Strukturen:**
  - Höhere Ebenen (Systemkomponenten) werden einfacher, weil sie Dienste der tieferen Ebenen (Systemkomponenten) benutzen können.
  - Änderungen auf höheren Ebenen haben keinen Einfluss auf tiefere Ebenen.
  - Höhere Ebenen können abgetrennt werden, tiefere Ebenen bleiben trotzdem funktionsfähig (wiederverwendbar).
  - Tiefere Ebenen können getestet werden, bevor die höheren Ebenen lauffähig sind.
- **Jede Hierarchieebene kann als abstrakte oder virtuelle Maschine aufgefasst werden.**
  - Programme der Schicht i benutzen die Programme der Schicht i–1 als abstrakte Maschine.
  - Abstrakte Maschine der Schicht i dient wiederum als Basismaschine für die Implementierung der abstrakten Maschine der Schicht i+1.

Vorteile der Schichtenbildung (2)

- **Eine abstrakte Maschine entsteht aus der Basismaschine durch Abstraktion.**
  - Einige Eigenschaften der Basismaschine werden verborgen.
  - Zusätzliche Fähigkeiten werden durch Implementierung höherer Operationen für die abstrakte Maschine bereitgestellt.

DBVS als Beispiel für Schichtenarchitektur

- **Motivation**
  - Bisher bekannt:
    - Ein DBMS ist ein "normales" Anwendungsprogramm, das auf der Grundlage eines Betriebssystems abläuft.
  - Aber:
    - Ein DBMS ist ein großes und komplexes System, das strukturiert aufgebaut sein muss.
- **Folgerung**
  - DBMS ist *das* Beispiel eines schichtweisen Systementwurfs.
- **Fokus**
  - Die richtigen Schnittstellen (Abstraktionen) finden!
  - Mehrere alternative Techniken zur Realisierung eines Dienstes in einer Schicht
  - Zusammenspiel mehrerer Schichten

1.2 Datenbanktechnologie

Konzepte, Methoden, Werkzeuge und Systeme für die

- **dauerhafte**, Lebensdauer Daten > Dauer Erzeugungsprozess
- **zuverlässige**, Integrität, Konsistenz, Sicherheit vor Verlust
- **unabhängige** wechselseitige Änderungsimmunität AP ↔ DBS

Verwaltung und

- **komfortable**, "höhere" abstrakte Schnittstelle
- **flexible** Ad-hoc-Zugriffsmöglichkeit

Benutzung von

- **großen**, Datenvolumen >> Hauptspeicher
- **integrierten**, kontrollierte Redundanz von/für mehrere Anwendungen
- **mehrfach benutzbaren** gleichzeitiger Zugriff

Datenbanken

Einordnung wichtiger DBS-Konzepte

- Datenintegration
- anwendungsorientierte Datenbeschreibung
- Datenunabhängigkeit

} Datenmodelle, Schemata, Sichten

- Konsistenzkontrolle
- Mehrbenutzerbetrieb
- Datenintegrität
- Wiederanlauf

} Transaktionen

- Zugriffskontrolle
- Programmankopplung

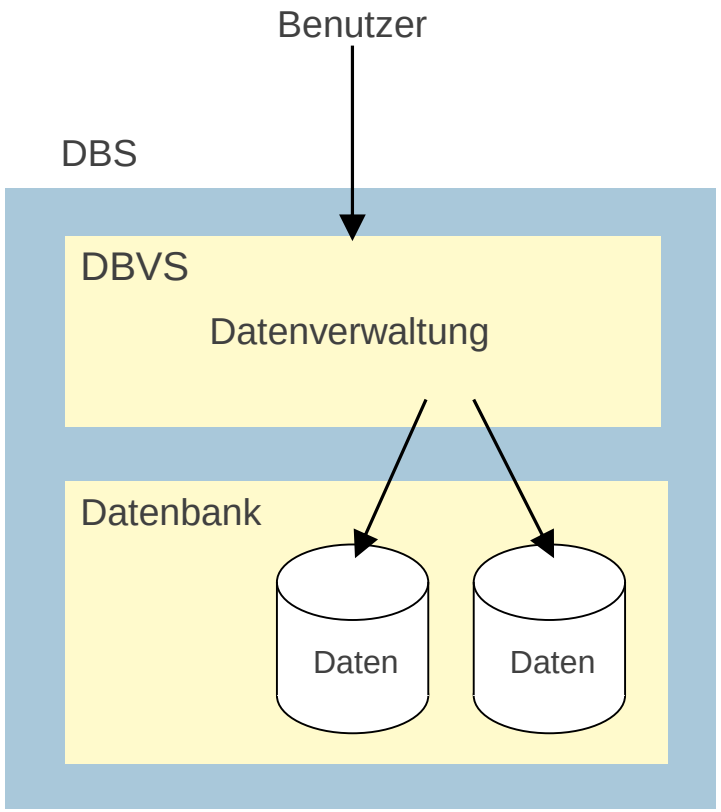
} Datenschutz, Programmiersprachen-einbettung

- Ad-hoc-Anfragen
- Zugriffspfade
- Speicherungsstrukturen
- Verteilung

} Leistungsaspekte, Verfügbarkeit

Datenbanksystem (DBS)

- **Begriff**
  - System zur Beschreibung, Speicherung und Wiedergewinnung von umfangreichen Datenmengen, die von mehreren Anwendungs-programmen benutzt werden
- **Komponenten**
  - **Datenbank (DB)**, in der die Daten abgelegt werden
  - **Datenbank-Verwaltungssystem (DBVS)**:
    Software, die die Daten den vorgegebenen Beschreibungen entsprechend abspeichert, auffindet oder weitere Operationen mit den Daten durchführt
    - (Quelle: Informatik-Duden)



Wann ein Datenbanksystem?

- **Großes Datenvolumen**
  - Auch, aber nicht das Entscheidende

- **Daten sollen wiederverwendbar sein.**
  - Speicherung offen für neue Anwendungen
- **Von mehreren Anwendungen gleichzeitig nutzbar, bei hoher Aktualität der Daten**

(Ziel)

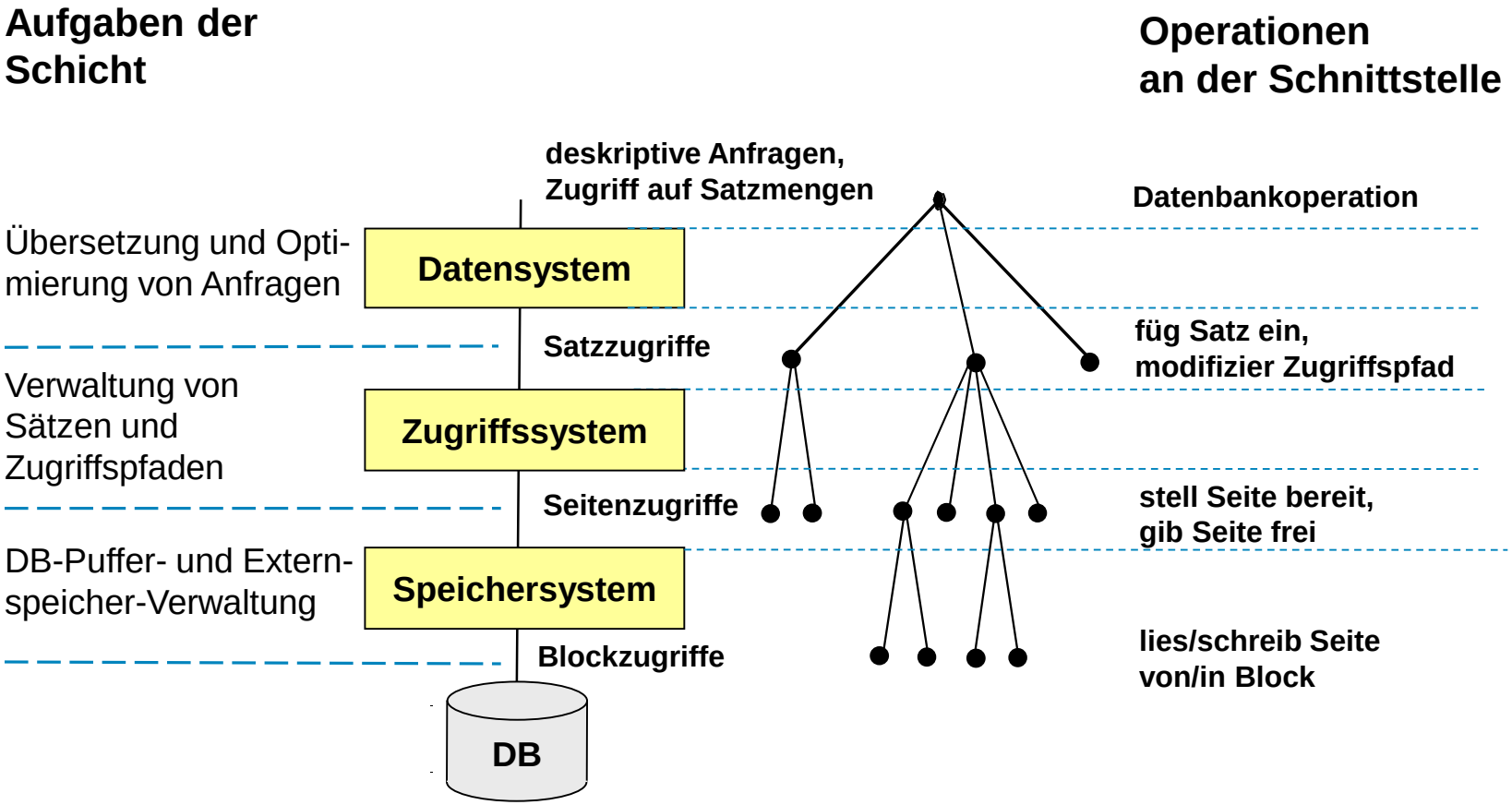
- **Wohlstrukturiert**
- **Redundanzfrei**
- **Flexibel abfragbar (recherchierbar)**
- **Ausfallsicher**

(Konsequenzen)

- **Ach ja, und leistungsfähig, schnell, …**

(Randbedingung)

Schichtenmodell eines DBVS (vereinfacht)



1.3 Zusammenfassung

- **"Schichtologie"**
  - Wesentliches Hilfsmittel zur Konstruktion großer Software-Systeme
  - Eine **Schicht *i*** realisiert für **übergeordnete Schicht *i*+1** einen Dienst unter Inanspruchnahme **darunter liegender Schicht *i*–1**.
- **Schichtenmodell eines Datenbanksystems**
  - Zentrales Hilfsmittel
    - zur Strukturierung
      und auch Visualisierung
      der vielfältigen Aufgaben eines Datenbanksystems und
    - zum generischen Systementwurf
      - Anwendungsspezifika (Schema, Anfragen)
        werden erst zur Laufzeit dem System bekanntgegeben

Vorlesung Implementierung von Datenbanksystemen

2. Dateiverwaltung

Prof. Dr. Klaus Meyer-Wegener

Wintersemester 2019/20

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

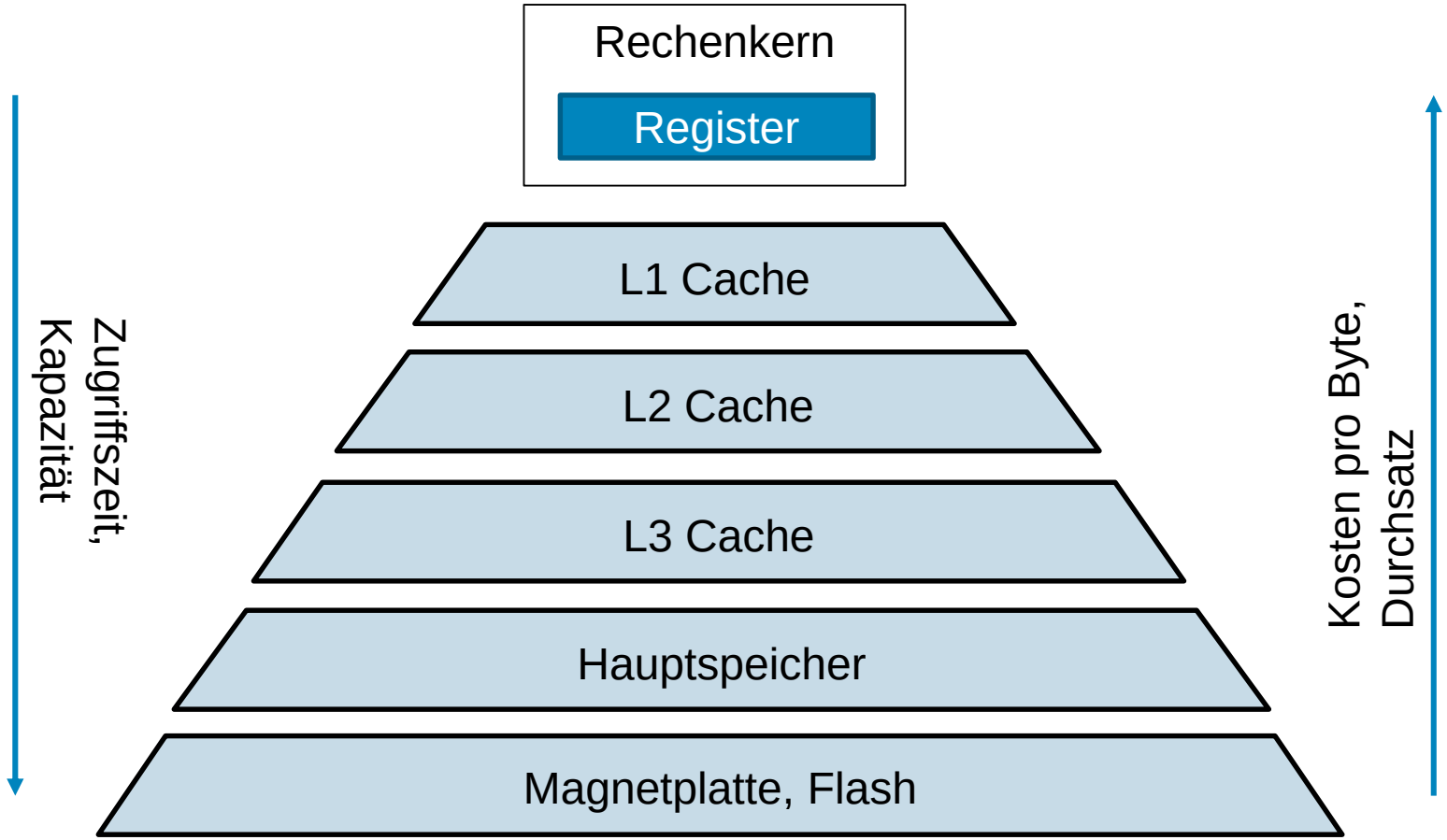
2. Dateiverwaltung

- **Basis-Datenverwaltungsdienst des Betriebssystems**
  - Grundlage für Datenbanksysteme
- **Schichtenbildung:**
  - Ganz unten: **physische Speichergeräte** (Hardware)
  - Darüber: **"logische" Speichergeräte** (mit verbesserter Störsicherheit)
  - Und: **Dateien**
- **Nichtflüchtig ("non-volatile"):**
  - Datenträger, rein passiv, dauerhafte Speicherung auch ohne Strom
- **(Hintergrund-) Speichergeräte ("storage"):**
  - Platzierung und Lokalisierung der Daten mittels Zugriffsmechanismen
  - Linear: Magnetband
  - Wahlfrei: für Datenverwaltung weiterhin am wichtigsten, **Magnetplattenspeicher**

2.1 Physische Speichergeräte

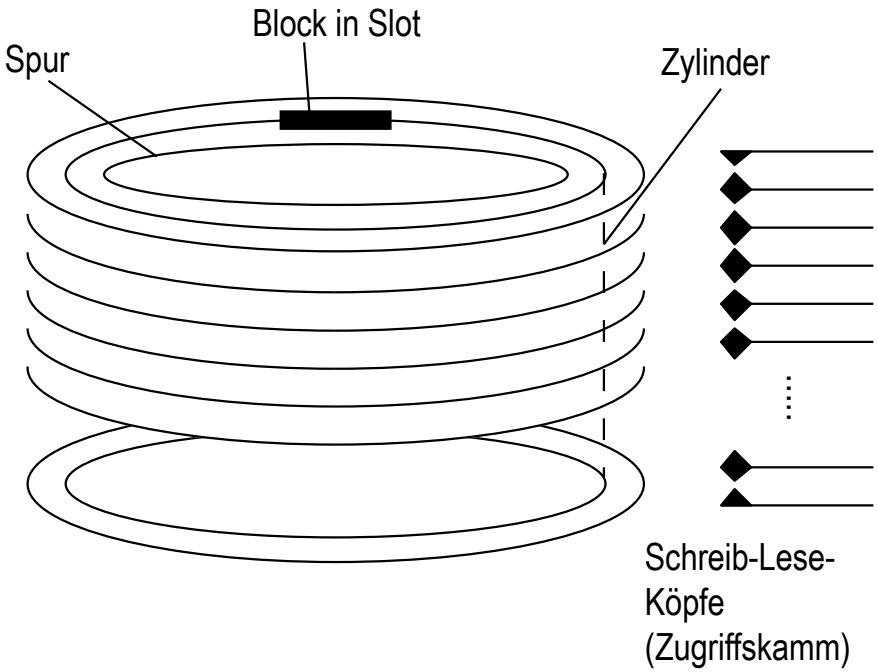
- **Magnetplattenspeicher**
  - Feste Struktur: Zylinder, Spuren, Slots (Sektoren)
  - Feste Größe
  - **Wahlfreier Zugriff** auf einen Slot-Inhalt (= Block): ~ 5 ms
  - Asynchrone Ein-/Ausgabe möglich
- **Magnetbandspeicher**
  - Kostengünstig, **sequenzieller Zugriff**, Sicherung und Archivierung
- **Alternative Speicher**
  - Optische Speicher (DVD, …), MEMS, Flash / SSDs, Holographie, …
    - Forschungsgegenstand
  - Am interessantesten wohl: **Flash**
    - Begrenzte Lebensdauer
    - Schreiben sehr viel langsamer als Lesen
  - Und: **Hauptspeicher** (Arbeitsspeicher, DRAM, "memory")
    - Seit neuestem (2019) sogar nichtflüchtig (NVM)

Speicherhierarchie



Magnetplattenspeicher

- **Schematischer Aufbau:**



- **Block:**
  - Datenmenge, Bytefolge
  - Elementare Einheit des Transports zwischen Platte und Hauptspeicher
- **Slot (Sektor):**
  - Platz auf einer Spur, der zur Aufnahme eines Blocks vorgesehen ist

2.2 Logische Speichergeräte

- **Erhöhung der Störsicherheit**
  - Verdecken von Gerätefehlern
    - Z.B. bei Paritätsfehler nochmal lesen
      und bei Schreibfehler nochmal schreiben
- **Einzelheiten dieser E/A-Prozeduren verdecken**
- **Satz von E/A-Prozeduren bereithalten, parametrisieren**
  - Verwaltung und Aufruf durch eigene Instanz: **Gerätetreiber**
    - E/A-Prozeduren gerätespezifisch, daher für jedes Gerät eigener Treiber
- **Außerdem: flexible Zuordnung von logischen Geräten (Laufwerken) zu den physischen**
  - Mehrere physische Geräte
    als ein großes logisches anbieten (z.B. RAID)
  - Physisches Laufwerk (bzw. Datenträger)
    in mehrere logische Platten aufteilen ("partitionieren")

Logische Speichergeräte (2)

- **Typische Lese- und Schreiboperation ("Dienste"):**

```
int Device::readBlock (
   int CylinderNo, int TrackNo, int SlotNo,
   char *BlockBuffer )
```

  - Rückgabewert: Quittung, Status
  - Speicherplatz für gelesenen Block (Puffer) muss vom Aufrufer vorher angelegt worden sein

```
int Device::writeBlock (
   int CylinderNo, int TrackNo, int SlotNo,
   char *BlockBuffer )
```

- Das heißt: **physische Adressierung** der Slots

Logische Speichergeräte (3)

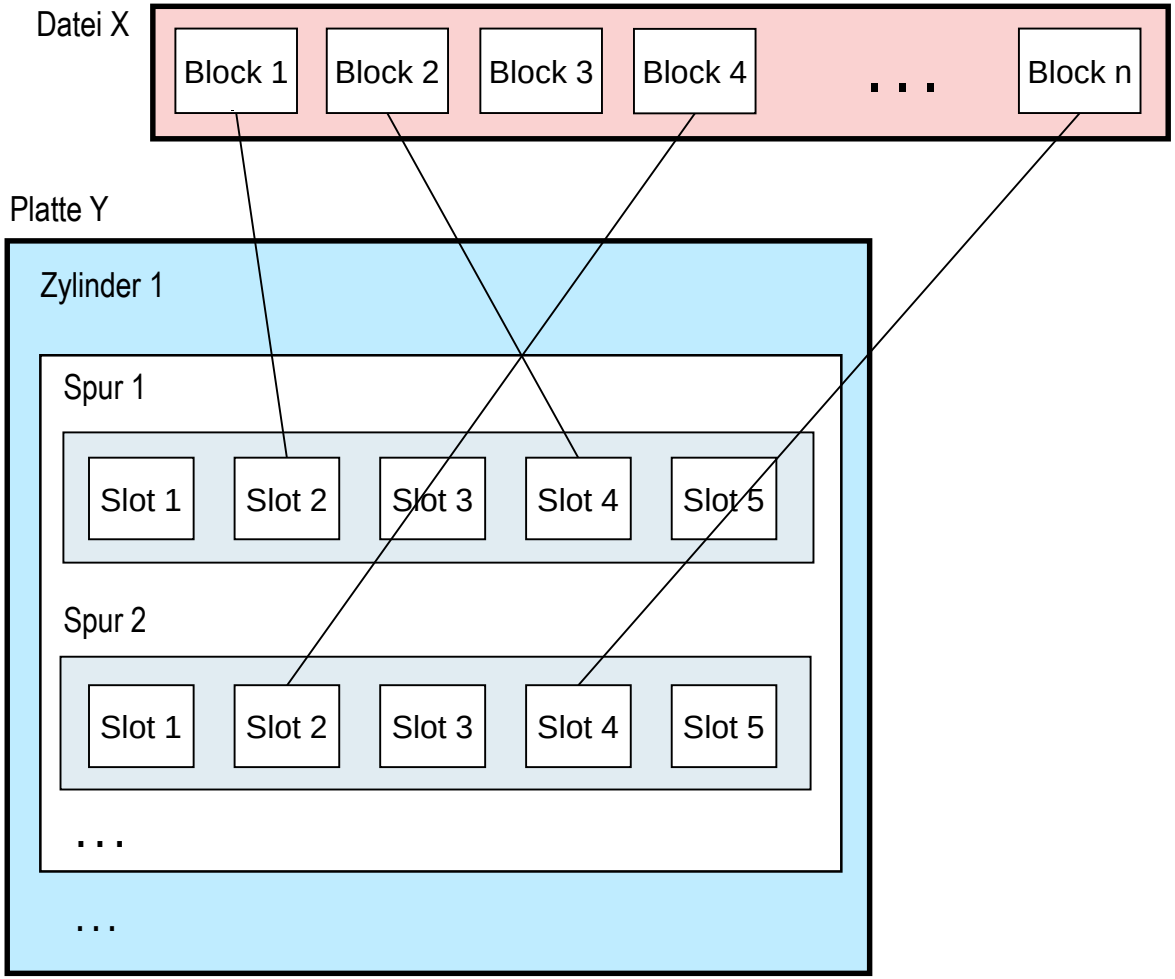
- **Annahme:**
  - Anwendungsprogramme arbeiten direkt mit dieser Ein-/Ausgabeschnittstelle
    - Frühzeit der Datenverarbeitung
- **Vorteile:**
  - Schnell
  - Maximale Speicherausnutzung – keine Verwaltungsdaten
- **Nachteile:**
  - Kein Schutz:
    - Alle Blöcke der Platte von allen Anwendungsprogrammen lesbar und schreibbar
  - Wissen über die Zuordnung der Blöcke (ggf. auch ganzer Spuren oder Zylinder) zu den Anwendungsprogrammen
    - nur außerhalb des Systems (in den Köpfen der Programmierer)
  - Reservierung zusätzlichen Speicherplatzes
    - nur durch Absprache unter den Programmierern
  - Zahlreiche Fehlermöglichkeiten
    - Z.B. Überschreiben eines vermeintlich freien Blocks
  - Oft starre Aufteilung einer Platte mit ungenutztem Platz; schwierig zu ändern

2.3 Dateiverwaltung eines Betriebssystems

- **Einrichten einer Indirektion**
  - zwischen Anwendungsprogramm und logischem Gerät
- **Verwaltung von Dateien:**
  - Haben einen **Namen**
  - **Folge** von Blöcken (linear)
    - Direkt ansprechbar über **laufende Nummer**
    - Auf der Platte aber nicht notwendigerweise sequenziell abgelegt
  - **Dynamisch erweiterbar** um zusätzliche Blöcke
- **Abstraktion**
  - Verborgen: Größe der Magnetplatte; Zylinder und Spuren

(Siehe auch Vorl. Systemprogrammierung 2)

Dateikonzept



Zugriffsoperationen (Dienste)

- **Sog. blockorientierte Zugriffsmethode ("Access Method") für eine Datei**
  - z.B. UPAM im BS2000 von Siemens, BDAM im MVS (heute: z/OS) von IBM
- **Gekennzeichnet durch Operationen wie die folgenden:**

```
BlockFile::BlockFile ( char *FileName, char Mode,
                       int *BlockSize )
```

  - **öffnet die Datei** mit dem angegebenen Namen
    und richtet eine temporäre Verwaltungsdatenstruktur für die Arbeit mit ihr ein ("Dateikontrollblock")
  - **Mode** gibt an, ob Datei gelesen, geschrieben oder ergänzt werden soll
  - Blockgröße definiert bei neuer Datei die Blockgröße
    oder liefert bei vorhandener Datei die verwendete Blockgröße zurück

```
BlockFile::~BlockFile ()
```

  - **schließt die Datei** und löscht den Dateikontrollblock

Schreib- und Leseoperationen


```
int BlockFile::append ( int NumberOfBlocks )
```

- erweitert Datei um die angegebene Zahl von Blöcken
- Rückgabewert gibt an, wie viele Blöcke tatsächlich angelegt werden konnten (bei Fehler: **0**).

```
int BlockFile::write ( int BlockNo, char *BlockBuffer )
```

- überschreibt den angegebenen Block mit dem im Blockpuffer bereitgestellten neuen Inhalt
- Rückgabewert ist ein Fehlercode, der z.B. anzeigt, dass die Blocknummer ungültig ist.

```
int BlockFile::read ( int BlockNo, char *BlockBuffer )
```

- liest den angegebenen Block der Datei in den Blockpuffer

Weitere Operationen

`int BlockFile::size ()`

- gibt die aktuelle Größe der Datei in Blöcken an

`void BlockFile::drop ( int NumberOfBlocks )`

- gibt angegebene Zahl von Blöcken am Ende (!) der Datei wieder frei
- Inverse Operation zu **append**

Operation zum Freigeben beliebiger Blöcke?

- Denkbar, aber in der Realisierung sehr aufwändig
- Außerdem verschieben sich alle Blocknummern hinter den gelöschten, was leicht zum Zugriff auf einen falschen Block führen kann!

Dateikatalog

- **Verwaltungsdatenstruktur**
  - Verzeichnis aller Dateien auf einem oder mehreren Speichergeräten
  - Selbst mit auf der Platte abgelegt
    - Meist auf einer festen Position (Zylinder 0, Spur 0, Block 0 o.ä.)
  - Kann hierarchisch gegliedert sein (UNIX, VMS, MS-DOS, …)
  - Leistet für jede Datei
    die **Abbildung des Dateinamens auf eine Folge von Blöcken**
    - Muss dann zu einer gegebenen Blocknummer
      die physische Slot-Adresse liefern können
- **Zusätzlich: Freispeicherverwaltung**
  - Liefert die unbenutzten Blöcke auf einer Platte
    (die momentan keiner Datei zugeordnet sind)
    - Also schnell zu finden bei **append**
  - Z.B. als Bitliste

Geräteunabhängigkeit

- **Vergleich:**
  - Programme, die mit blockorientierten Schreib-Lese-Operationen arbeiten, und Programme, die direkt auf die Platte zugreifen

Kriterium	direkt	blockorientiert
Geschwindigkeit	sehr schnell	langsamer wegen Katalogzugriff
Verwaltungsdaten	keine	Katalog
Schutz	keiner	Prüfung von Zugriffsrechten beim Öffnen
Erweiterung einer Datei	nur durch externe Absprache	systemverwaltet (Freispeicher, Extents)
Reorganisation (Verschieben von Blöcken)	Programmänderung	Änderung des Katalogs, Programme stabil
Wechsel des Plattenspeichertyps	Programmänderung	Änderung des Katalogs, Programme stabil

Geräteunabhängigkeit (2)

- **Anwendungsprogramme,**
  die blockorientierten Dateizugriff verwenden, sind
  unabhängig von verwendeten Speichergeräten
  (Magnetplatten, opt. Platten, Flash)

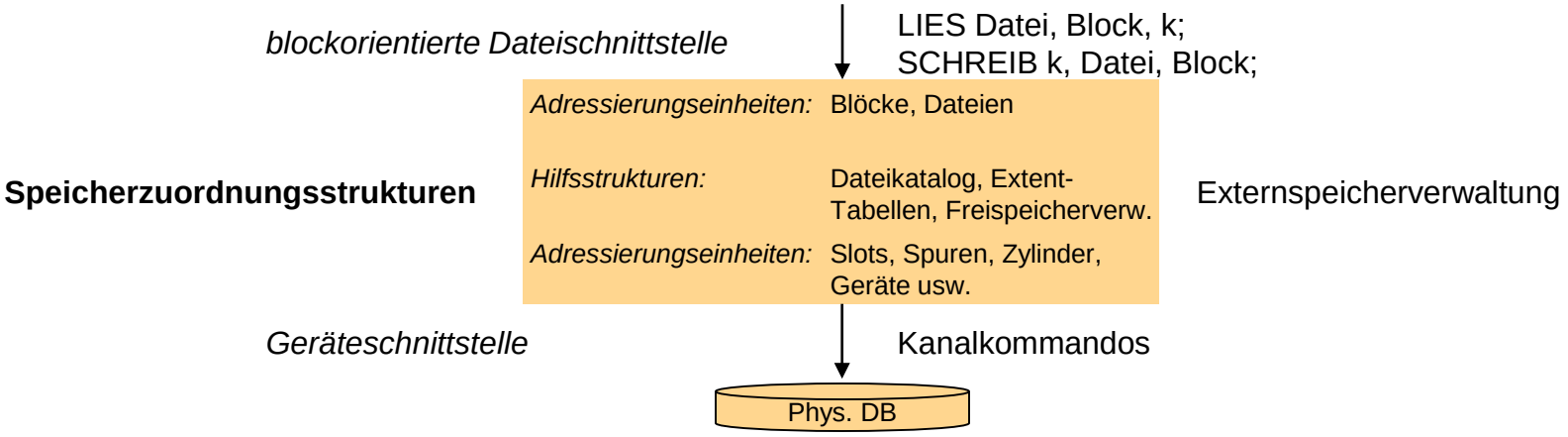
- **Das heißt:**
  - Speichergerät kann reorganisiert oder ausgewechselt werden, ohne dass Programme geändert werden müssen
  - Für Programme ist Datei abstrakte Sicht auf diverse, evtl. verstreute Abschnitte (Slots) eines oder mehrerer Plattengeräte
    - Deshalb auch: "virtuelles Speichergerät" oder "logische Platte" – aber eigentlich noch abstrakter als das Logische Speichergerät von oben
  - Für dieselbe Datei verschiedene Abbildungen auf Speichergeräte möglich
    - Slots direkt hintereinander auch möglich!

Geräteunabhängigkeit (3)

- **Ein Fall von "Information Hiding":**
  - Abbildung der Blöcke einer Datei
    auf Slots eines Plattenspeichers
    wird dem Anwendungsprogramm verborgen

- **Im folgenden:**
  - Blockdatei selbst Basis für weitere Abstraktionen (weitere virtuelle Speicher)

Resultierende Schicht

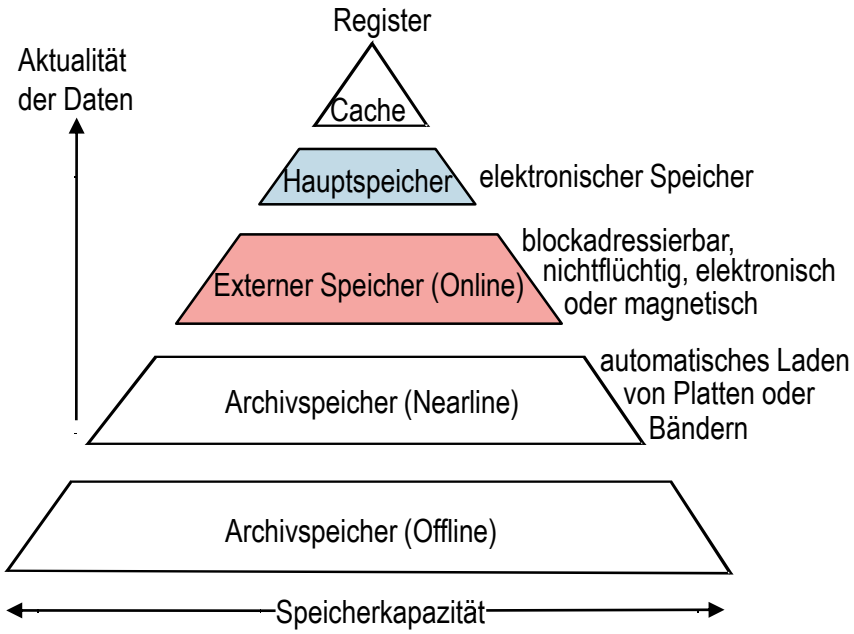


Anhang

- **Nicht mehr verwendete Folien**
- **Zum Vertiefen einzelner Aspekte und zum Nachschlagen bei Bedarf**

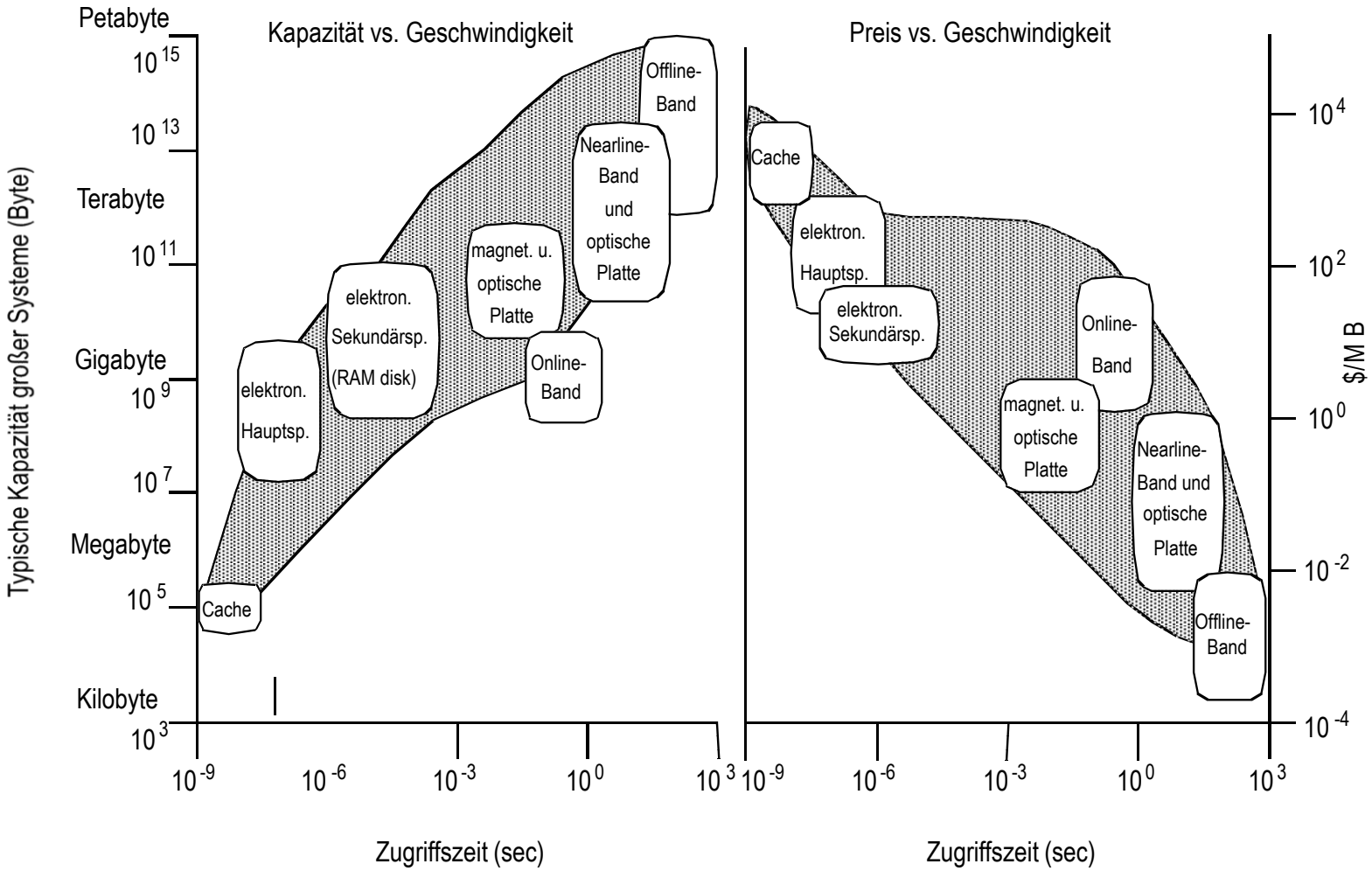
2.A.1 Physische Speichergeräte

- **Anforderungen**
  - Verwaltung externer Speichermedien / Unterstützung von Speicherhierarchien
  - Adressierung physischer Blöcke
  - Kontrolle des Datentransports vom/zum Hauptspeicher
  - Fehlertoleranzmaßnahmen (RAID-Systeme, ...)
- **Datenbankperspektive**
  - Betriebsysteme realisieren einen Verzeichnisdienst zur Verwaltung von Dateien mit unstrukturiertem Inhalt



Leistung versus Kosten bei Speichersystemen

(absolute Zahlen überholt, Verhältnisse aber heute ähnlich)



Zahlenbeispiel (veraltet)

- **Plattenlaufwerk Seagate Cheetah:**
  - 19036 Zylinder
  - 4 Spuren pro Zylinder
  - 947 Blöcke (Sektoren) pro Spur
  - 512 Byte pro Block
- **Daraus errechnen sich Kapazitäten von:**
  - 484.864 Byte (473,5 KB) pro Spur,
  - 1.939.456 Byte (1,85 MB) pro Zylinder und
  - 36.919.484.416 Byte (34,38 GB) pro Laufwerk
- **Drehzahl 10.000 UPM, also 166 Hz,**
  Dauer einer Umdrehung 6 ms
  - In einer Umdrehung max. die 473,5 KB einer Spur übertragen – theoretische **Datentransferrate** von 78.917 KB/s oder rund 77 MB/s
- **Positionierungszeit für den Zugriffskamm ("seek"-Zeit):**
  - durchschnittlich 6 ms, von Spur zu Spur 1,5 ms
- **Durchschnittliche Zugriffszeit: ~ 10 ms**

Optische Platten

- **CD-ROM**
  - Nur lesbar (wird "gedruckt")
  - 650 MB
  - Für DBS bisher nicht relevant
- **WORM**
  - Einmal schreibbar, dann nur noch lesbar ("write once, read many times")
  - 1 – 5 GB
  - Niedrigste Kosten pro Bit!
  - DBS: Protokolldaten, Versionen
- **Wiederbeschreibbare optische Platte**
  - Z.B. magneto-optische Platte
  - Wie alle optischen Platten relativ langsam (wahlfreier Zugriff 100 ms)
  - Eher Ersatz für Magnetband als für Magnetplatte

Zugriff auf Plattenspeicher (Dienste)

- **Typische Befehle an eine Plattensteuerung:**

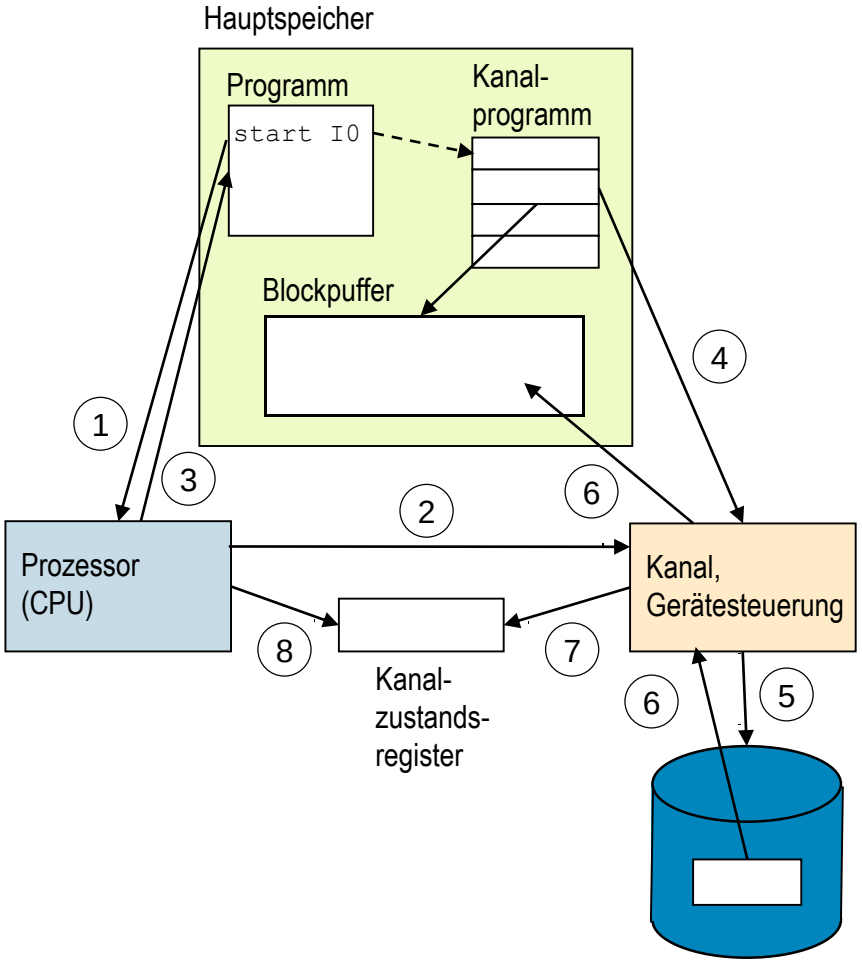
SEEK n	positionier Zugriffskamm auf Zylinder n
SET HEAD n	schalt Schreib-Lese-Kopf n ein
SECTOR ROTATE n	wart auf Anfang des Blocks n in der Spur
READ a	lies nächsten Block in den Hauptspeicher ab Adresse a
WRITE a	überschreib nächsten Block mit Hauptspeicherinhalt ab Adresse a
VERIFY a	vergleich nächsten Block mit Hauptspeicherinhalt ab Adresse a
SELECT g	Auswahl des Laufwerks g für die nächsten Befehle

Zugriff auf Plattenspeicher (2)

- **Folge zusammengehöriger Dienste**
  - zu Prozedur zusammenfassen
    - Z.B. Schreiben eines Blocks an eine bestimmte Stelle:
      Folge von **select, seek, set head, sector rotate, write, sector rotate** und **verify**
  - Abwicklung der Prozedur an *Spezialrechner* delegieren: **Kanal, Gerätesteuerung**
- **Minimalforderungen (notwendig für Persistenz):**
  - Erfolgreiche Schreiboperation hat tatsächlich Block an richtige Stelle geschrieben, ohne Nebenwirkungen
  - Bitfehler werden beim Lesen erkannt (z.B. Parität)
  - Alle Fehler werden korrekt diagnostiziert und gemeldet
    - Außerbetriebnahme, Fehlen des Datenträgers, mechanische Verklemmung, Schreibsicherung, Überlastung usw.
- **Ausführung der Operationen auch asynchron möglich:**
  - Abschluss der Operation (erfolgreich oder nicht)
    über Hardware-Register (Kanalzustandsregister) angezeigt

Kanal (Gerätesteuerung)

- **Lesen eines Blocks:**



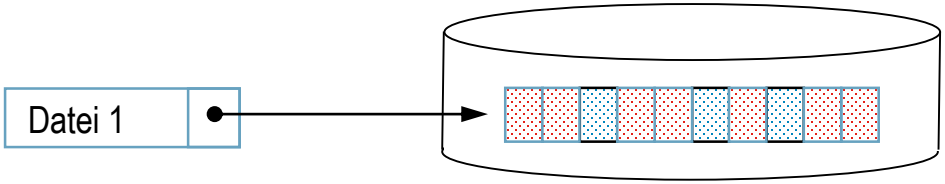
- **Schreiben eines Blocks**
  - analog
- **Lesen/Schreiben mehrerer Blöcke hintereinander**
  - mit einem Kanalprogramm möglich

2.A.2 Dateikatalog

- **Dateikatalog: Verwaltungsdatenstruktur**
  - zur Abbildung des Dateinamens auf eine Folge von Blöcken
  - Selbst mit auf der Platte abgelegt
    - Meist auf einer festen Position (Zylinder 0, Spur 0, Block 0 o.ä.)
  - Kann hierarchisch gegliedert sein (UNIX, VMS, MS-DOS)
  - Einstieg über den Dateinamen
    - Muss dann zu einer gegebenen Blocknummer die physische Slot-Adresse liefern können
- **Zusätzlich: Freispeicherverwaltung**
  - Unbenutzte Blöcke auf einer Platte (momentan keiner Datei zugeordnet)
  - Z.B. als Bitliste

Katalogeinträge

- **Möglichkeit A:**
  - Eintrag = (physische Slot-Adresse des ersten Blocks; Anzahl der Slots, die ab dieser Adresse belegt sind)

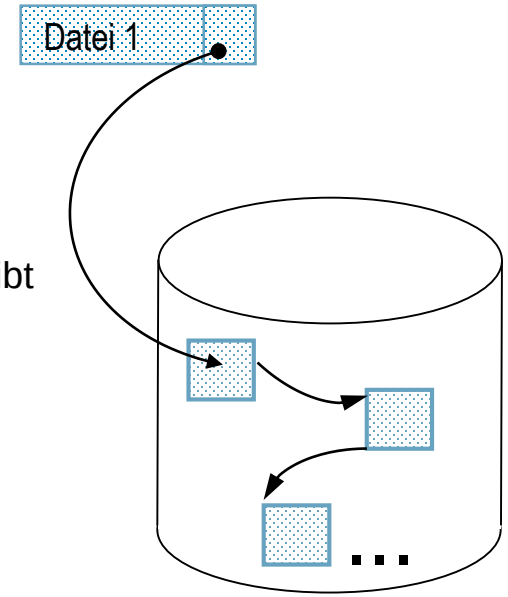


- Benötigte Zahl von Blöcken auf physisch sequenzielle und lückenlose Folge von Slots abbilden (ggf. über Spur- und Zylindergrenzen hinweg)

  + Kompakter Dateikatalog (Eintrag pro Datei sehr klein)
  + Slot-Adresse von Block i kann errechnet werden
  + Sequenzielles Lesen aller Blöcke mit minimalen Zugriffskammbewegungen
  – Dynamische Erweiterung der Datei kann Umkopieren der ganzen Datei in größeren zusammenhängenden Bereich erfordern oder unmöglich sein
  – "Zerklüftung" der Platte: nicht nutzbare kleine Stücke zwischen den Dateien

Katalogeinträge (2)

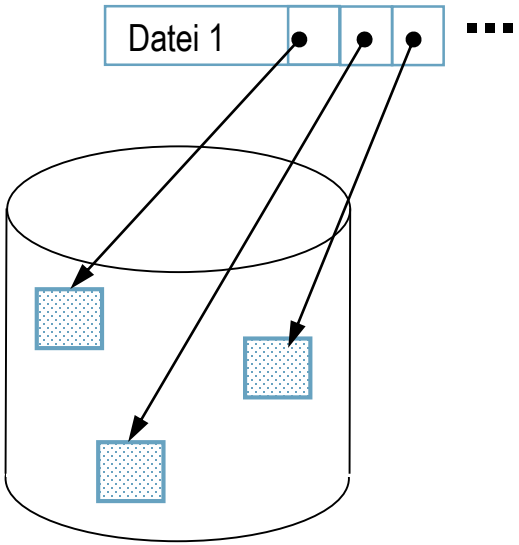
- **Möglichkeit B:**
  - Eintrag = (physische Slot-Adresse des ersten Blocks)
  - Verkettung der Blöcke als lineare Liste



    + Kompakter Dateikatalog
    + Erweiterung um neue Blöcke:
      immer möglich,
      wenn es auf der Platte irgendwo noch genug freie Slots gibt
    – Speicherplatz:
      in jedem Block muss Platz reserviert werden
      für Zeiger auf Nachfolger (Slot-Adresse, 6–8 Byte)
    – Sequenzielles Lesen der ganzen Datei:
      i.Allg. sehr langsam
      (viele Bewegungen des Zugriffskamms)
    – Zugriff auf Block i:
      erst nach Lesen der Blöcke 1 bis i–1 möglich
      (Kette entlanglaufen) – nicht akzeptabel!

Katalogeinträge (3)

- **Möglichkeit C:**
  - Eintrag = (Array mit den Slot-Adressen aller Blöcke)
  - Bekanntestes Beispiel: FAT



+ Zugriff auf Block i:
  problemlos
+ Erweiterung um neue Blöcke:
  immer möglich,
  wenn es auf der Platte irgendwo noch
  genug freie Slots gibt
– Sequenzielles Lesen aller Blöcke:
  langsam (viele Zugriffskammbewegungen)
– Katalog sehr groß,
  variabel lange Einträge, ändern sich bei jeder Erweiterung

Katalogeinträge (4)

- **Möglichkeit D:**
  - Eintrag = (Array von (Slot-Adresse; Zahl der von dieser Adresse an sequenziell belegten Slots))
    - 0 Katalog in der Regel deutlich kleiner als bei C, aber auch variabel lange Einträge
    - + Slot-Adresse von Block i berechenbar
    - + Erweiterung immer möglich, wenn irgendwo noch genügend freie Slots (ggf. in Teilstücken)
    - 0 Sequenzieller Zugriff um so besser, je weniger Teilstücke es gibt
    - – Kann zu C degenerieren
    - + Kann durch Reorganisation der Platte in A überführt werden (ohne Änderung der Anwendungsprogramme!)
  - **Extent**
    - Zusammenhängendes (physisch sequenzielles) Teilstück einer Datei
    - Typischerweise im Zuge einer Erweiterung belegt
    - Durch ein einzelnes Array-Element im Katalogeintrag repräsentiert
  - Katalogeintrag einer Datei: **Extent-Tabelle**

Vorlesung Implementierung von Datenbanksystemen

3. Sätze

Prof. Dr. Klaus Meyer-Wegener

Wintersemester 2019/20

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

3. Satzstrukturierte Dateien

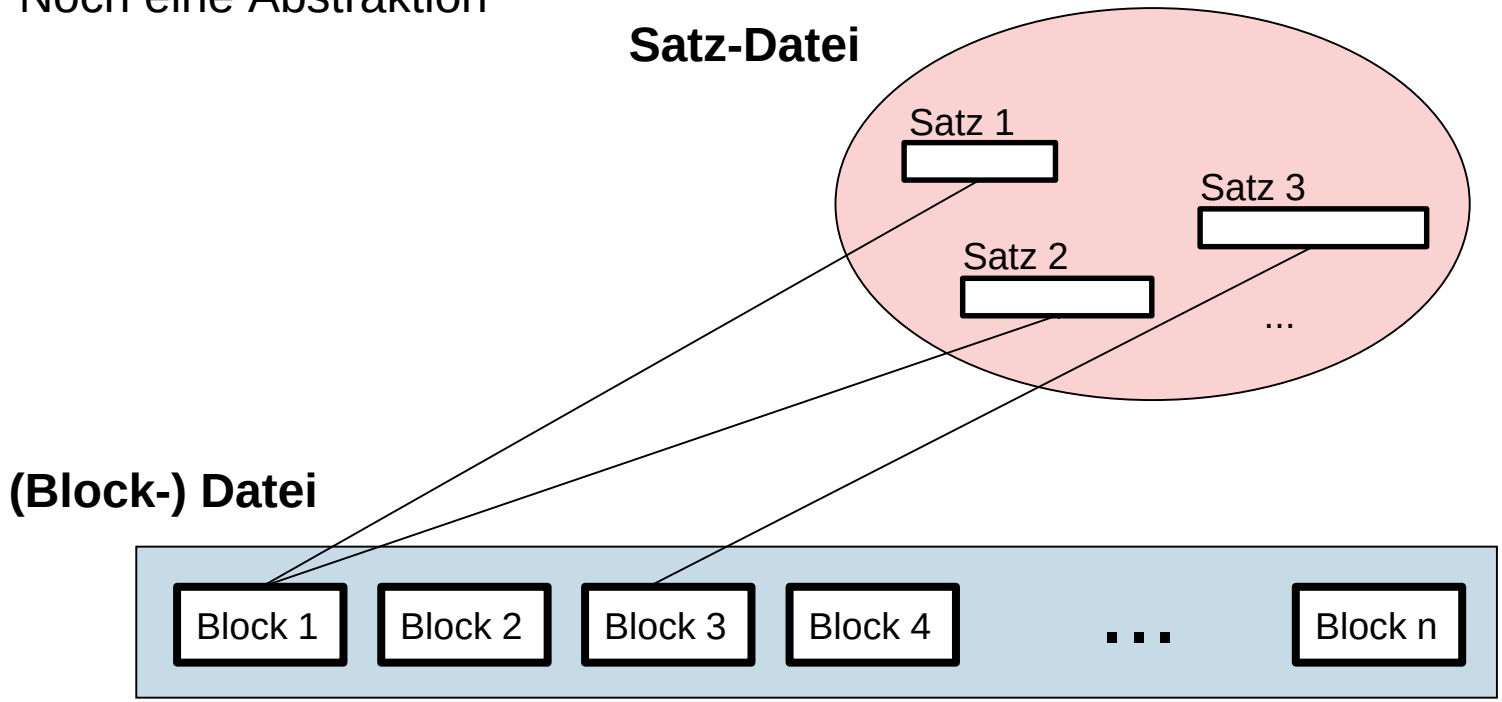
- **Block**
  - Einheit des Transports zwischen Hintergrund- und Hauptspeicher
- **Größe der Blöcke**
  - ist vom System vorgegeben (Formatierung des Hintergrundspeichers)
  - Typisch: 512 Bytes, 2K, 4K; manchmal in Grenzen wählbar
    - Werden tendenziell größer – dazu später mehr
  - **Kleine Blöcke**: viele E/A-Operationen
  - **Große Blöcke**: Mitlesen bzw. Mitschreiben uninteressanter Daten (die im selben Block abgelegt sind)
    oder Platzverschwendung (wenn man den Rest leer lässt)
- **Deshalb neue Abstraktion: Satz**
  - Neue Schicht oberhalb der Block-Dateien

Satz

- **Genau die Daten, die zu einem Gegenstand der Anwendung gehören**
  - Beispiele: Personenstammsatz, Teilestammsatz, Auftragssatz
- **Entscheidend: Größe (Länge) von der Anwendung bestimmt**
  - Und nicht (wie bei den Blöcken) vom System!
- **Insbesondere: variable Länge**
  - Von Satz zu Satz, und auch bei einem einzelnen Satz im Laufe der Zeit
  - Feste Länge als Spezialfall natürlich auch noch möglich
- **"Record" oder "Struktur" oder "Objekt" oder …**
  - In vielen Programmiersprachen
  - Dann aus Feldern zusammengesetzt (Bestandteile des Satzes); für Dateiverwaltung auf dieser Stufe aber noch irrelevant
- **Satz (wie Block) hier nur Folge von Bytes**
  - Keine weitere innere Struktur

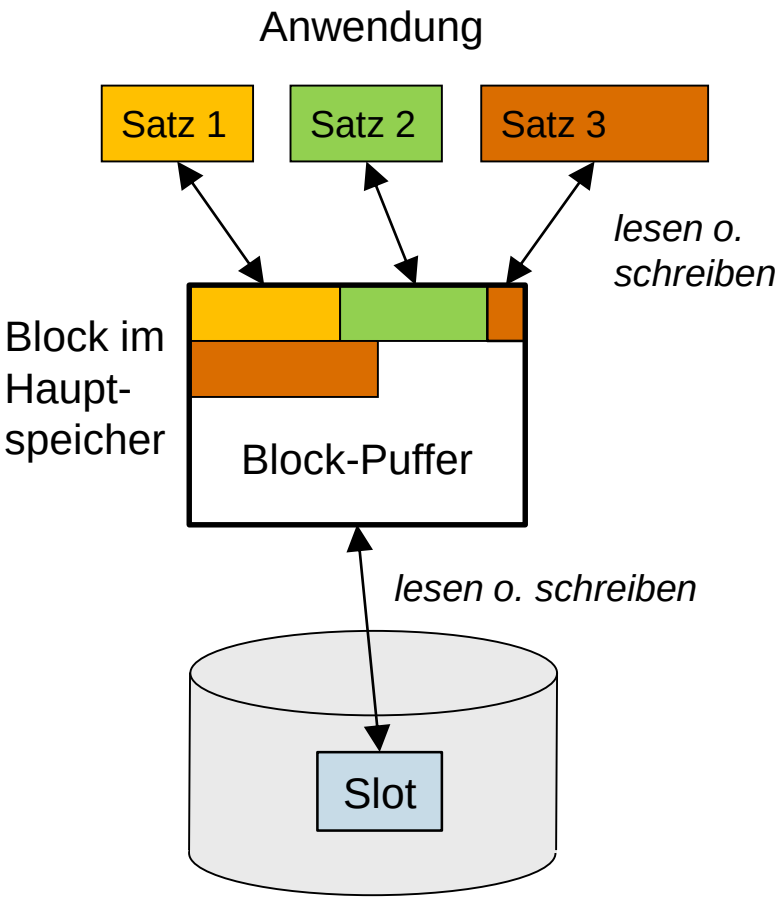
Satz-Datei

- **Menge (!) von Sätzen variabler Länge**
  - D.h. Reihenfolge unbestimmt
  - Noch eine Abstraktion



Sätze in Blöcken: Blockung

- **Typischerweise passen mehrere Sätze in einen Block.**
  - Satzlängen 100 – 1000 Bytes
  - Variable Anzahl von Sätzen pro Block
- **Vereinfachende Annahme:**
  - Keine blockübergreifenden Sätze ("spanned records")
  - D.h. jeder Satz vollständig in einem Block abgelegt
    - Ergänzung für längere Sätze aber kein großes Problem – siehe Übungen

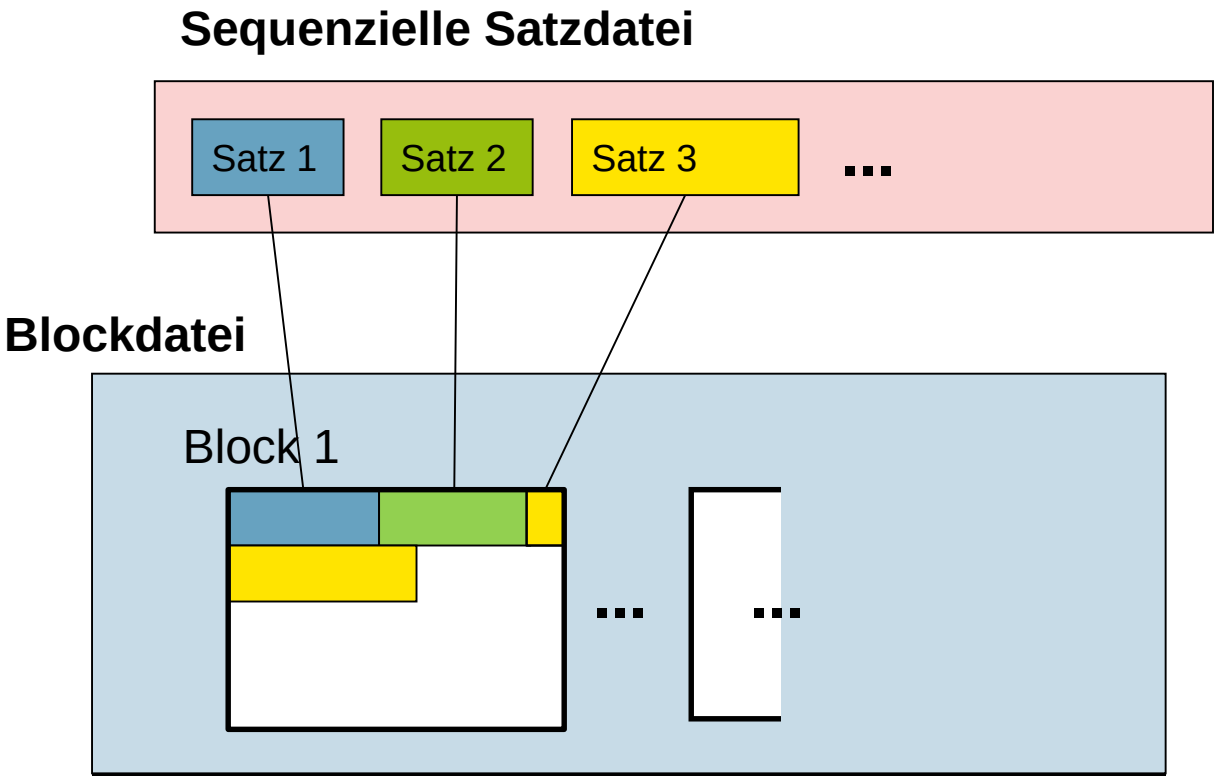


Blockung (2)

- **Feste Satzlänge:**
  - Als Sonderfall möglich – dann auch ausnutzen (kein Längenfeld)
  - Feste Zahl von Sätzen pro Block
    - Aus Blockgröße und Satzlänge errechenbar
  - Meist ungenutzter Speicherplatz am Ende eines Blocks
- **Variable Satzlänge:**
  - Zahl der Sätze ändert sich von Block zu Block
  - Erinnerung: Reihenfolge der Sätze beliebig (Menge!)
    - Man darf also z.B. umsortieren, bis nur noch wenig ungenutzter Speicherplatz am Ende eines Blocks übrig bleibt.
- **Was hier gewählt wurde ist dann eine Eigenschaft der Datei.**

3.1 Sequenzielle Satzdatei

- **Sonderfall: Folge (!) von Sätzen fester oder variabler Länge**



- Wird nur sequenziell geschrieben und gelesen – einfacher

Sequenzielle Satzdatei (2)

- **Gibt es tatsächlich**
  - Z.B. als Zugriffsmethode SAM (Sequential Access Method)
  - oder ESDS (Entry-Sequenced Data Set) in der Zugriffsmethode VSAM (Virtual Sequential Access Method) der IBM (z/OS)
- **Schreibreihenfolge**
  - = Abspeicherungsreihenfolge
  - = Lesereihenfolge
  - Vom Benutzer definiert
    - Das System erhält sie, interpretiert sie aber nicht.
      - (Und kontrolliert sie deshalb auch nicht, z.B. Sortierung)

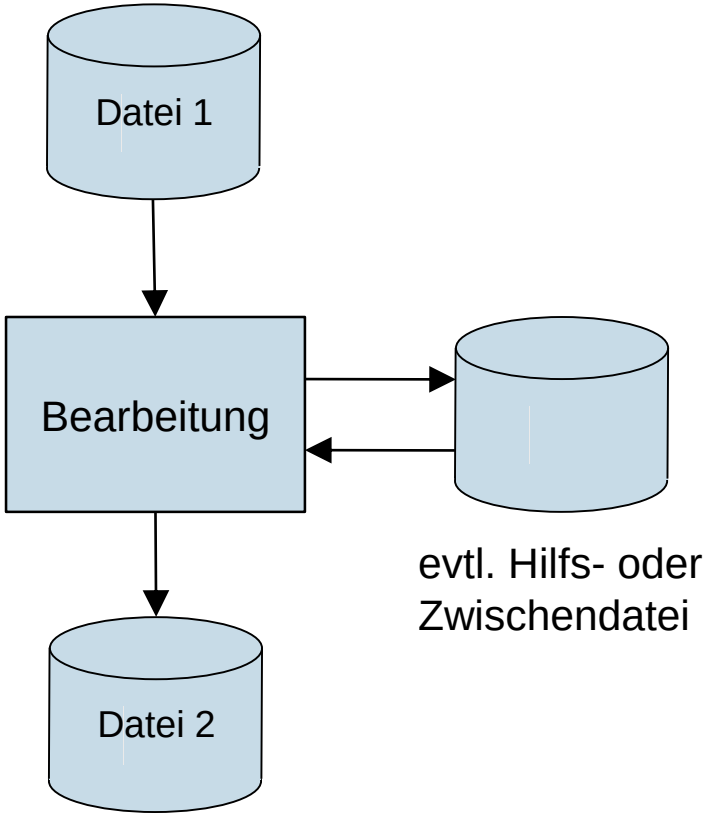
Sequenzielle Satzdatei (3)

- **Primäre Zugriffsarten:**
  - Sequenzielles Lesen (der ganzen Datei)
  - Sequenzielles (Über-) Schreiben der ganzen Datei
  - Sequenzielles Anhängen weiterer Sätze an eine Datei

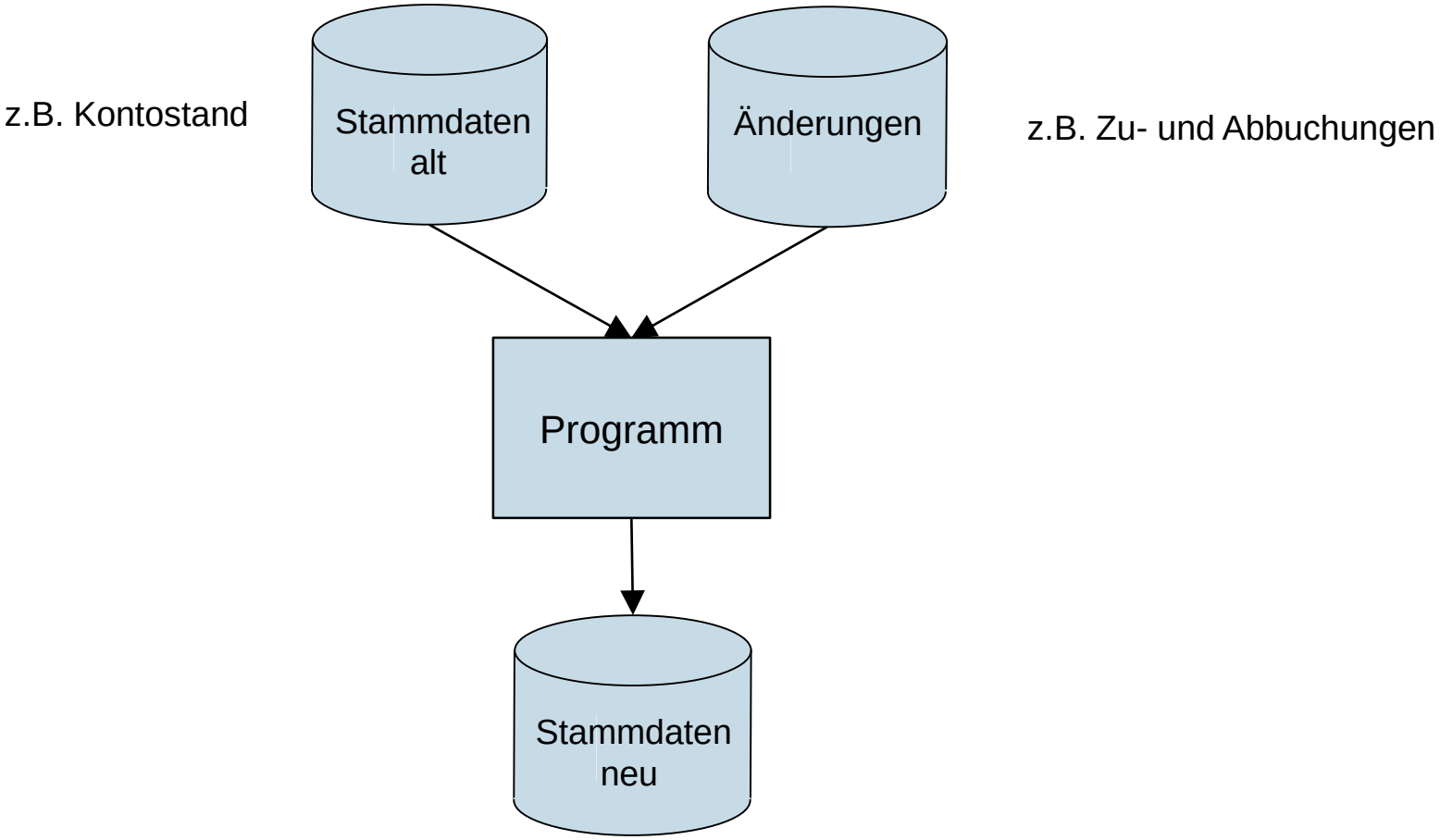
- **Insbesondere werden *nicht* unterstützt:**
  - Wahlfreies Lesen eines bestimmten Satzes
  - Einfügen eines Satzes
  - Ändern eines Satzes
    - I.Allg. mit Längenänderung verbunden

Stapelverarbeitung (Batch Processing)

- **Typische Bearbeitungsform für sequenzielle Dateien**
  - Insbesondere: **Sortierung**



Stapelverarbeitung (2)



Zugriffsoperationen (Dienste)

- **Datei öffnen:**

```
SeqRecordFile::SeqRecordFile (
  char *FileName,
  char Mode,
  int *RecLength )
```

- Dateiname je nach Betriebssystem
- Modus gibt an, wie die Datei geöffnet werden soll
  - Zulässige Werte: nur Lesen ("r"), Schreiben ("w"), Anhängen ("a")
- Satzlänge
  - beim lesenden Zugriff Ausgabeparameter
  - beim schreibenden Zugriff Eingabeparameter,
    0 heißt: variable Satzlänge

Zugriffsoperationen (2)

- **Nächsten Satz lesen:**

```
void SeqRecordFile::read-next (
   int *RecordLength,
   char *RecordBuffer )
```

  - Satzlänge als Eingabeparameter: maximale Länge,
    als Ausgabeparameter: tatsächliche Länge
    - Verhindert unbeabsichtigtes Überschreiben von Teilen des Arbeitsspeichers, wenn erwartete Länge < tatsächliche Länge
      - "Buffer Overflow"
  - Statt Satzpuffer kann auch **Zeiger** auf Satzanfang zurückgeliefert werden
    - Vorteil: kein Kopieren
    - Nachteil: Zeiger nur bis zum nächsten **read-next** benutzbar
  - Dateikontrollblock enthält auch **Leseposition** in der Datei
    - Wird jedes Mal weitergerückt, so dass nachfolgendes **read-next** den nächsten Satz liest (Dienstprotokoll)

Zugriffsoperationen (3)

- **Nächsten Satz schreiben:**

```
void SeqRecordFile::write-next (
   char *RecordBuffer,
   int RecordLength )
```

  - Nur beim Öffnungsmodus "Schreiben" oder "Anhängen" erlaubt
  - Schreibt den Satz an das Ende der Datei
  - Wurde die Datei mit fester Satzlänge geöffnet, so wird die Angabe einer Satzlänge ignoriert
  - Mögliche Fehler:
    - Satz zu lang
    - Datei war nur zum Lesen geöffnet
    - Kein Platz mehr auf dem Hintergrundspeicher
- **Schließen der Datei:**

```
SeqRecordFile::~SeqRecordFile ()
```

Blockunabhängigkeit

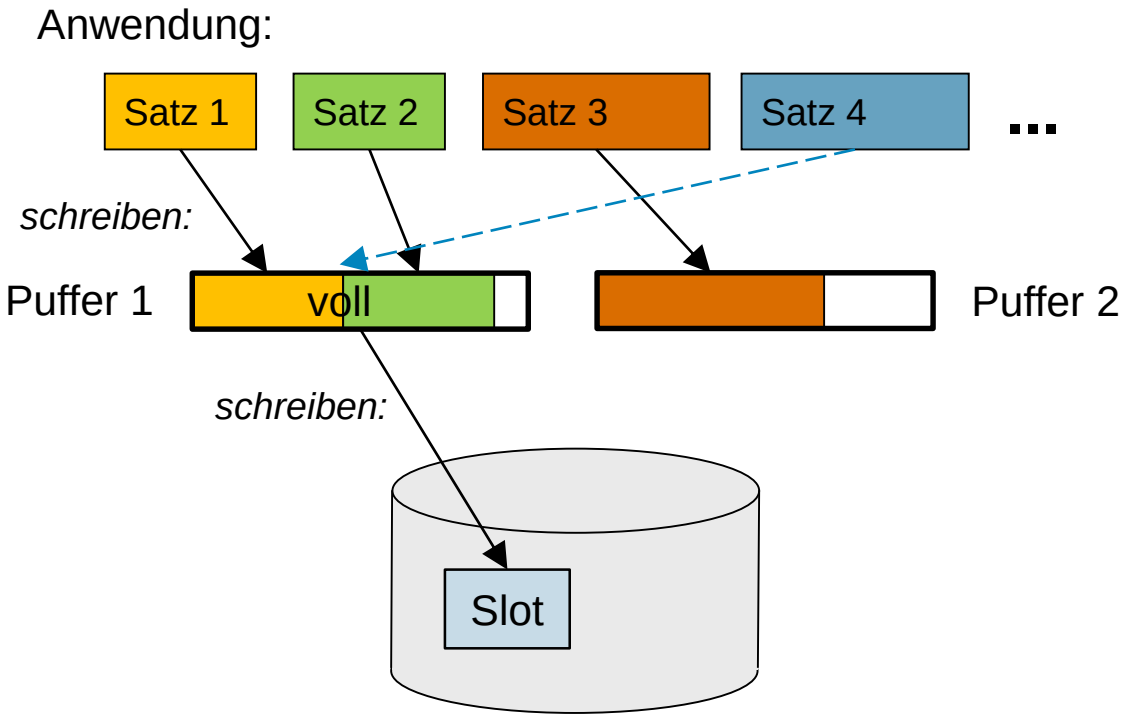
- **Anwendungsprogramm, das mit sequenzieller Datei arbeitet,**
  - kennt (und benutzt) die Einheit des physischen Transports zwischen Speichergerät und Hauptspeicher (= Block) nicht mehr.
    - Also kann diese verändert werden,
      ohne dass das Programm betroffen ist.
  - kann ebenso gut mit einem Magnetband arbeiten!
  - braucht nur zwei Operationen zu kennen
    - Deshalb z.B. beim Dateikonzept der Programmiersprache Pascal verwendet
  - kann einige Schreib- oder Leseoperationen ohne "physische" Ein-/Ausgabe ausführen (n – 1 bei n Sätzen im Block)
- **Nachteile:**
  - Kein Direktzugriff auf Sätze
  - Beim Schreiben werden immer nur einzelne Blöcke hinzugefügt (oder eine feste Zahl) – Gefahr der Zerstückelung der Datei!

Puffer

- **Block**
  - ist weiterhin die Einheit des physischen Transfers zwischen Hintergrund- und Hauptspeicher
- **Satz lesen:**
  - Immer einen ganzen Block vom Hintergrundspeicher lesen – enthält meist außer dem gewünschten noch andere Sätze
- **Satz schreiben:**
  - Block erst dann auf den Hintergrundspeicher schreiben, wenn er nach mehreren Satz-Schreiboperationen ganz mit Sätzen gefüllt ist
- **Sätze werden also gepuffert**
- **Dateipuffer**
  - Hauptspeicherbereich, der (mindestens) einen Block der Datei aufnehmen kann, damit Sätze herausgeholt oder hineingepackt werden können ("Blockpuffer")

Wechselpuffertechnik

- **Optimierungsmöglichkeit bei sequenziellen Dateien: Verwendung von *zwei* Blockpuffern**



Wechselpuffertechnik (2)

- **Beim Schreiben:**
  - Wenn Puffer 1 voll, **asynchron** auf Hintergrundspeicher schreiben und gleichzeitig Puffer 2 zur Aufnahme weiterer Sätze verwenden
  - Ist auch Puffer 2 voll, ebenfalls asynchron schreiben und mit Puffer 1 fortfahren
- **Beim Lesen analog:**
  - Während Sätze in Puffer 1 nach und nach gelesen werden, *gleichzeitig* schon nächsten Block vom Hintergrundspeicher in Puffer 2 lesen
    - Wird auch "Prefetching" genannt
  - Der ist dann (idealerweise) bereits da, wenn sein erster Satz gebraucht wird.
  - Übernächsten Block wieder in Puffer 1 lesen usw.
- **Rein interne Optimierung:**
  - *Unsichtbar* für Programmierer (bzw. höhere Schichten)
  - Möglich geworden durch "Kapselung" des Blocktransfers

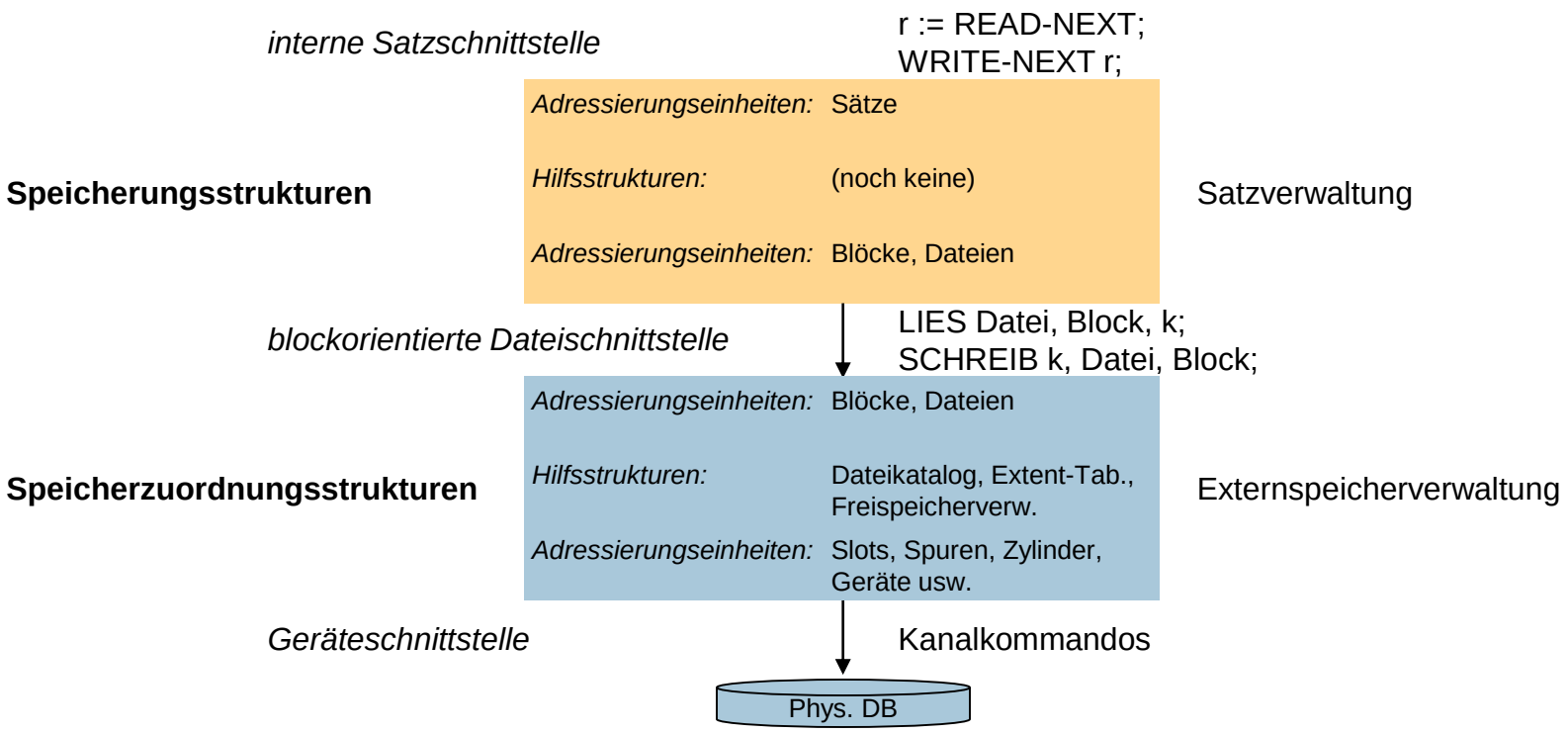
Allgemeiner Dateipuffer

- **Verallgemeinerung:**
  - Nicht nur zwei Blockpuffer, sondern beliebig viele
  - Dann auch **Pufferrahmen** – engl. buffer frames – oder **Kacheln** genannt
  - Möglich wegen zunehmend größerer Hauptspeicher
- **Beim Lesen**
  - die nächsten *n* Blöcke auf einmal lesen (Prefetching)
- **Beim Schreiben**
  - mehrere Blöcke füllen, dann auf einmal ausschreiben
- **Noch günstiger als Wechselpuffertechnik:**
  - Spart Zugriffsarmbewegungen
  - Größerer Nutzen vor allem bei nicht-sequenziellen Dateien (siehe unten)
  - Aber: Persistenz wird (zunächst) ein Problem – nach Schreiben eines Satzes steht der erst einmal nur im (flüchtigen) Puffer …

Sequenzielle Satzdatei – Fazit

- **So primitiv sie auch sein mag**
- **Zeigt sie doch schon einiges an Kapselung und Optimierungspotenzial:**
  - **Blockunabhängigkeit:**
    - Blöcke als Einheit des Transfers zwischen Hintergrund- und Hauptspeicher sind nicht mehr sichtbar;
    - könnten also durch Reorganisation (Formatierung) verändert werden.
  - **Pufferung:**
    - Mehr Speicherplatz bereitstellen für eine bessere Leistung
    - Blöcke mit Sätzen füllen oder Sätze aus Blöcken holen und *gleichzeitig* andere Blöcke auf den Hintergrundspeicher schreiben bzw. von ihm lesen
    - Völlig unsichtbar für Benutzer der Schnittstelle, rein interne Optimierung

Resultierende Schicht



3.2 Direktzugriff auf Sätze

- **Flexiblerer Ansatz als bei sequenziellen Dateien gesucht:**
  - Auch Einfügen, Löschen und Ändern *einzelner Sätze* soll möglich sein!
- **Generelle Frage:**
  - In welchem Block einen neuen Satz **ablegen**,
    und wie später diesen Satz **wiederfinden**,
    auch wenn zwischenzeitlich etliche andere Sätze
    gelöscht und eingefügt wurden?
- **Annahmen:**
  - Variable Satzlänge (allgemeiner Fall)
  - Reihenfolge der Abspeicherung
    muss nicht mehr Reihenfolge des Einfügens sein.
    - Information über Einfügereihenfolge darf verlorengehen.
    - Datei ist jetzt wirklich eine *Menge* von Sätzen.
  - Direkter Zugriff auf einzelne Sätze über ihre **Satzadresse**

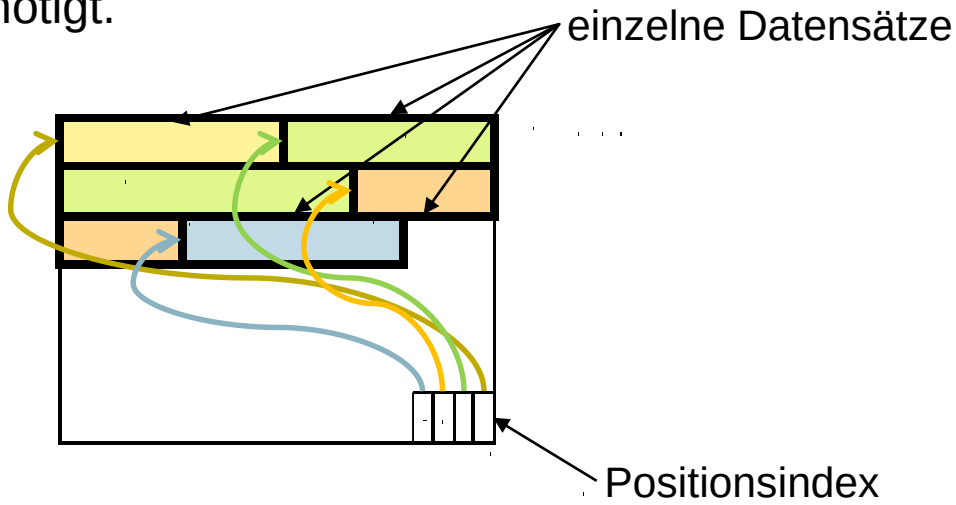
Satzadresse

- **Beim Einfügen eines Satzes vom DBVS zugeteilt ("Handle")**
- **Später zum Zugriff auf denselben Satz verwenden**

- **Zwei grundlegende Eigenschaften:**
  - **eindeutig**
  - **unveränderlich**
    - Auch bei beliebigen Einfügungen, Änderungen und Löschungen anderer Sätze
    - Auch dann, wenn der Satz verschoben werden muss, innerhalb des Blocks oder in anderen Block
      - Z.B. weil er länger geworden ist

TID-Konzept

- **Verbreitetste Realisierung einer Satzadresse**
- **Satzadressierung über Indirektion innerhalb der Blöcke**
  - Hilfsstruktur: Feld (Array) mit Anfangsadressen (Byte-Positionen) aller Sätze dieses Blocks (sog. "Positionsindex")
  - Satzadresse ist ein Paar aus Blocknummer und Index in diesem Feld
  - ("**TID** = Tuple IDentifier")
  - Für den Zugriff auf einen Satz wird nur ein Blockzugriff benötigt.
  - Struktur eines Blocks:

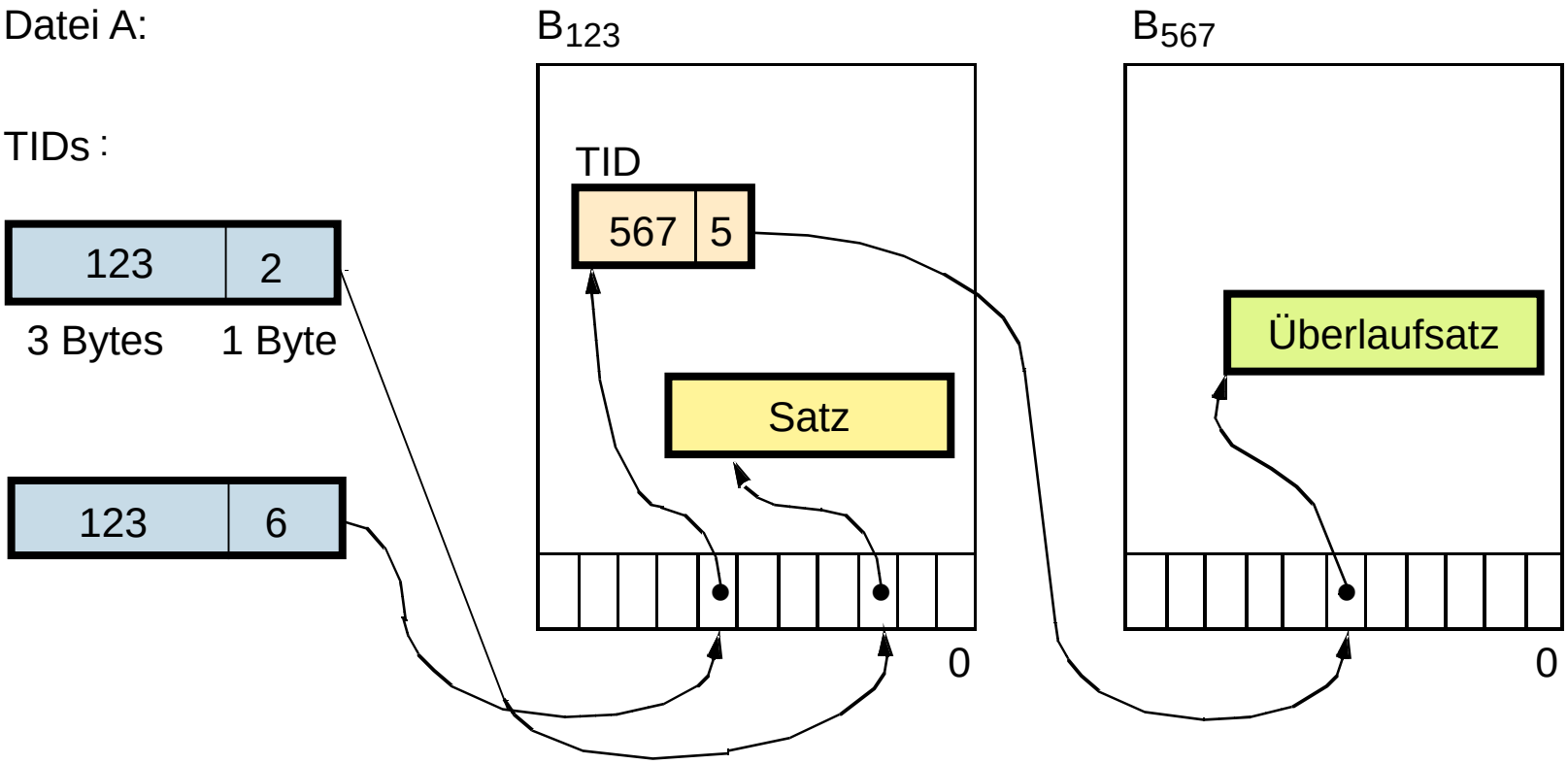


TID-Konzept (2)

- **Löschen eines Satzes**
  - Der entsprechende Eintrag im Positionsindex wird als ungültig gekennzeichnet.
  - Alle anderen Sätze im selben Block können verschoben werden, um den freien Platz zu maximieren –
    es ändern sich nur ihre Anfangsadressen im Positionsindex.
  - Alle Satzadressen bleiben stabil.
- **Änderungsoperation auf einem Satz**
  - Es kann sich die Länge des Satzes ändern !!!
  - Satz schrumpft oder wird größer (ohne Überlauf):
    - Alle Sätze werden innerhalb des Blocks verschoben und der Positionsindex wird angepasst.
  - Satz wird größer und der freie Platz im Block reicht nicht mehr für die Speicherung des jetzt größeren Satzes (Überlauf):
    - Verschieben des Satzes in einen anderen Block!

Überlaufbehandlung beim TID-Konzept

- **Verschieben eines Satzes**



Überlaufbehandlung beim TID-Konzept (2)

- **Vorgehen**
  - Im alten Block verbleibt an der Stelle des Satzes eine neue Satzadresse, die auf den neuen Block verweist.
  - In diesem (seltenen) Fall müssen also zwei Blöcke gelesen werden, um einen Satz über seine TID zu finden.
  - Wird der Satz ein weiteres Mal verlagert, so wird die (Weiterleitungs-) Satzadresse im ersten Block verändert. Dadurch bleibt es bei maximal einer Indirektion.
- **Trick**
  - Die Länge der Überlaufkette ist also immer kleiner oder gleich 1, d.h. ein Überlaufsatz darf nicht weiter "überlaufen", sondern wird von seiner Hausadresse aus neu platziert.
- **Vorteile**
  - Ein Satz kann innerhalb eines Blocks und über Blockgrenzen hinweg verschoben werden, ohne dass die TID sich ändert.

3.3 Direkte Satzdateien

- **Mit Hilfe der Satzadressen (TID's) sind einzelne Sätze beliebig zugreifbar, änderbar und löschbar**
- **Zugriffsoperationen:**

```
DirectRecordFile::DirectRecordFile ( ... )
```

  - Wie gehabt, nur ist im Schreibmodus jetzt auch Lesen möglich

```
RecordAddress DirectRecordFile::insert
   ( char *RecordBuffer, int RecordLength )
void DirectRecordFile::read ( RecordAddress Address,
   char *RecordBuffer, int *RecordLength )
void DirectRecordFile::modify ( RecordAddress Address,
   char *RecordBuffer, int RecordLength )
void DirectRecordFile::delete ( RecordAddress Address )
```

  - (Bei NoSQL-Systemen auch "CRUD" genannt: create, read, update, delete)

Freispeicherverwaltung

- **Z.B. in jedem Block am Anfang**
  - Zahl der freien Bytes
  - Suche bei **insert** (und evtl. **modify**) dann aber zu mühsam: alle Blöcke abklappern
- **Alternative:**
  - *k* Blöcke am Anfang der Datei mit Feld "Freispeichertabelle" (neuer Blocktyp)
  - in diesen Blöcken (hinter dem Header) nur Einträge des Typs vorzeichenlose Festpunktzahl, für leere Blöcke mit (Blockgröße – Header-Größe) initialisiert
  - Eintrag *i* der Freispeichertabelle stimmt mit Eintrag "freie Bytes" im Header des *i*-ten Blocks überein (falls es den überhaupt noch gibt)
  - Suche nach Block mit freiem Platz viel schneller
  - Aber: beim Speichern und Löschen immer auch die Freispeichertabelle ändern – mindestens zwei Blöcke ändern (und zurückschreiben)
  - Erweitern der Datei um neue Blöcke schwieriger (warum?)

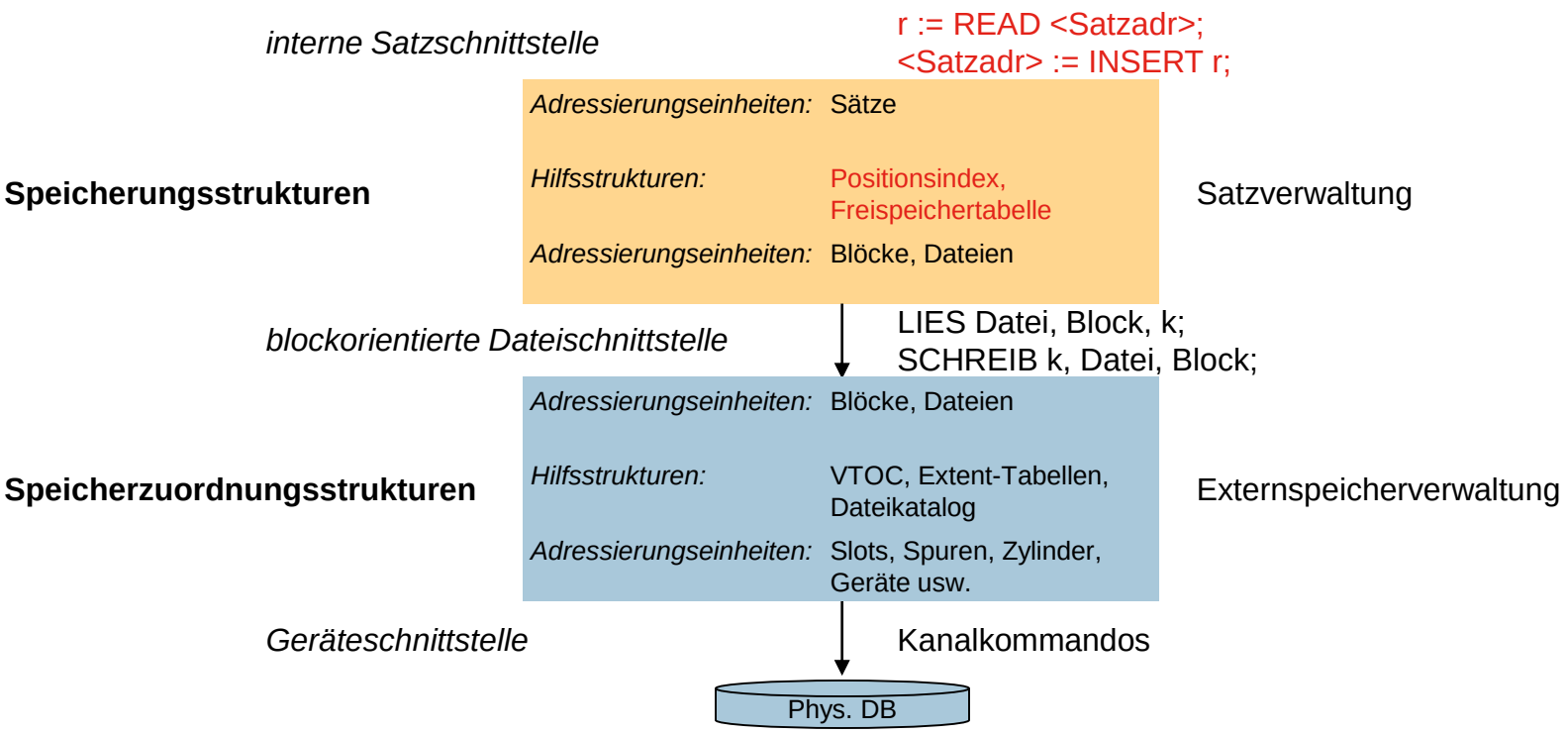
Weitere Operationen

- **Sequenzielles Lesen aller Sätze auch möglich:**

```
void DirectRecordFile::read-first
   ( char *RecordBuffer, int *RecordLength )
void DirectRecordFile::read-next
   ( char *RecordBuffer, int *RecordLength )
```

- **Aber:**
  - System entscheidet über Abspeicherungsreihenfolge, nicht Benutzer!
  - Man bekommt alle Sätze – in irgendeiner Reihenfolge (heute so, morgen evtl. anders).
  - System verwaltet die Menge der Sätze und keine Folge
- **Primär eben: Einzelsatzverarbeitung**
  - Erster Schritt von Stapelverarbeitung zu interaktiver Verarbeitung

Resultierende Schicht



Anhang

- **Nicht mehr verwendete Folien**
- **Zur Vertiefung einzelner Aspekte oder zum Nachschlagen bei Bedarf**

3.A.1 Satzadresse

- **Erste Idee: laufende Nummer des Satzes**
  - Kein zusätzlicher Speicherplatzbedarf
  - Zugriff auf Satz *i* durch Abzählen von vorn (direkter Zugriff???)
  - *Instabil!*
    ändert sich bei Einfügungen und Löschungen
    sowie bei Änderungen in der Abspeicherungsreihenfolge
- **Zweite Idee: Blocknummer und Byte-Position (Offset) im Block**
  - Kein zusätzlicher Speicherplatzbedarf
  - Direktzugriff auf Satz
  - *Instabil!*
    ändert Satz seine Länge, müssen i.Allg. andere Sätze im Block verschoben werden; wird Satz zu lang für Block, so muss er in anderen Block verlegt werden

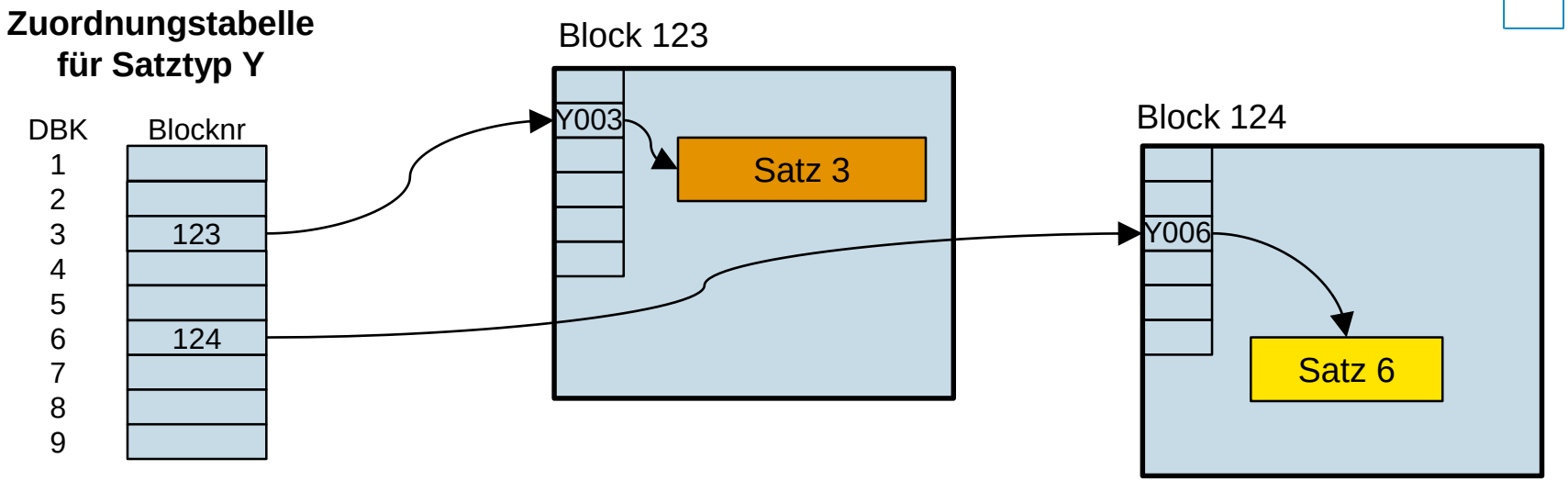
Zuordnungstabelle

- **Indirektion**
- **Verwaltung eines Feldes (in den ersten *k* Blöcken der Datei), das zu jeder Satznummer (Feldindex) Blocknummer und Byte-Position enthält**
  - Bei Einfügungen grundsätzlich neue Satznummer vergeben
  - Bei Löschungen Eintrag der entsprechenden Satznummer als ungültig kennzeichnen
  - **Stabil**: bei Verschiebungen im Block oder Verlagerungen in anderen Block nur Eintrag des Satzes im Feld ändern; Satznummer bleibt unverändert – Satz also über dieselbe Satznummer weiterhin auffindbar
  - Zugriff auf Satz erfordert nun zwei Blockzugriffe:
    einen für Feld, einen für Block mit dem Satz selbst
  - Bei Verschieben eines Satzes muss auch Eintrag im Feld geändert und wieder auf Hintergrundspeicher geschrieben werden (Persistenz) – zusätzliche E/A-Operation
  - Feld selbst beansprucht Speicherplatz
  - Auch bekannt als **Database Key Translation Table** (DBTT)

Zuordnungstabelle (2)

- **Optimierung:**
  möglichst keine Änderung der Zuordnungstabelle,
  wenn Satz innerhalb eines Blocks verschoben wird
  - Zusätzlich Speicherplatz im Block bereitstellen
    für kleine Zuordnungstabelle der Sätze in diesem Block
    - Enthält Satznummer und Anfangsadresse jedes Satzes
    - Ähnlich wie bei TID
  - In der großen Zuordnungstabelle dann nur noch Blocknummer
  - Wenn Satz innerhalb eines Blocks verschoben wird:
    - Neue Anfangsadresse in die kleine Zuordnungstabelle in diesem Block eintragen
    - Nur ein Block geändert, nur ein Block zurückzuschreiben auf Hintergrundspeicher

Zuordnungstabelle (3)



- **Database Key (DBK) ist "nicht sprechende" Adresse (wie Telefonnummer)**
  - wird gebildet aus Satztypbezeichnung r und Folgenummer f;
  - identifiziert Satz während seiner Lebenszeit in der Datei.
- **Problem: Wo wird die Zuordnungstabelle abgespeichert,**
  - Am Anfang? Wie erweitern?
  - Am Ende? Wie den Datenbereich davor erweitern?
  - In einer eigenen Datei?

3.A.2 Verkettung von Sätzen

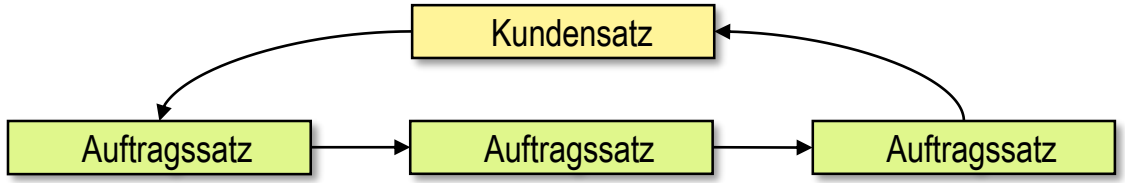
- **Wie und wo bewahrt man Satzadressen auf?**
- **Z.B. in anderen Sätzen**
  - Satzadressen als Verweise (Referenzen, Zeiger, Pointer) auf andere Sätze verwenden
- **Einfachstes Beispiel: lineare Liste der Sätze**
  - Next-Pointer
  - Anwenderdefinierte Ordnung nach irgendeinem Kriterium – unabhängig von der Speicherungsreihenfolge
- **Aber z.B. auch: Gruppenbildung**
  - Sätze mit bestimmtem Merkmal verketten (z.B. alle Autos eines Typs)
  - Evtl. mehrdimensional: ein Satz in diversen Ketten
- **Einfügen und Löschen**
  - wie bei verketteten Listen sonst auch

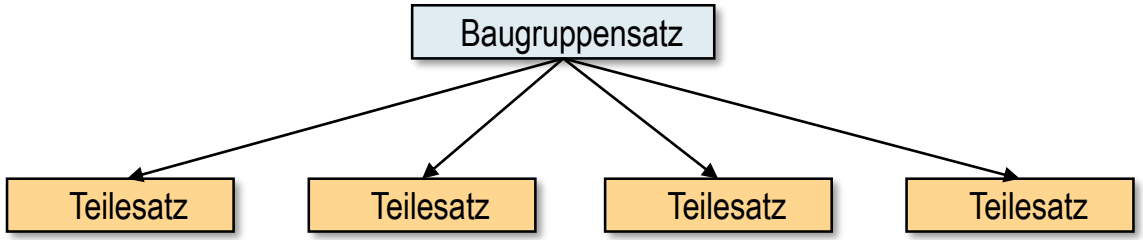
Verkettung von Sätzen (2)

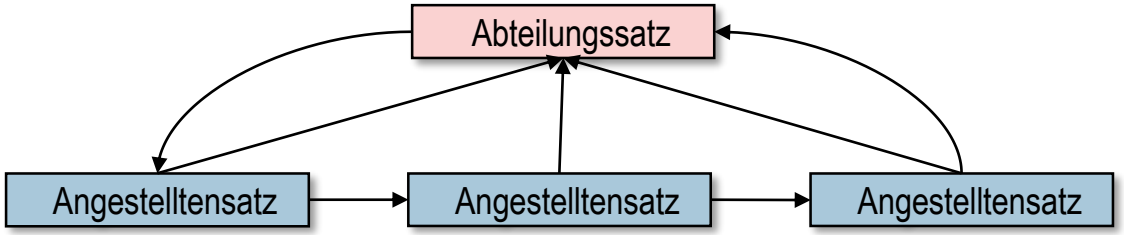
- **kombiniert benutzerdefinierte Ordnung (unabhängig von Speicherungsreihenfolge) mit der Möglichkeit des Direktzugriffs auf Sätze**
  - Sehr flexibler Mechanismus
  - Beliebige Beziehungen der Anwendungswelt darstellbar
  - "Navigation" in den Satzmengen
  - Auch zwischen Sätzen verschiedener Dateien

Verkettung von Sätzen (3)

- **Beispiele:**







Vorlesung Implementierung von Datenbanksystemen

4. Schlüsselzugriff – Teil 1

Prof. Dr. Klaus Meyer-Wegener

Wintersemester 2019/20

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

4. Satzzugriff über Schlüsselwerte

- **Satzadressen haben nichts mit Anwendungsdaten zu tun.**
  - Vom System vergeben, künstlich (wie Telefonnummer)
- **Ziel statt dessen:**
  - Über **Inhalt** (bestimmte Felder) eines Satzes zugreifen zu können
    - Z.B. Kundennummer, Name, Wohnort
  - Auch "assoziativer" Zugriff genannt
- **Diese Felder nennt man (Such-) Schlüssel.**
- **Besonders sinnvoll in interaktiven Anwendungen:**
  - Für den Schlüssel wird ein Wert vorgegeben
    - Z.B. vom GUI-Formular eingelesen
  - Datenhaltungssystem (Zugriffsmethode)
    soll den Satz der Datei liefern,
    der diesen Schlüsselwert enthält
    - Eigentlich alle Sätze, die ihn enthalten

Schlüsselzugriff

- **Bisher nicht unterstützt**
  - Sequenzielle und direkte Satzdatei können das nicht.
- **Einzige Möglichkeit:**
  - Alle Sätze der Reihe nach geben lassen (sog. **Scan**)
    und dann selbst auf gewünschten Schlüsselwert abprüfen – ineffizient!
    - Wird deshalb auch nicht als Operation angeboten.
- **Also: neue Hilfsstrukturen erforderlich**
  - Zugriff über bestimmten Schlüssel
    muss im voraus geplant
    und vom System speziell unterstützt werden
  - D.h. Sätze gleich so abspeichern,
    dass man sie gut wiederfinden kann
    - Macht ein bisschen mehr Arbeit

4.1 Gestreute Speicherung (Hashing)

- **Ziel:**
  - **Berechnung der Speicherposition** (Adresse) aus dem Schlüsselwert
  - Dann direktes Lesen des Blocks, der den gesuchten Satz enthält
    - Eine einzige E/A-Operation!
- **Umsetzung:**
  - Berechnung der Blocknummer genügt
    - Suchaufwand innerhalb des Blocks vernachlässigbar
- **(Fast) keine Hilfsstrukturen erforderlich**
  - Nur eine geeignete Berechnungsfunktion
- **Voraussetzungen:**
  - Maximale Anzahl der Sätze abschätzbar → zulässige Blocknummern
  - Hinreichend viele Blöcke für alle Sätze von Anfang an bereitstellen
    - Mit etwas "Luft"
      - Z.B. bei durchschnittlich 10 Sätzen pro Block und 10.000 Sätzen: 1.100 Blöcke bereitstellen

Gestreute Speicherung (2)

- **Blöcke, die mit gestreuter Speicherung gefüllt werden:**
  - Bildhaft auch "**Buckets**" (Eimer) oder "Bins" (Kästen) genannt
  - Spezielle Art von Blöcken
    - Neben denen für Sätze mit Zugriff über Satzadresse und denen für Freispeicherverwaltung
      - Es wird noch mehr Arten geben …

- **Vergabe einer Satzadresse**
  - weiterhin möglich, aber umständlich
  - Sollte ersetzt werden durch eindeutigen Schlüssel
    - Siehe unten

Hash-Funktion

- **Aufgabe: Umrechnung Schlüsselwert in Blocknummer**
- **Ziel dabei:**
  - Möglichst gleichmäßige Verteilung der Sätze auf die Buckets
- **Kollisionen**
  - Gleiche Blocknummer bei verschiedenen Schlüsselwerten
  - Bis zu einem gewissen Grad sogar erwünscht:
    Es passen ja mehrere Sätze in ein Bucket.
- **Auswahl der geeigneten Hash-Funktion**
  - hängt vom Typ des Schlüssels und der Verteilung der Schlüsselwerte ab
  - Je genauer die Verteilung der Schlüsselwerte bekannt ist,
    desto besser kann die Hash-Funktion darauf abgestimmt werden.
  - Es konnte (mathematisch) bewiesen werden:
    Wenn nichts bekannt ist, ist das **Divisions-Rest-Verfahren** am besten:
    - **h(k) = (k modulo q)**
    - mit k Schlüsselwert (ggf. in Zahl umgerechnet oder als Zahl interpretiert),
      q Anzahl der Buckets und h(k) errechnete Blocknummer (0 bis q–1)

Überlaufbehandlung

- **Völlig gleichmäßige Verteilung gelingt selten**
  - D.h. einige Buckets bekommen mehr Sätze ab, andere weniger.
- **Bucket kann "überlaufen":**
  - Zu klein für alle Sätze, die ihm durch die Hash-Funktion zugeteilt werden

- **Erste Lösung: Open Addressing**
  - Ausweichen auf Nachbar-Buckets in festgelegter Reihenfolge
    - \+ Kein zusätzlicher Speicherplatz benötigt
    - – Beim Suchen findet man in einem Bucket nicht nur Treffer, sondern auch "Überläufer".
    - – Beim Löschen muss man Überläufer "zurückholen".
    - – In den Nachbar-Buckets werden u.U. zusätzliche Überläufe erzeugt.

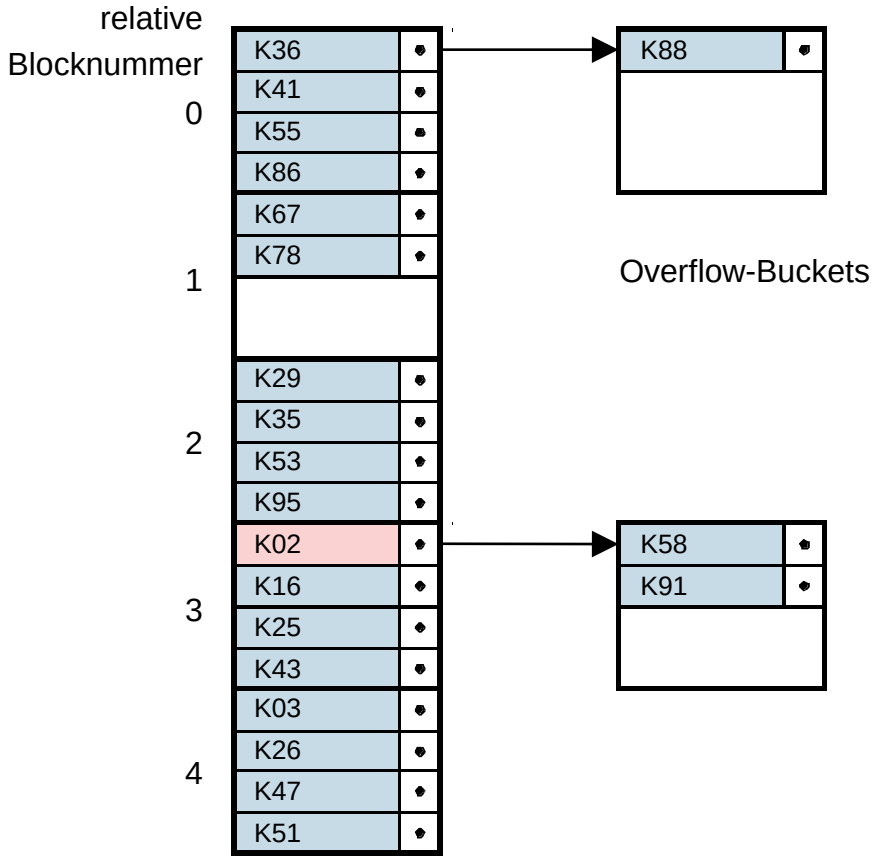
Überlaufbehandlung (2)

- **Zweite Lösung: Overflow-Buckets (mit "Separate Chaining"):**
  - Anlegen spezieller zusätzlicher Überlauf-Buckets
    - Mit Verkettung: Primär-Bucket zeigt auf sein Überlauf-Bucket
      - Pointer = Blocknummer
  - Und zwar für jedes übergelaufene Bucket separat
    - – Speicherplatzbedarf
    - + Keine Mischung von Sätzen
    - + Keine Beeinträchtigung der Nachbar-Buckets

- **Nachteil in beiden Fällen:**
  - Zwei Blockzugriffe (sonst nur einer)

Überlaufbehandlung (3)

- **Hash-Berechnung für Schlüsselwert K02**



$$
\begin{array}{r}
1101\ 0010 \\
\oplus\ 1111\ 0000 \\
\oplus\ 1111\ 0010 \\
\hline
1101\ 0000 = 208_{10}
\end{array}
$$

208 mod 5 = 3

Hash-Funktion:
EBCDIC-Darstellung jedes Zeichens
als Bitkette betrachten,
in 8-Bit-Stücken mit EXOR verknüpfen
("falten")
und Ergebnis als Zahl interpretieren

Anschließend Divisions-Rest-Verfahren

Zugriffsoperationen (Dienste)

- **Hash-Funktion nur intern verfügbar!**
  - Ausgewählt vom Systemadministrator, nicht vom Benutzer
  - D.h. Administrator kann den Datenbestand reorganisieren:
    - Neue Hash-Funktion auswählen, Zahl der Buckets erhöhen und alle Datensätze erneut abspeichern ("Rehash"),

    ohne dass die Anwendungen betroffen sind.
  - Operationen nehmen keinen Bezug auf Hash-Funktion.
- **Abspeichern eines Satzes:**

```
void HashedRecordFile::insert ( char *RecordBuffer,
        int RecordLength, char *KeyValue )
```

- **Auffinden eines Satzes über Schlüsselwert:**

```
char *HashedRecordFile::read ( char *KeyValue,
        int *RecordLength )
```

Zugriffsoperationen (2)

- **Operationen modify, delete, read-first und read-next**
  - wie bei direkten satzstrukturierten Dateien,
    nur mit Schlüsselwert statt Satzadresse
  - Satz und Schlüssel jeweils für sich verwaltet,
    daher auch eigene Änderungsoperation:

```
void HashedRecordFile::modify-key ( char *OldKeyValue,
  char *NewKeyValue )
```

- **Wenn Schlüssel *nicht* eindeutig: Sonderbehandlung erforderlich**
  - Gefundene Sätze einzeln abrufen
    - **next** o.ä. mit Leseposition ("Cursor")
  - Anhand anderer Felder den richtigen Satz (zum Ändern oder Löschen) identifizieren
  - Änderungen dann auf diesen Satz beziehen (aktuelle Leseposition)

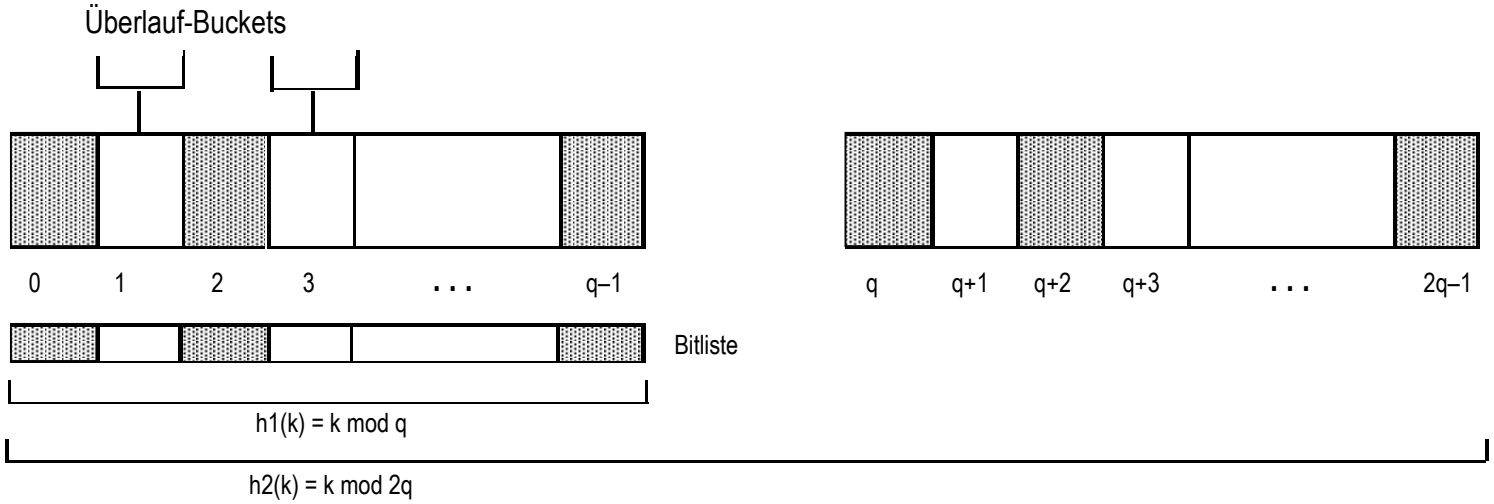
Bewertung

- **Schneller Zugriff über Schlüssel (1 bis 2 Blockzugriffe)**
  - Entscheidender Vorteil
  - Schafft keine andere Schlüssel-Zugriffsmethode
- **Sätze werden durcheinandergewürfelt**
  - Auch nicht nach Schlüsselwert geordnet
- **Speicherplatz muss im voraus belegt werden!**
  - Macht man ihn vorsichtshalber zu groß: Verschwendung
  - Macht man ihn zu klein: aufwändige Reorganisation
    ("offline" – Datei muss vorübergehend aus dem Verkehr gezogen werden)
    - Anmerkung: Es gibt "dynamische" Hash-Verfahren, die Erweiterung um Buckets wie auch Freigeben von Buckets zulassen ohne Reorganisation, siehe unten.
- **Gestreute Speicherung kann nur nach *einem* Schlüssel erfolgen!**
  - Bei Entscheidung z.B. für den Nachnamen
    muss nach Geburtsdatum, Wohnort usw. sequenziell gesucht werden

Virtuelles Hashing

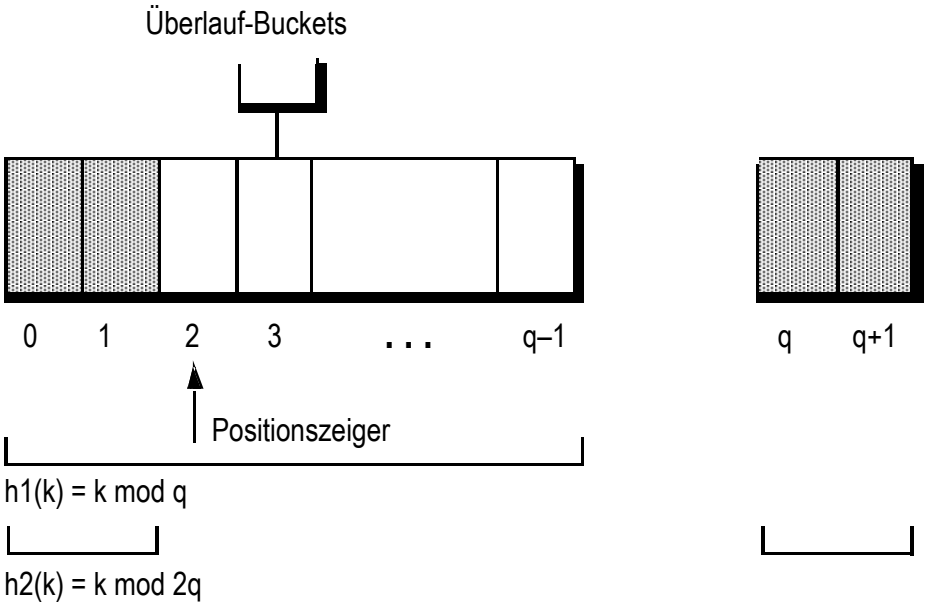
- **Idee: andauernde ("online") Reorganisation der Bucket-Folge während der Einfügungen und Löschungen**
  - Nebeneinander alte Hash-Funktion (für kleinere Zahl von Buckets) und neue Hash-Funktion (für größere Zahl) benutzen
  - Neue Hash-Funktion muss die Schlüssel, die die alte auf einen Bucket abgebildet hat, auf zwei Buckets verteilen (möglichst gleichmäßig)
  - Mit **q** Buckets beginnen;
    jeder mit maximal **b** Sätzen ("Bucket-Faktor")
    - Kapazität also insgesamt $q \times b$ Sätze
  - Belegungsfaktor β := Anzahl gespeicherter Sätze / Kapazität
  - Verwendung eines **Schwellenwerts** α für den Belegungsfaktor,
    $0 \leq \alpha \leq 1$, typischer Wert: 0,8
  - Wenn $\beta > \alpha$:
    Menge der Buckets vergrößern
    - Bis dahin ggf. Überlauf-Buckets wie gehabt

Erster Ansatz: "VH1"



- Wenn α überschritten: auf einen Schlag q, 2q, 4q, … neue Blöcke belegen, direkt hinter den bisherigen Buckets (Datei vergrößern)
- Komplettes Umspeichern aller Sätze verzögern:
  Erst wenn beim nächsten Einfügen eines Satzes in Bucket i (i < q) Überlauf-Bucket gefunden wird, Rehash der Sätze dieses Buckets inkl. Überläufer mit h2; Verteilung der Sätze auf Buckets i und i+q, dann Bit i setzen

Zweiter Ansatz: "Lineares Hashing"



- Wenn α überschritten: *einen* neuen Bucket hinten anfügen und Bucket, auf dem Positionszeiger p steht, aufteilen (auch wenn kein Überlauf-Bucket!)
- Immer zuerst h1 anwenden; falls Ergebnis kleiner als p, auf h2 übergehen
- Nachdem Bucket q–1 aufgeteilt wurde, p wieder auf Null setzen;
  h1 dann nicht mehr benötigt;
  nach Aufteilen von Bucket 0 h2 und h3 im Einsatz

Lineares Hashing (2)

- **Dynamisches Wachsen (und Schrumpfen) des (primären) Hash-Bereichs**
  - Buckets werden in einer fest vorgegebenen Reihenfolge gesplittet.
    $\Rightarrow$ Einzige Hilfsstruktur: nächstes zu splittendes Bucket (Positionszeiger)
  - D.h. minimale Verwaltungsdaten
  - Keine großen Directories für die Hash-Datei

  - Aber: keine Möglichkeit, Überlaufsätze vollständig zu vermeiden!
    - Auch eine hohe Rate von Überlaufsätzen kann als Indikator dafür genommen werden, dass die Datei eine zu hohe Belegung aufweist und deshalb erweitert werden muss.

Anhang

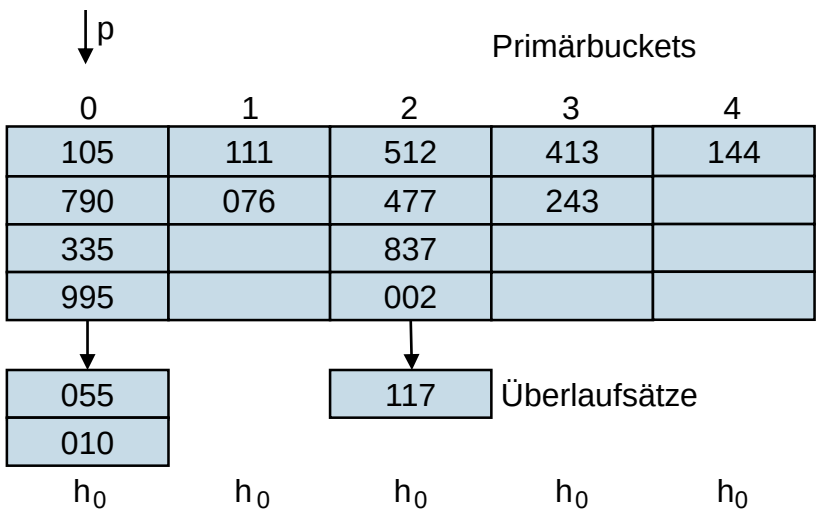
- **Nicht mehr verwendete Folien**
- **Zum Nachschlagen bei Bedarf**

4.A.1 Lineares Hashing

- **Prinzipieller Ansatz**
  - q: Größe der Ausgangsdatei in Buckets
  - Folge von Hash-Funktionen h_0, h_1, ...
    - wobei $h_0(k) \in \{0, 1, ..., q-1\}$ und
      $h_{L+1}(k) = h_L(k)$ oder
      $h_{L+1}(k) = h_L(k) + 2^L \cdot q$
      für alle $L \geq 0$ und alle Schlüssel k gilt
  - Beispiel:
    - $h_L(k) = k \bmod (2^L \cdot q)$, L = 0, 1, ...

Lineares Hashing (2)

- **Beschreibung des Dateizustands**
  - L: Anzahl der bereits ausgeführten Verdopplungen
  - N: Anzahl der gespeicherten Sätze
  - b: Kapazität eines Buckets
  - p: zeigt auf nächstes zu splittendes Bucket ($0 \leq p < q\ 2^L$)
  - β: Belegungsfaktor = $\dfrac{N}{\left(q \cdot 2^L + p\right) \cdot b}$
- **Beispiel:**
  - $h_0(k) = k \bmod 5$
  - $h_1(k) = k \bmod 10$, ...
  - $b = 4$, $L = 0$, $q = 5$
  - Splitt, sobald $\beta > \alpha = 0{,}8$
  - aktuell: $N = 16$,
    $\beta = 16 / (5 + 0) \cdot 4 = 0{,}8$



Lineares Hashing (3)

- **Splitt**
  - Einfügen von 888 erhöht Belegungsfaktor auf $\beta = 17/20 = 0{,}85$
  - Einfügen von 244, 399 und 100 erhöht Belegungsfaktor auf $\beta = 20/24 = 0{,}83$
  - Auslösen eines weiteren Splitts:

p

Primärbuckets

0	1	2	3	4	5
790	111	512	413	144	105
010	076	477	243		335
		837	888		995
		002			055
		117			

Überlaufsätze

h_1	h_0	h_0	h_0	h_0	h_1

p

Primärbuckets

0	1	2	3	4	5	6
790	111	512	413	144	105	076
010		477	243	244	335	
100		837	888	399	995	
		002			055	
		117				

Überlaufsätze

h_1	h_1	h_0	h_0	h_0	h_1	h_1

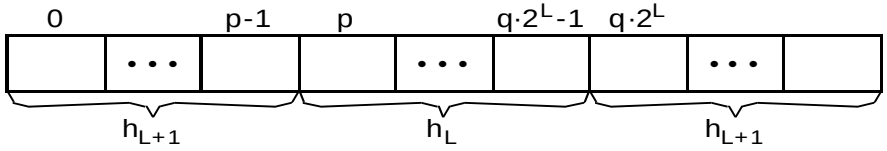
Lineares Hashing (4)

- **Splitt**
  - Auslöser: β
  - Position: p
  - Datei wird um 1 Bucket vergrößert
  - p wird inkrementiert: $p = (p+1) \bmod (2^L \cdot q)$
  - Wenn p wieder auf Null gesetzt wird (Verdopplung der Datei beendet), wird L inkrementiert.

- **Adressberechnung**
  - Wenn $h_0(k) \geq p$, dann ist $h_0(k)$ die gewünschte Adresse.
  - Wenn $h_0(k) < p$, dann ist Bucket bereits gesplittet. $h_1(k)$ liefert die gewünschte Adresse.
  - allgemein:

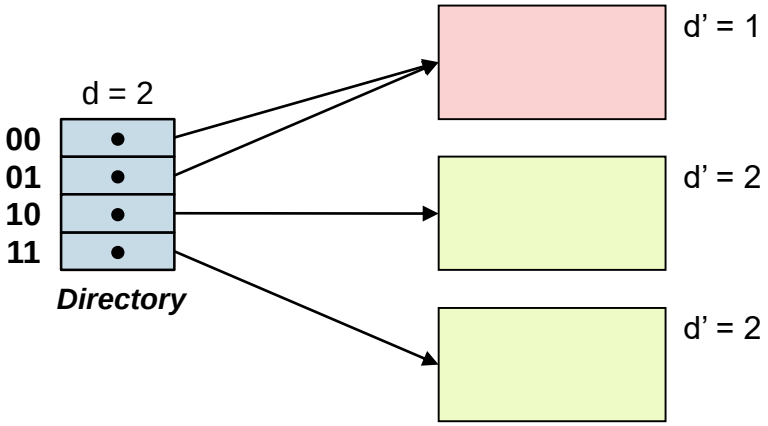
    $h := h_L(k);$

    if $h < p$ then $h := h_{L+1}(k);$



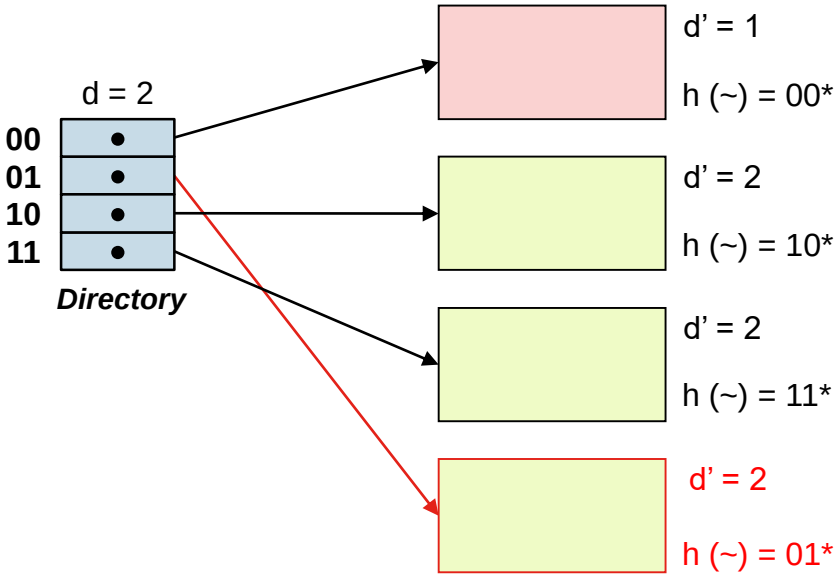
4.A.2 Erweiterbares Hash-Verfahren (Extendible Hashing)

- **Ebenfalls dynamisches Wachsen und Schrumpfen des Hash-Bereichs**
  - Buckets erst bei Bedarf bereitgestellt
  - Hohe Speicherplatzbelegung möglich
- **Keine Überlauf-Bereiche, jedoch Zugriff über Directory**
  - Zwei Blockzugriffe, 1 bis 2 E/A-Operationen
  - Hash-Funktion generiert Pseudoschlüssel zu einem Satz: Folge von Bits
  - Die ersten **d** Bits des Pseudoschlüssels werden zur Identifizierung eines Directory-Eintrags verwendet (d = globale Tiefe).
  - Directory enthält 2^d Einträge; Eintrag verweist auf Bucket.
  - In einem Bucket werden genau die Sätze gespeichert, deren Pseudoschlüssel in den ersten **d'** Bits übereinstimmen (d' = lokale Tiefe).
  - d = MAX (d')



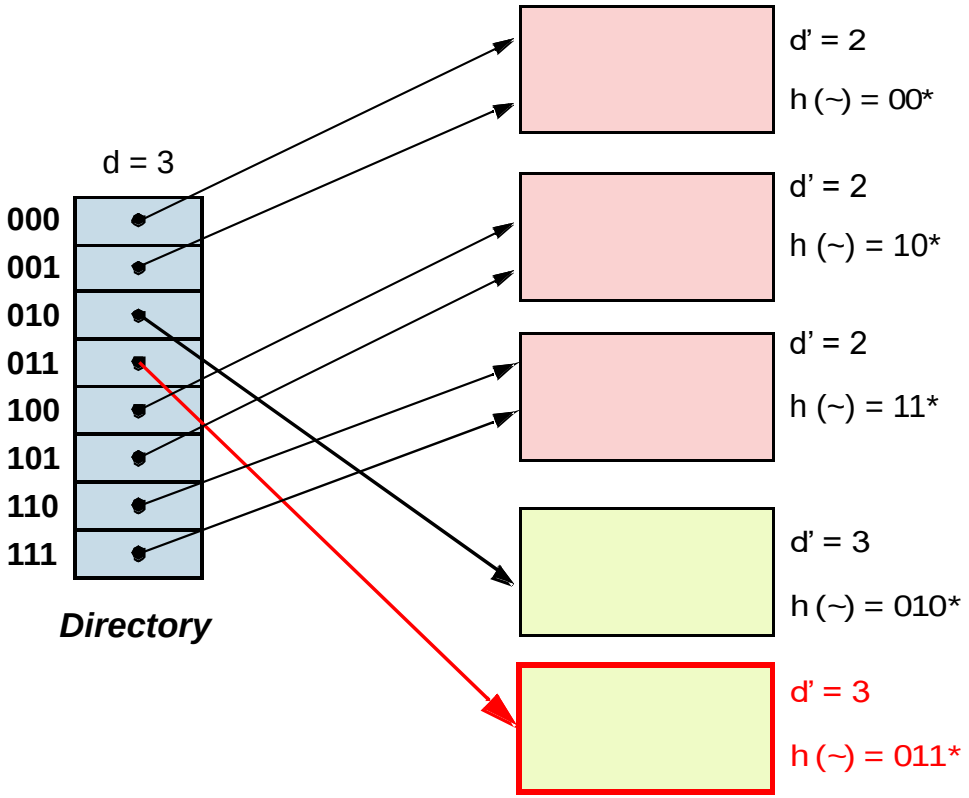
Erweiterbares Hashing (2)

- **Situation**
  - Neuer Satz soll gespeichert werden, aber Bucket ist voll: Überlauf
- **Fall 1**
  - Überlauf eines Buckets, dessen lokale Tiefe kleiner als die globale Tiefe d ist
    - D.h. mehrere Einträge im Directory zeigen auf diesen Bucket
    - Neuen Bucket anlegen
    - Erhöhung der lokalen Tiefe
    - Umverteilung der Sätze nach ihren Bits in der neuen Tiefe
    - Lokale Korrektur der Pointer im Directory



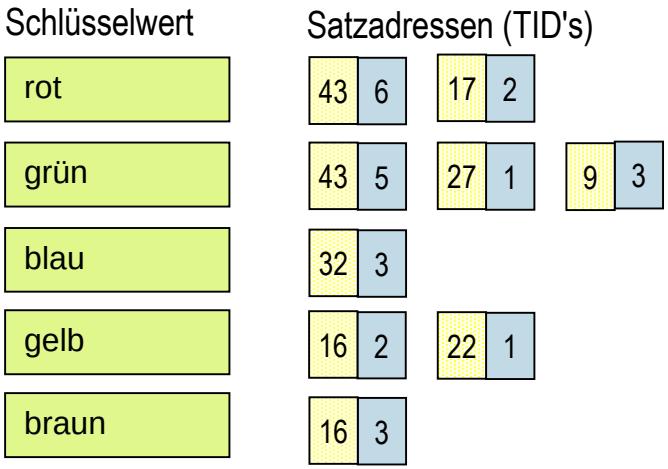
Erweiterbares Hashing (3)

- **Fall 2**
  - Überlauf eines Buckets, dessen lokale Tiefe gleich der globalen Tiefe ist
    - Verdopplung aller Einträge im Directory (Erhöhung der globalen Tiefe)
    - Neuen Bucket anlegen
    - Umverteilung der Sätze nach ihren Bits in der neuen Tiefe
    - Lokale Korrektur der Pointer im Directory



4.A.3 Invertierte Speicherung von Sätzen

- **Verwaltung einer Tabelle**
  - die zu jedem auftretenden Schlüsselwert **Liste von Satzadressen** verwaltet: genau die Sätze, in denen der Schlüsselwert vorkommt



- **Auch Index genannt**
  - Wie in einem Buch: Stichwort und Seitennummern

Invertierte Speicherung (2)

- **Hilfsstruktur**
  - Steht am Anfang/Ende der Datei (neuer Blocktyp) oder in eigener Datei
  - Erheblich kompakter als Menge aller Sätze, d.h. selbst bei sequenziellem Durchsuchen weniger Blöcke zu lesen
- **Sortierung der Tabelleneinträge nach Schlüsselwert**
  - verbessert Suche weiter
  - Binäre Suche allerdings erschwert durch variabel lange Einträge
- **Abspeicherung der Sätze selbst nicht reglementiert!**
  - Kann also mit gestreuter Speicherung nach einem anderen Schlüssel oder auch mit sequenzieller Abspeicherung kombiniert werden
- **Lesen aller Sätze sortiert nach Schlüssel möglich**

Invertierung (3)

- **Nachteile:**
  - Zusätzlicher Speicherplatzbedarf
  - Einfügen oder Löschen eines Satzes
    ändert die Zahl der Satzadressen in einem Eintrag
    - D.h. Verschieben der nachfolgenden Einträge
      (innerhalb des Blocks? über mehrere Blöcke hinweg?)
  - Neuer Satz kann sogar neuen Schlüsselwert mitbringen
    - Sortierordnung erhalten, neuen Eintrag einschieben
- **Also:**
  - Zugriff über Schlüssel effizienter als bei sequenziellem Suchen,
    aber Wartung der Invertierungstabelle bei Änderungen mühsam
- **Lösung:**
  - Hierarchische Strukturierung der Tabelle (Baum)

Vorlesung Implementierung von Datenbanksystemen

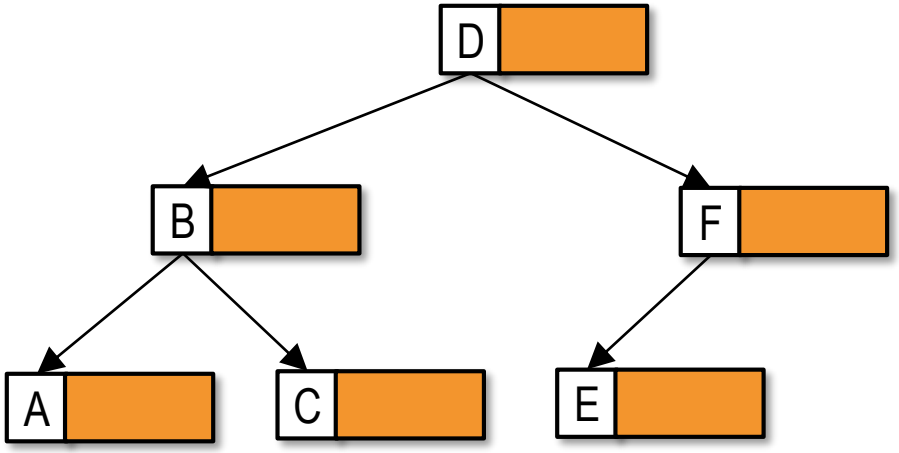
5. Schlüsselzugriff – Teil 2

Prof. Dr. Klaus Meyer-Wegener

Wintersemester 2019/20

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

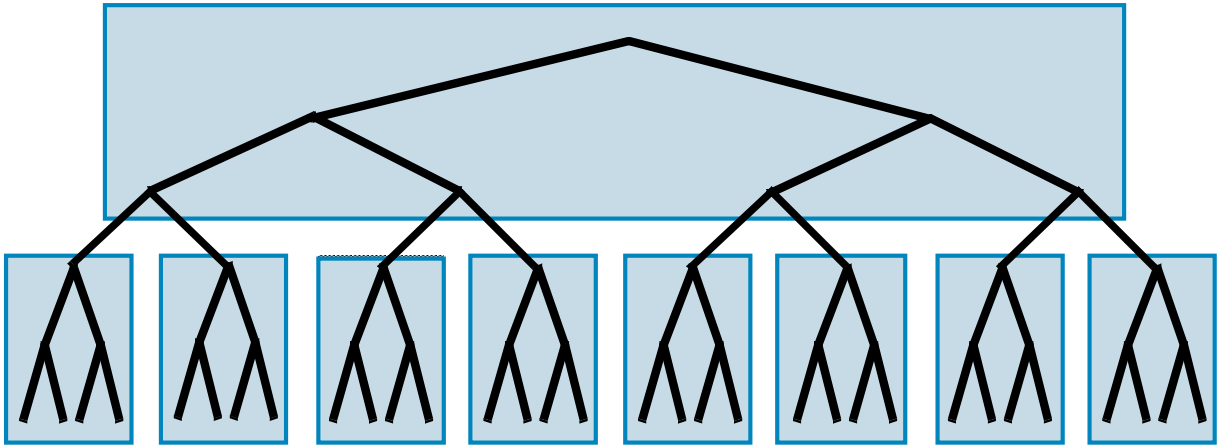
- **Ausgangspunkt: Binäre Such-Bäume (balanciert)**



- Entwickelt für Hauptspeicher
- Prinzipiell machbar auch für Sätze in Dateien:
  Satzadressen als Zeiger
- Dann aber unzureichend:
  Zu viele Blockzugriffe beim Abstieg durch den Baum (einer pro Stufe)

B-Bäume (2)

- **Idee (Bayer und McCreight 1972):**
  - Zusammenfassung ganz bestimmter Sätze in einem Block



  - Mehrweg-Baum,
    bei dem jeder Knoten genau einem Block entspricht
- **Das Ergebnis heißt B-Baum.**
  - B steht für "Block", sagt der Erfinder Rudolf **B**ayer …

Aufbau eines B-Baum-Knotens

n = Anzahl der verwendeten Einträge, **k** $\leq$ n $\leq$ 2k (bzw. in der Wurzel 1 $\leq$ n $\leq$ 2k)

(Ki, Di, Pi) bilden zusammen einen **Eintrag**

Ki = Schlüsselwert

Di = Datensatz

Pi = Zeiger auf den Nachfolgeknoten (= dessen Blocknummer)

Einträge nach Schlüsselwert aufsteigend sortiert

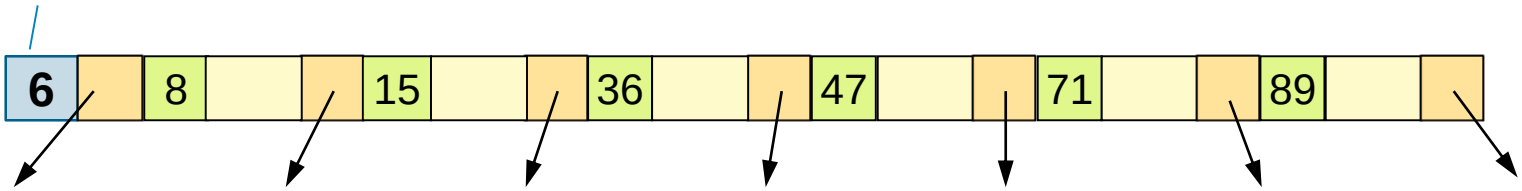
- Zugleich der Inhalt eines Blocks (Wieder ein neuer Blocktyp!)

- **Bedeutung:**
  - Alle Schlüsselwerte im Unterbaum, auf den P0 zeigt, sind kleiner als K1 oder gleich K1
  - Alle Schlüsselwerte im Unterbaum, auf den Pi zeigt (0 < i < n), sind größer als Ki und kleiner oder gleich Ki+1
  - Alle Schlüsselwerte im Unterbaum von Pn sind größer als Kn

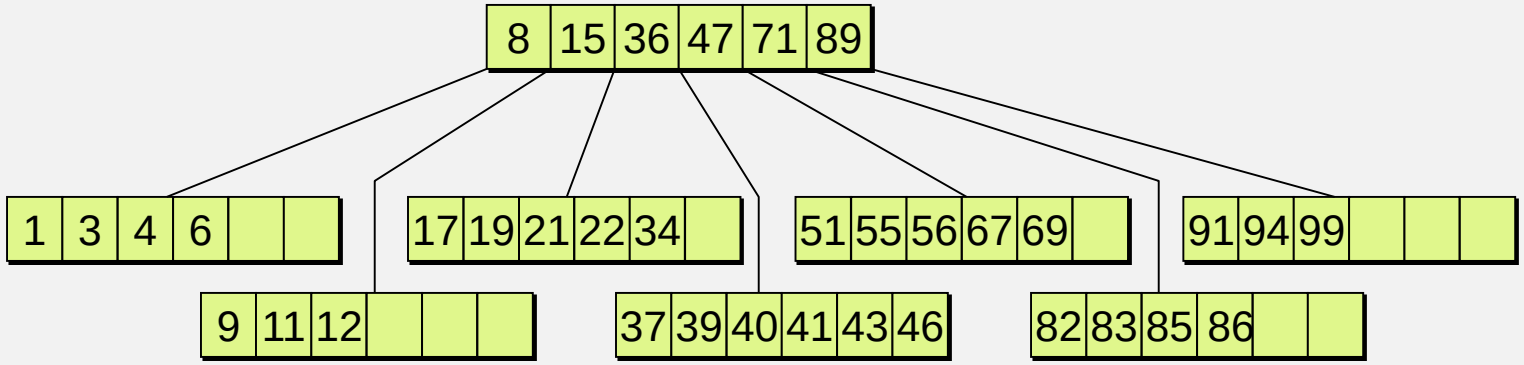
B-Baum: Beispiel mit k = 3

Aufbau eines Knotens: maximal 6 Einträge

Anzahl der tatsächlichen Einträge in diesem Knoten

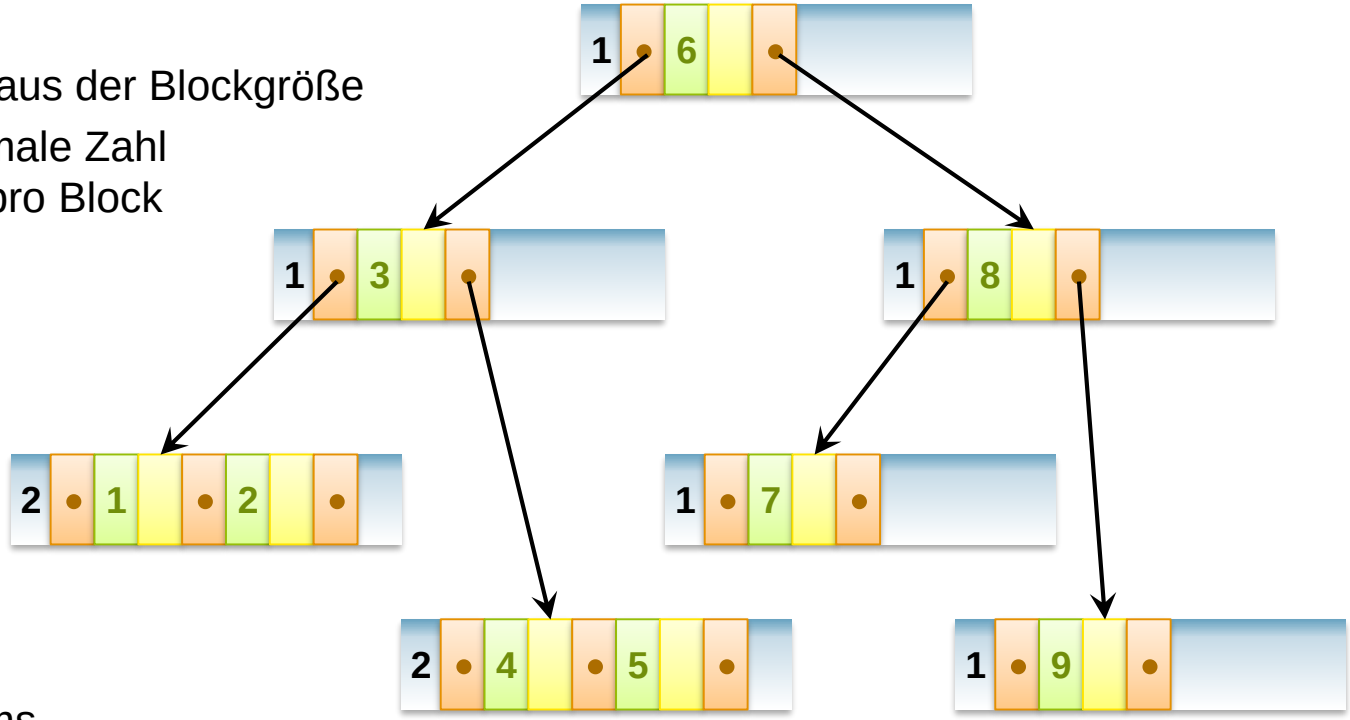


B-Baum der Höhe 2



B-Baum-Parameter

- **k**
  - Errechnet sich aus der Blockgröße
  - 2k ist die maximale Zahl von Einträgen pro Block



- **h**
  - Höhe des Baums
    - Anzahl der Kanten von der Wurzel bis zum Blatt plus 1, also Anzahl der Ebenen
  - Ergibt sich aus:
    Anzahl der gespeicherten Datenelemente und Einfügereihenfolge

Im Beispiel:
k = 1, h = 3

B-Baum-Eigenschaften

- **Jeder Pfad**
  - vom Wurzelknoten zu einem der Blattknoten hat **dieselbe Länge h–1.**
    - Baum ist perfekt balanciert.
- **Jeder Knoten**
  - mit Ausnahme des Wurzelknotens und der Blattknoten hat **mindestens k+1 Nachfolger.**
    - Jeder Block also mindestens halb voll: Speicherplatz-Ausnutzung > 50 % (bei jeder Zahl von Sätzen).
- **Der Wurzelknoten**
  - ist entweder ein Blattknoten oder hat mindestens 2 Nachfolger.
- **Jeder Knoten**
  - hat **höchstens 2k+1 Nachfolger.**
    - Ergibt sich aus der Block-Größe: Dann ist der Block voll.

Suche im B-Baum

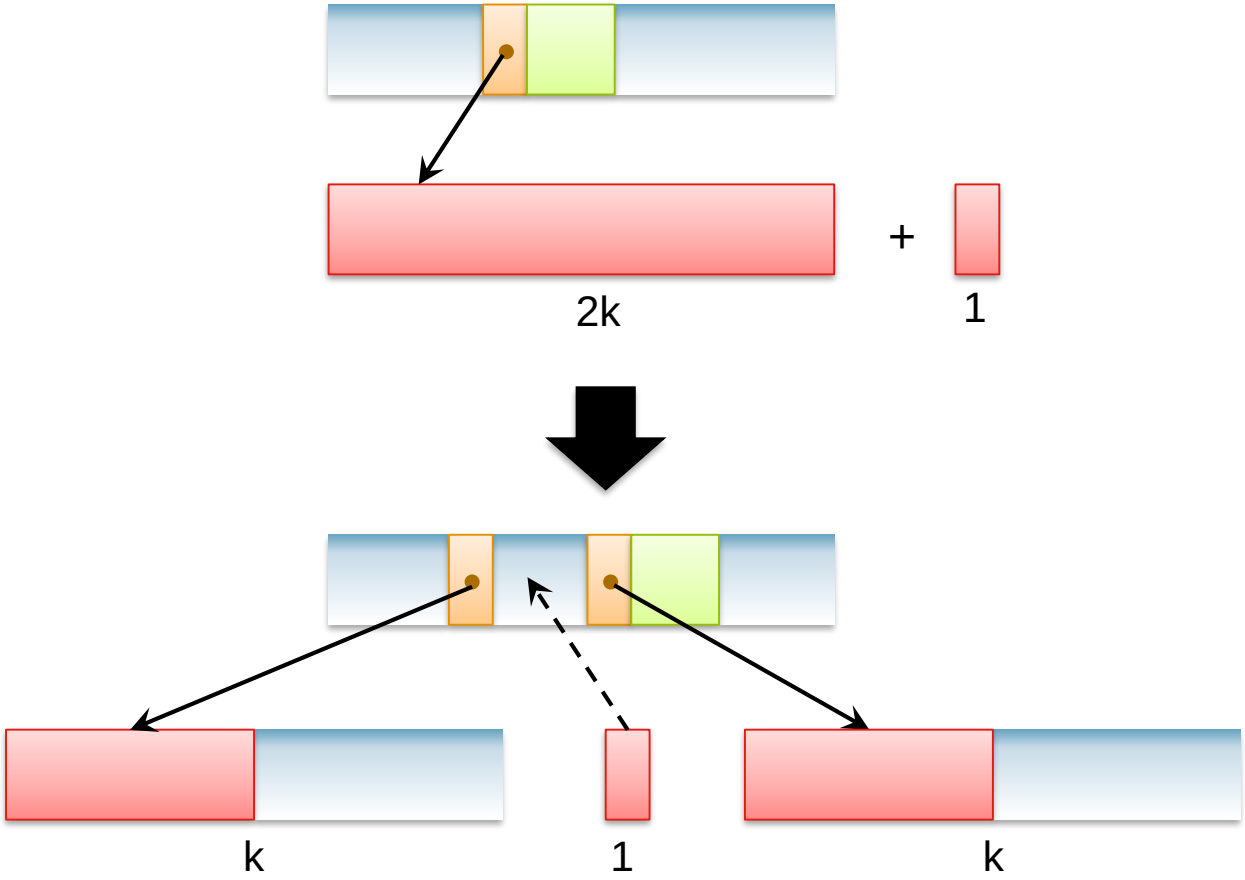
- **Beginnend mit dem Wurzelknoten,**
  wird ein Knoten jeweils von links nach rechts durchsucht:
  (1) Stimmt Ki mit dem gesuchten Schlüsselwert überein,
  ist der Satz gefunden.
  (2) Ist Ki größer als der gesuchte Wert,
  wird die Suche in der Wurzel des an Pi–1 hängenden Unterbaums fortgesetzt.
  (3) Ist Ki kleiner als der gesuchte Wert,
  wird der Vergleich mit Ki+1 wiederholt.
  (4) Ist auch Kn noch kleiner als der gesuchte Wert,
  wird die Suche im Unterbaum von Pn fortgesetzt.
- **Falls weiterer Abstieg in Unterbaum (über Pi–1 in (2) oder Pn in (4)) nicht möglich (d.h. Blattknoten):**
  - Suche abbrechen,
    kein Satz mit gewünschtem Schlüsselwert vorhanden

Einfügen eines Satzes im B-Baum

- **Eingefügt wird nur in Blattknoten!**
  - D.h. zunächst Abstieg durch den Baum wie bei der Suche
  - Im so gefundenen Blattknoten
    Satz entsprechend der Sortierreihenfolge einfügen

- **Sonderfall: Blattknoten schon voll (enthält 2k Sätze)**
  - **Splitt:** neuen Blattknoten erzeugen
  - Die 2k+1 Sätze (in Sortierordnung!) halbe-halbe aufteilen
    zwischen altem und neuem Blattknoten:
    - Die ersten k Sätze in den ersten (linken) Block
    - Die letzten k Sätze in den zweiten (rechten) Block
    - Den mittleren (k+1-ten) Satz als neuen "Diskriminator",
      d.h. als Verzweigungsinformation bei der Suche, in den Knoten eine Stufe höher einfügen, der auf den Blattknoten verweist (zusammen mit einem Verweis auf den neuen Blattknoten)

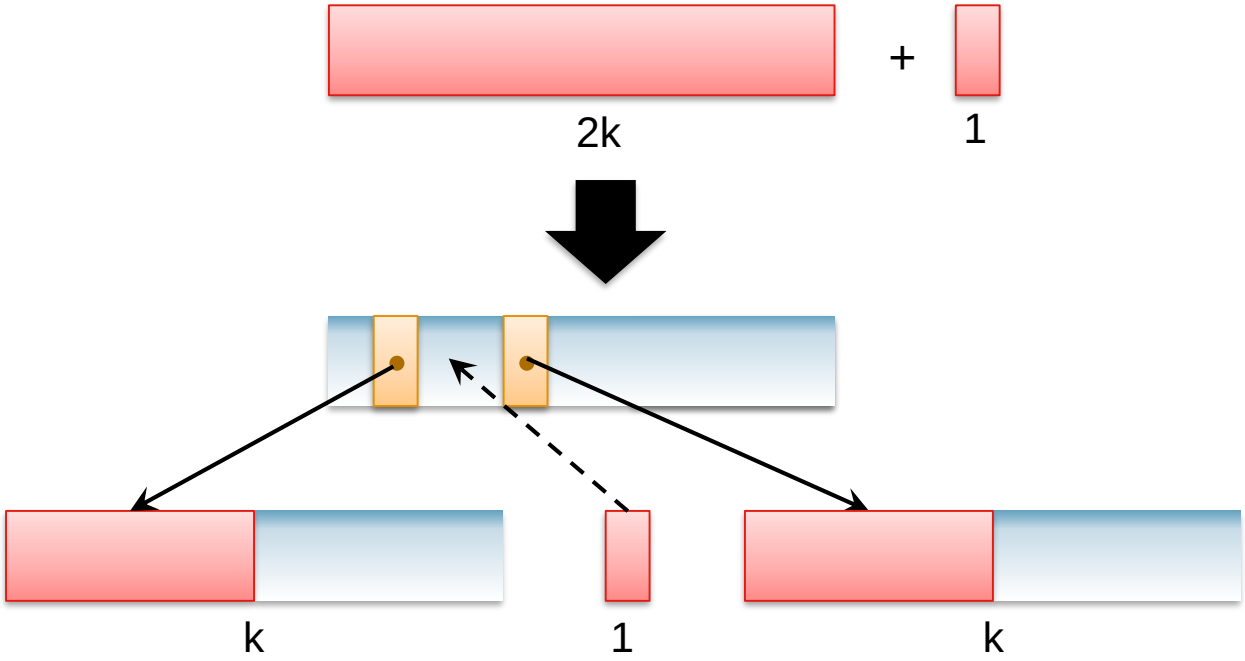
Einfügen B-Baum (2)



- Falls auch der übergeordnete Knoten voll:
  - Splitt auf dieser Ebene wiederholen

Einfügen B-Baum (3)

- **Weiterer Sonderfall: Splitt des Wurzelknotens**
  - Erzeugung von *zwei* neuen Knoten:



  - Dann (und nur dann!) wächst die Höhe des Baums um 1.
    - (Man sagt bildhaft: Der Baum "reißt von unten nach oben auf".)

Einfügen B-Baum (4)

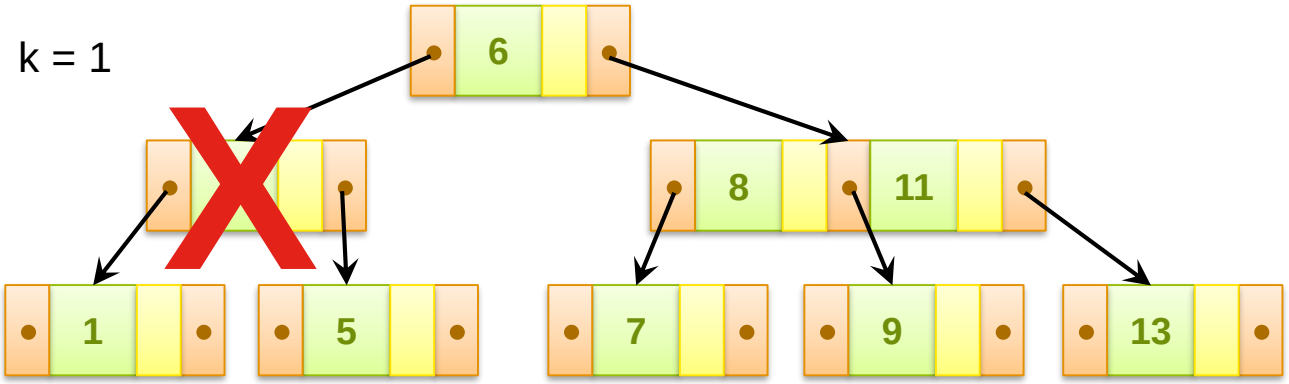
- **Dynamische Reorganisation**
  - Kein Entladen und Neuladen erforderlich
  - Baum immer balanciert

- **Speicherplatzausnutzung:**
  - Jeder Knoten (bis auf die Wurzel)
    ist immer mindestens halb voll,
    d.h. Speicherausnutzung garantiert $\geq$ 50 %
  - Bei zufälliger und gleichverteilter Einfügung ergibt sich eine Speicherausnutzung von ln 2, also rund 70 %

Löschen im B-Baum

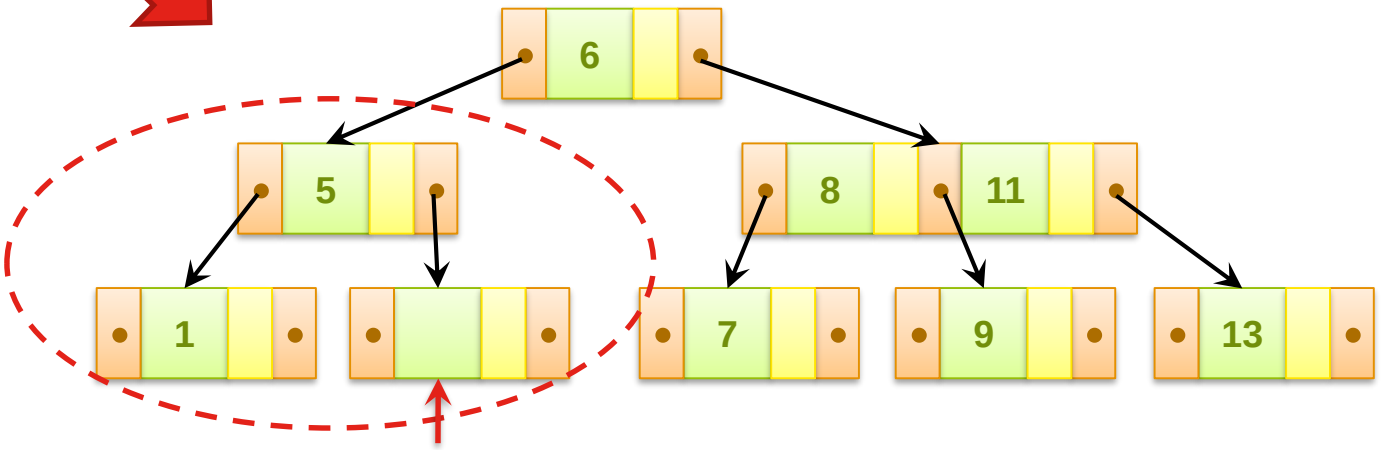
- **... erstmal am Beispiel !!**

k = 1



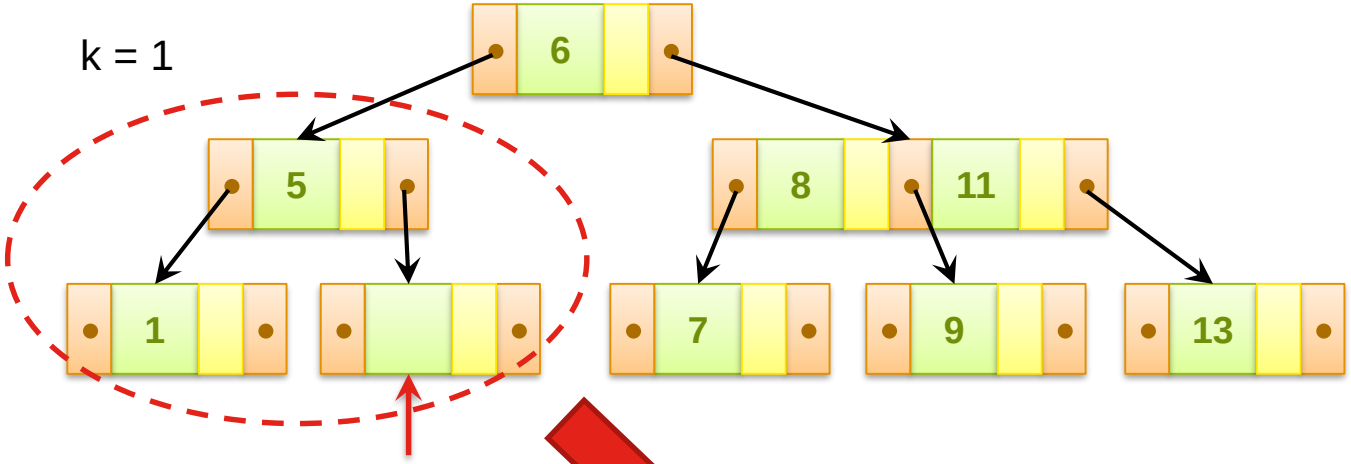
Lösche Schlüssel Nr. 3:

Unterlauf! ➔ Mischen

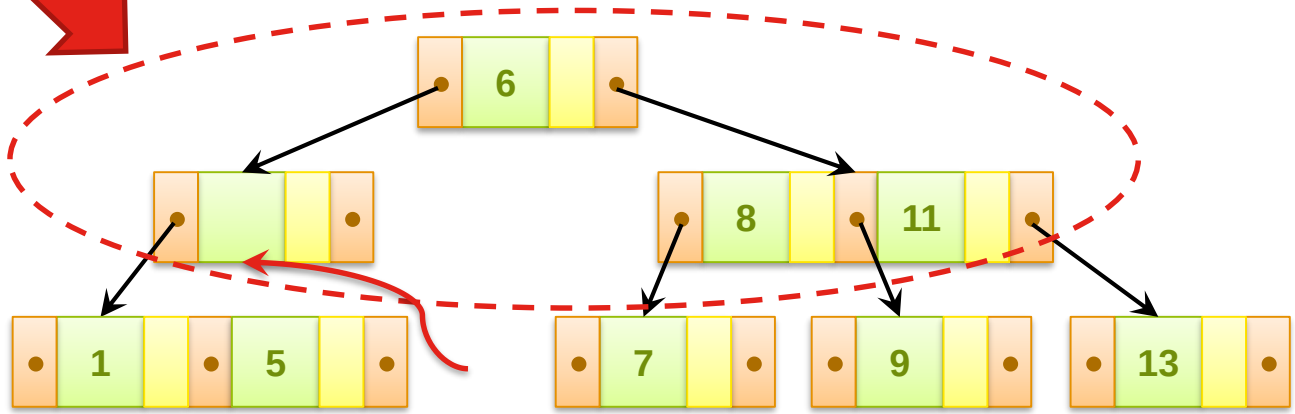


Löschen im B-Baum (2)

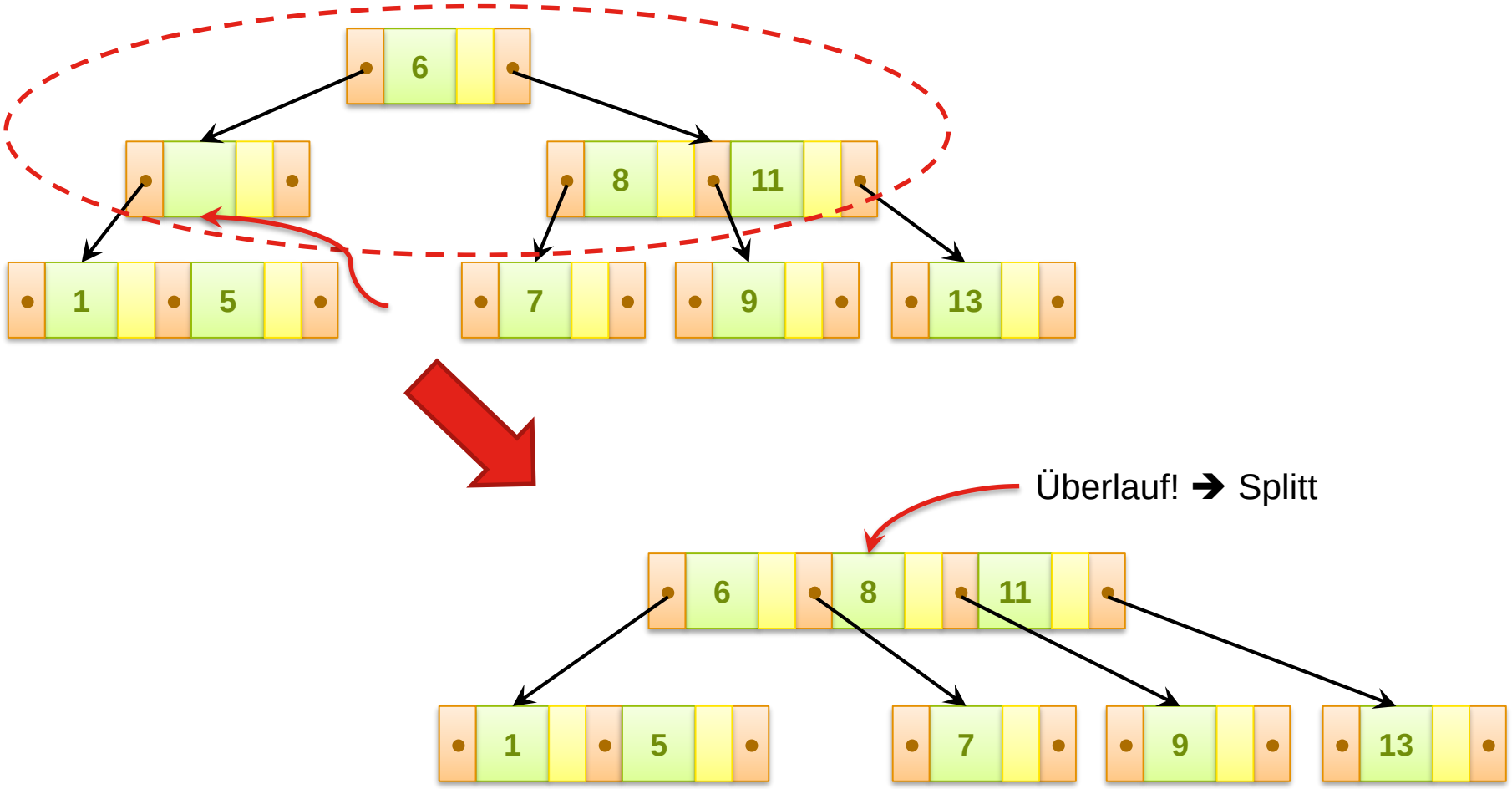
k = 1



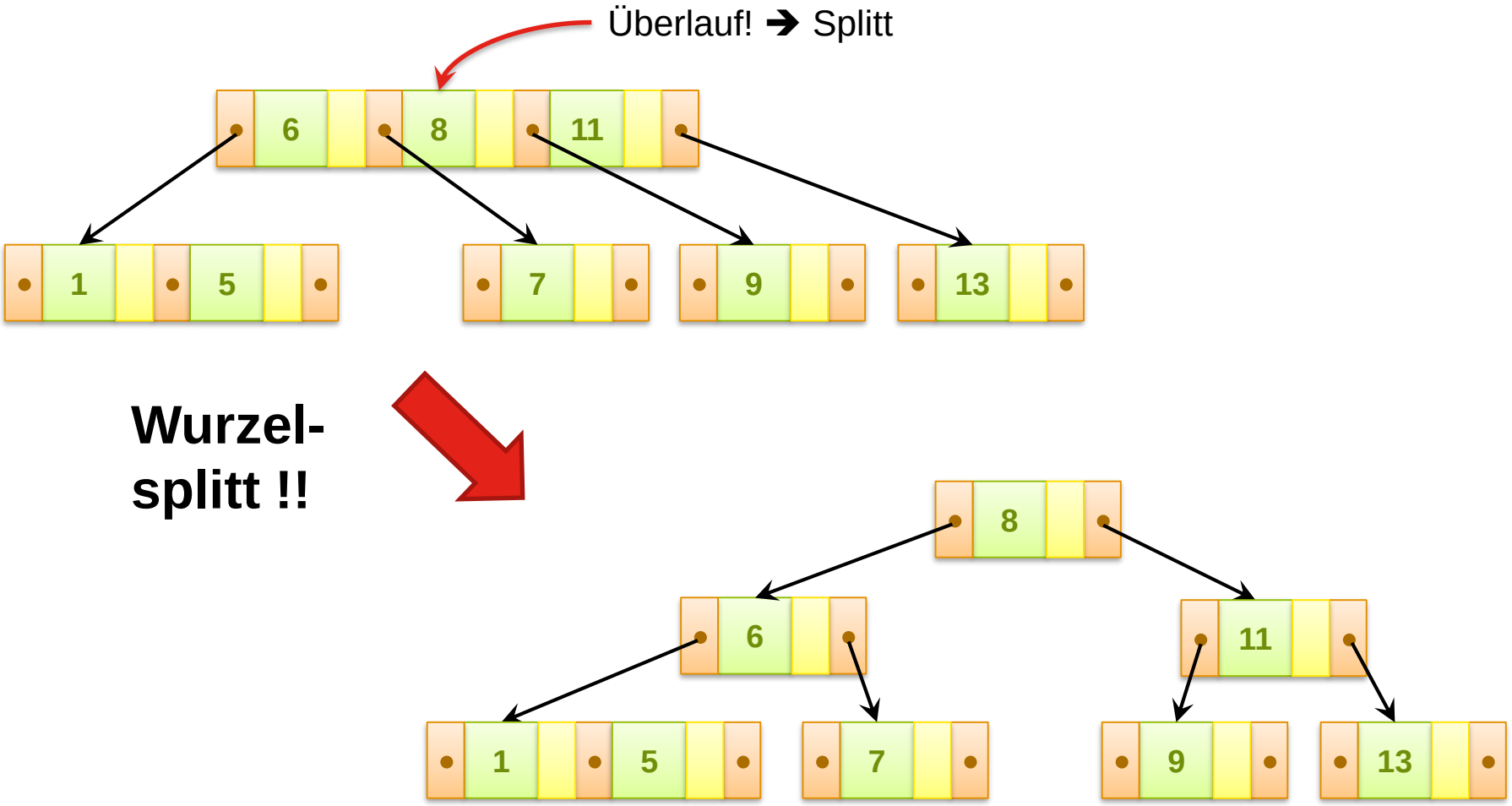
Unterlauf! ➔ Mischen



Löschen im B-Baum (3)



Löschen im B-Baum (4)



Löschvorgang – algorithmisch

- **War nur Beispiel – es gibt verschiedene Algorithmen**
  - Such den Knoten, in dem der **zu löschende Schlüssel S** liegt.
  - Falls Schlüssel S **im Blattknoten**,
    dann lösch S dort
    und behandle ggf. entstehenden Unterlauf.
  - Falls Schlüssel S **im innerem Knoten**,
    dann untersuch linken und rechten Unterbaum von S:
    - Betrachte Blattknoten mit direktem Vorgänger S' von S
      und Blattknoten mit direktem Nachfolger S'' von S.
    - Wähl den aus, der mehr Elemente hat.
      Falls beide gleich viele Elemente haben, wähl zufällig einen aus.
    - Ersetz den zu löschenden Schlüssel S
      durch S' bzw. S'' aus dem gewählten Blattknoten.
    - Lösch S' bzw. S'' im gewählten Blattknoten
      und behandle ggf. entstehenden Unterlauf.

Anmerkungen zum Unterlauf

- **Anmerkungen**
  - Ein endgültiger Unterlauf entsteht bei obigem Algorithmus erst auf Blattebene!
  - Unterlaufbehandlung wird durch Mischen des Unterlaufknotens mit seinem Nachbarknoten und dem darüber liegenden Diskriminator durchgeführt.
    - Sozusagen Splitt rückwärts ausführen
  - Wurde einmal mit dem Mischen auf Blattebene begonnen, so setzt sich dieses evtl. nach oben hin fort.
  - Das Mischen auf Blattebene wird so lange weitergeführt, bis kein Unterlauf mehr existiert oder die Wurzel erreicht ist.
  - Wird die Wurzel erreicht, kann der Baum in der Höhe um 1 schrumpfen. Beim Mischen kann es auch wieder zu einem Überlauf kommen. In diesem Fall muss wieder gesplittet werden.

Variante: B*-Baum

(Bisweilen auch B+-Baum genannt)

- **Eigenschaften und Unterschiede zum B-Baum**
  - Alle Sätze werden in den **Blattknoten** abgelegt.
  - **Innere Knoten** enthalten nur noch Verzweigungsinformation, keine Daten.

  - Aufbau von B*-Baum-Knoten:

Innerer Knoten

n	P_0	R_1	P_1	R_2	P_2	. . .	R_m	P_m	freier Platz

P_i = Zeiger auf Knoten, R_i = Referenzschlüssel, $k \leq m \leq 2k$

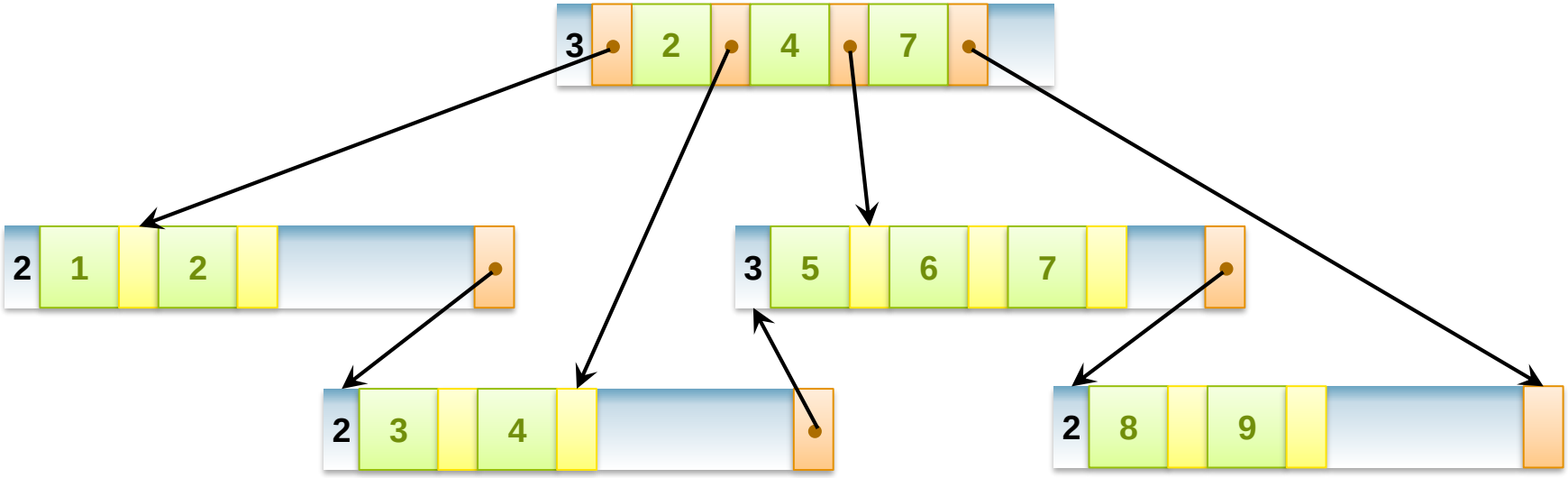
Blatt-Knoten

n	V	S_1	D_1	S_2	D_2	. . .	S_j	D_j	freier Platz	N

V = Vorgänger-Zeiger, N = Nachfolger-Zeiger, $k^* \leq j \leq 2k^*$

Beispiel eines B*-Baums

- Ohne Vorgänger-Zeiger

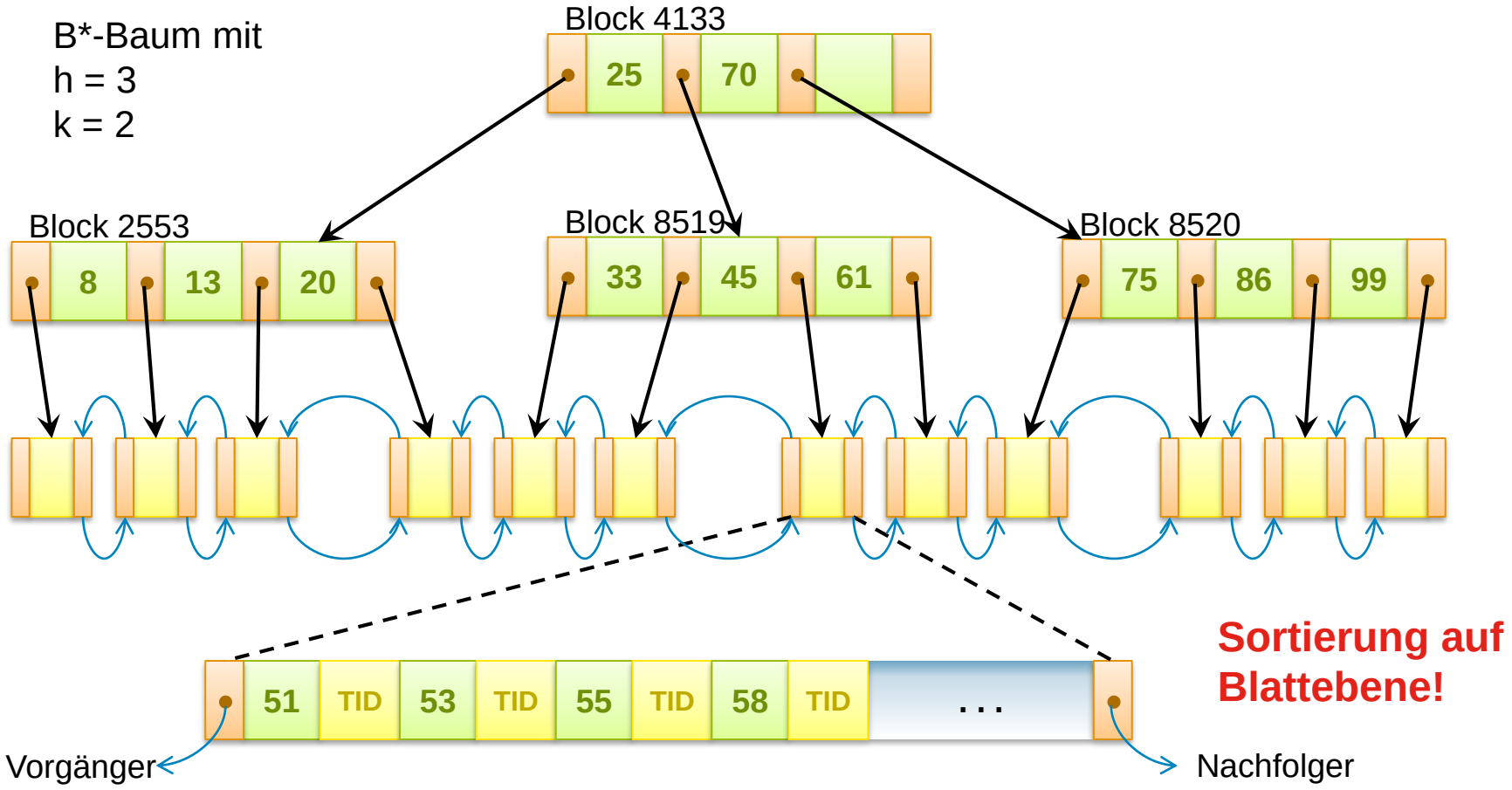


- **Sortierung der Sätze auf Blattebene!**

Weiteres Beispiel eines B*-Baums

- Etwas umfangreicher:

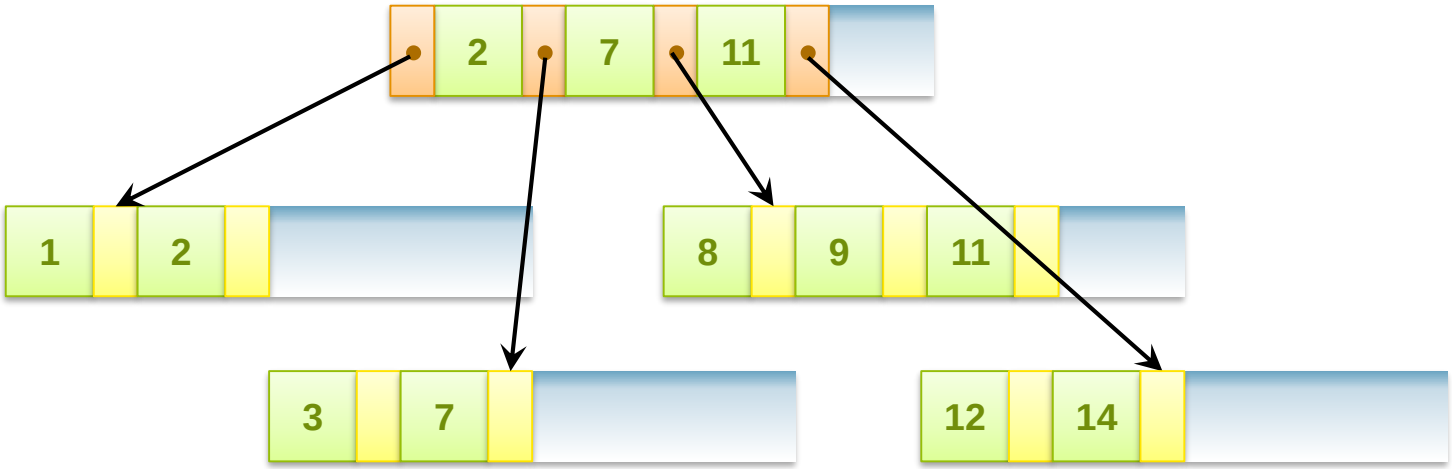
B*-Baum mit
h = 3
k = 2



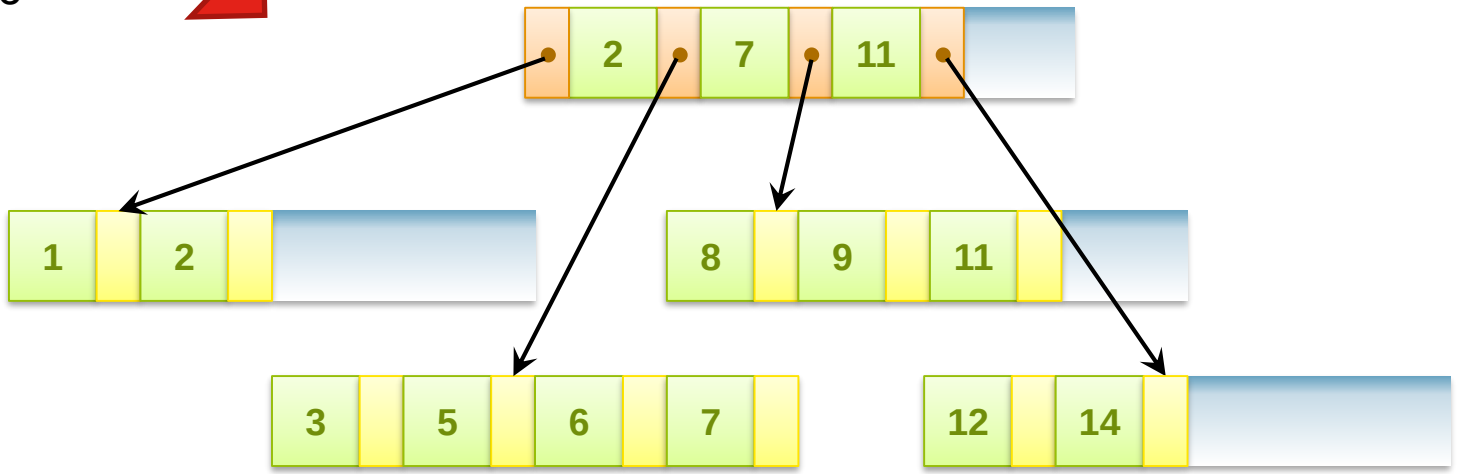
Sortierung auf Blattebene!

Einfügen im B*-Baum

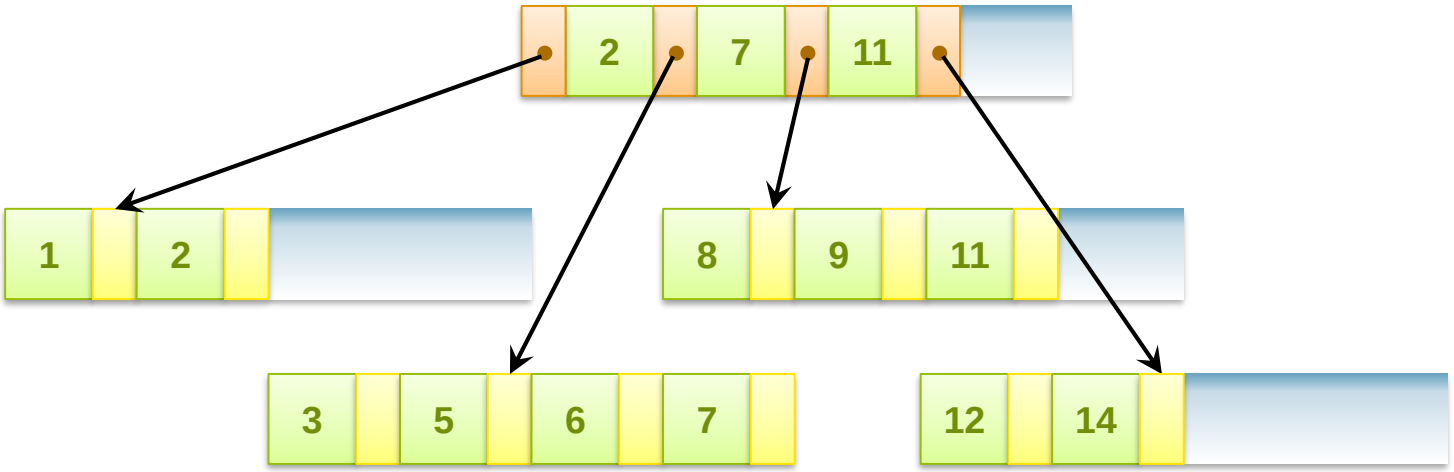
- ... am Beispiel



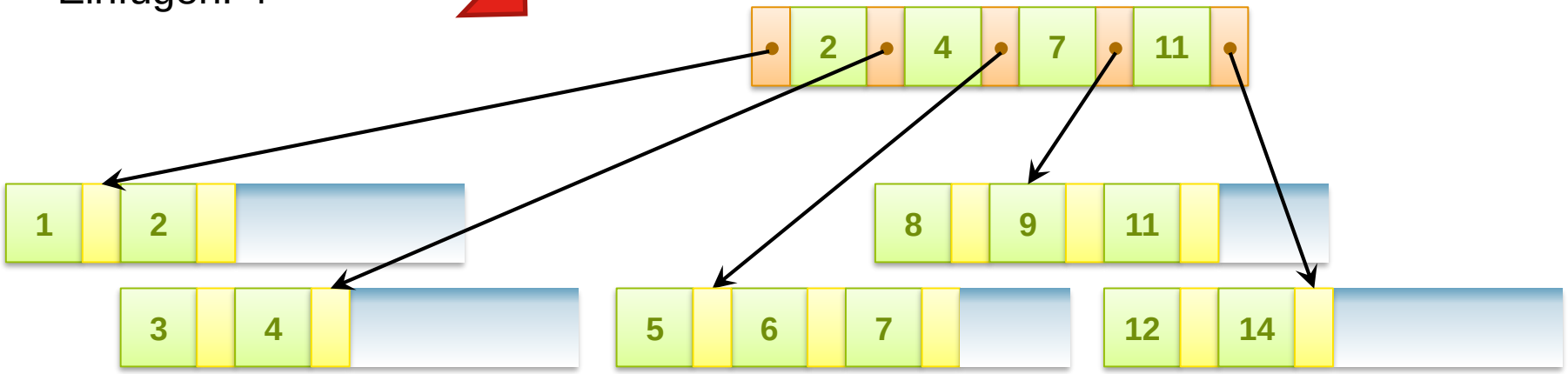
Einfügen: 5, 6



Splitt im B*-Baum

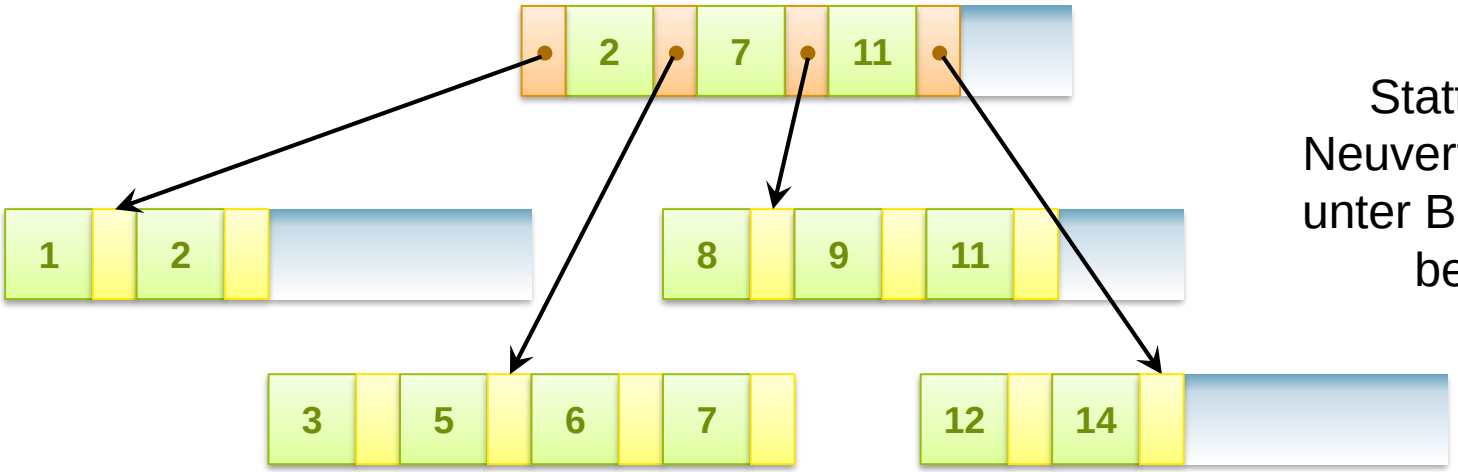


Einfügen: 4

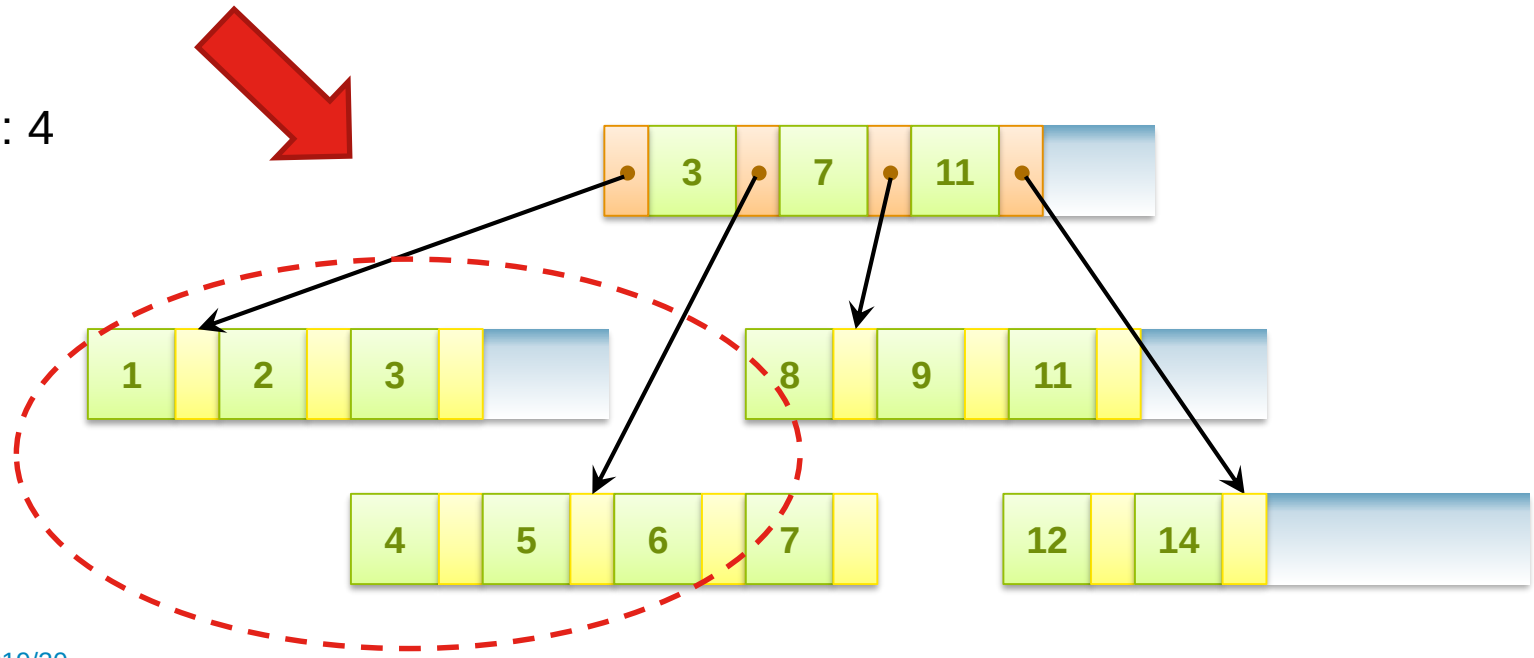


Variante beim Splitt im B*-Baum

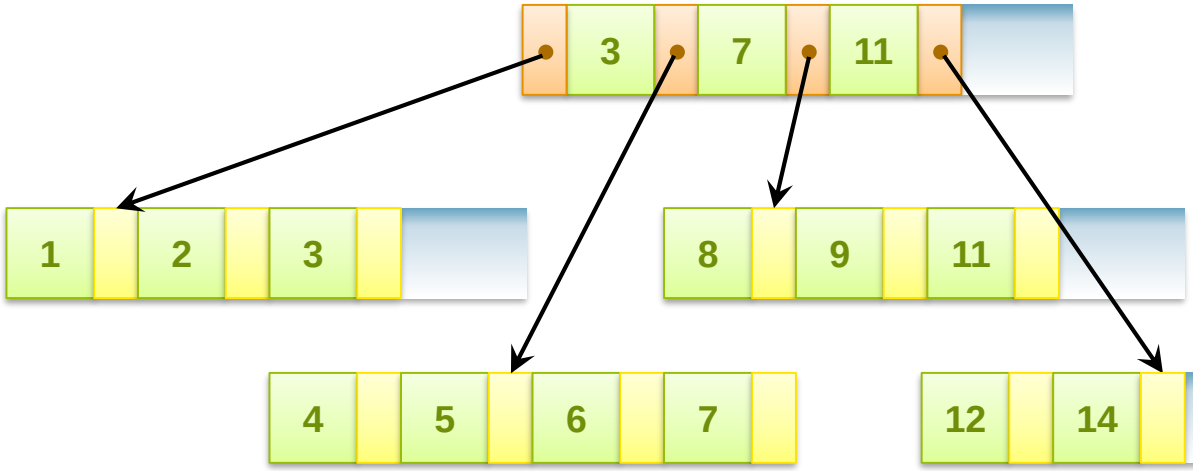
Statt Splitt bei Überlauf, Neuverteilung der Einträge unter Berücksichtigung der benachbarten Knoten



Einfügen: 4

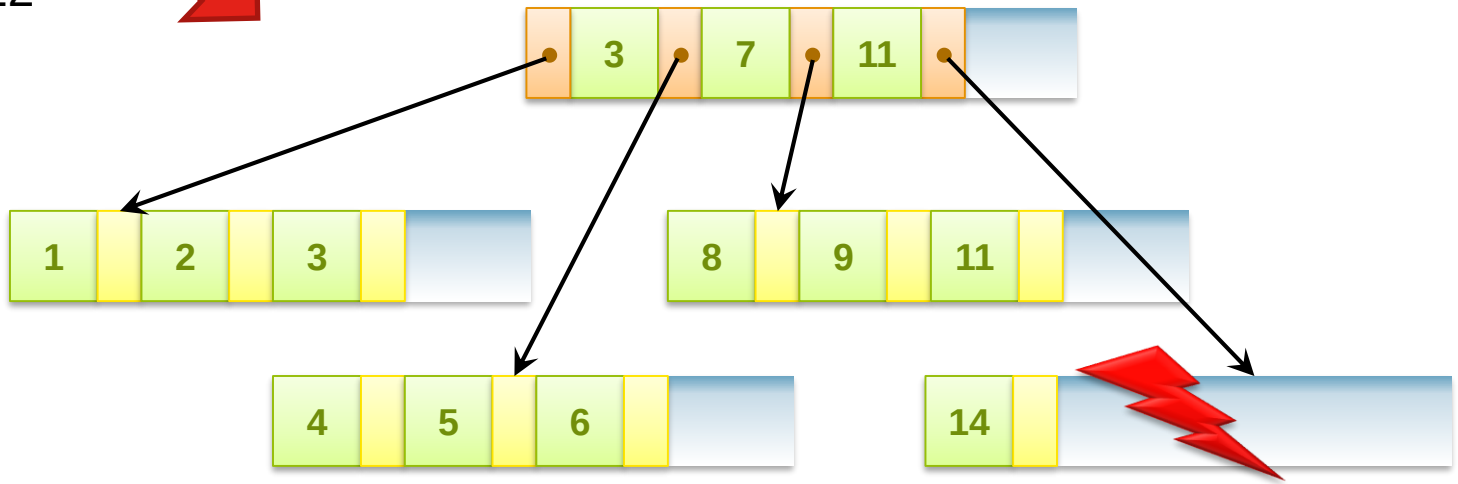


Löschen im B*-Baum



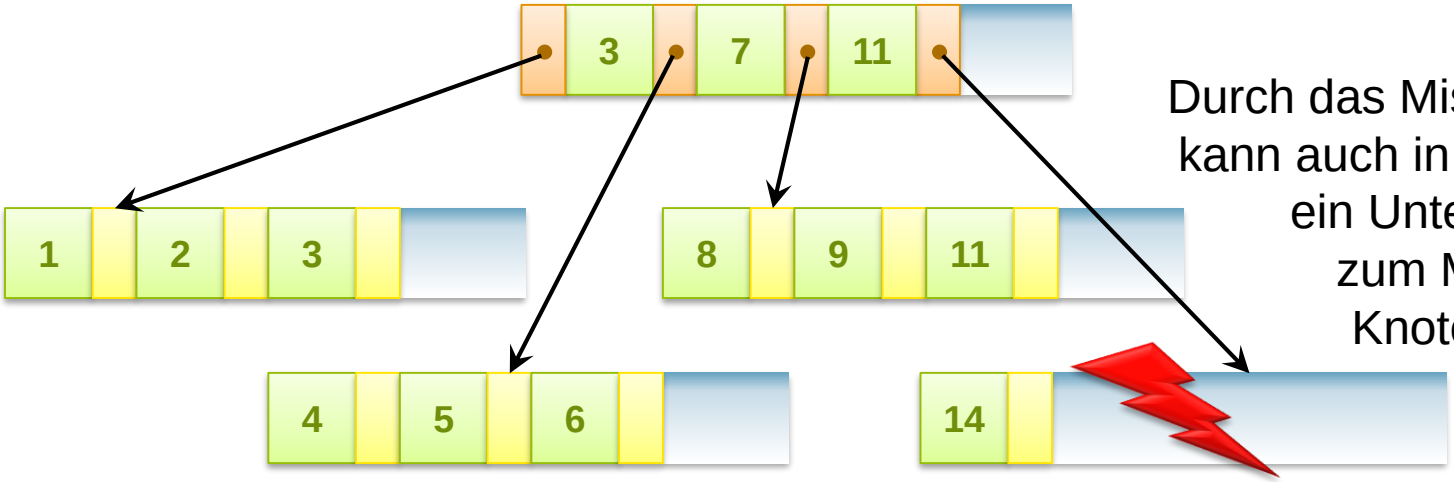
Merke:
Beim Löschen von "7" bleibt "7" als Diskriminator erhalten

Löschen: 7, 12

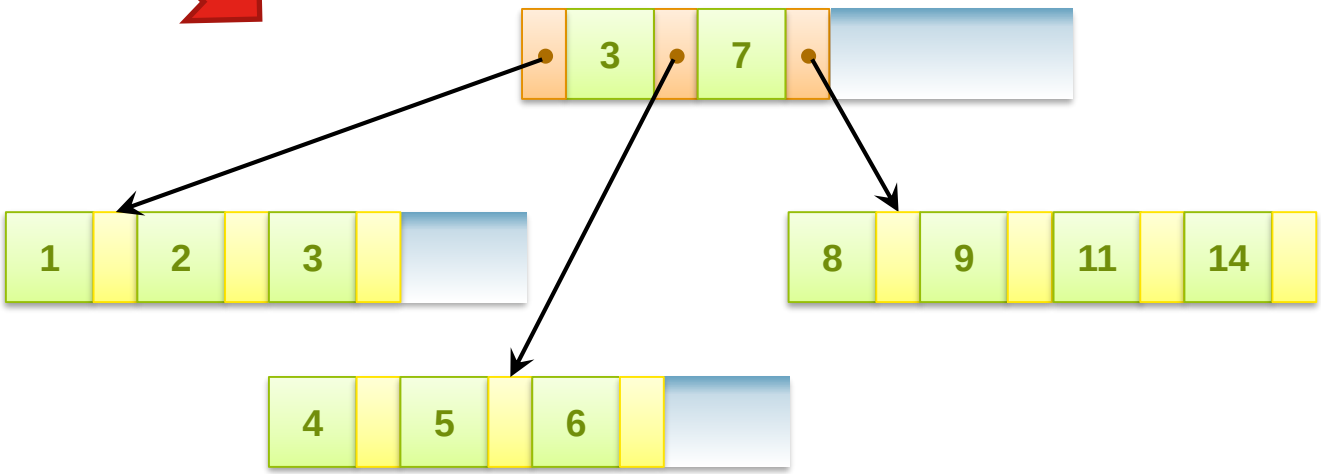


Unterlauf!!

Löschen im B*-Baum (2)



Durch das Mischen zweier Blätter kann auch in den inneren Knoten ein Unterlauf entstehen, der zum Mischen von inneren Knoten führt – kann sich analog zum Splitt bis zur Wurzel fortpflanzen.



Löschen von Sätzen aus B*-Baum

- **1. Such den zu löschenden Eintrag im Baum.**
- **2. Entsteht durch das Löschen ein Unterlauf? (#Einträge < k?)**
  - **NEIN**
    - Entfern den Satz aus dem Blatt.
      (Eine Aktualisierung des Diskriminators im Vaterknoten ist nicht erforderlich!)
  - **JA**
    - Prüf das Blatt zusammen mit einem Nachbarknoten:
    - Ist die Summe der Einträge in beiden Knoten größer als 2k?
    - **NEIN**
      - Misch beide Blätter zu einem Blatt zusammen.
      - Falls dabei ein Unterlauf im Vaterknoten entsteht: Misch die inneren Knoten analog.
    - **JA**
      - Teil die Sätze neu auf beide Knoten auf, so dass ein Knoten jeweils die Hälfte der Sätze aufnimmt.
      - Der Diskriminator im Vaterknoten ist entsprechend zu aktualisieren.

Vergleich B- und B*-Baum

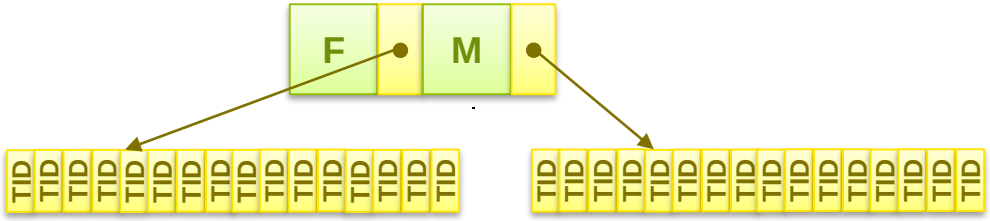
B-Baum	B*-Baum
Keine Redundanz	Schlüsselwerte teilweise redundant gespeichert
Lesen aller Sätze sortiert nach Schlüsselwert nur mit Verwaltung eines Stacks der max. Tiefe = Baumhöhe h	Kette der Blattknoten liefert alle Sätze nach Schlüsselwert sortiert.
Bei Einbettung der Datensätze geringe Verzweigungszahl ("Grad" oder "fan-out"), daher größere Höhe	Hohe Verzweigung in der inneren Knoten, daher geringere Höhe
Einige wenige Sätze (die in der Wurzel) werden mit einem Blockzugriff gefunden.	Für alle Sätze müssen h Blöcke gelesen werden.
	Schlüsselwerte in den inneren Knoten müssen nicht in den Datensätzen vorkommen (Optimierung beim Löschen von Sätzen).

B- und B*-Baum: Verfeinerungen

- **Die Di in den Knoten können sein:**
  - ein Datensatz
  - eine Liste von Datensätzen
    - Alle die mit gleichem Schlüsselwert
  - eine Satzadresse
    - Verweis auf Datensatz, der nach anderen Kriterien abgespeichert ist
  - eine Liste von Satzadressen
    - U.U. sogar von Sätzen verschiedenen Typs …
- **Die Di können variabel lang sein …**
  - Sowieso
  - Wie groß ist dann k?
  - Alles wird komplizierter, aber es geht – siehe Literatur

5.2 Bitmap-Index

- **Problem**
  - Am Beispiel: B-Baum auf Geschlecht bei Kundendatei mit 100.000 Sätzen resultiert in zwei Listen mit jeweils ca. 50.000 Einträgen



  - **Anfrage nach allen weiblichen Kunden erfordert 50.000 einzelne Block-Zugriffe**
    ⇒ **Sequenzieller Zugriff ist um Längen schneller**
- **Folgerung**
  - B-Bäume (und auch Hashing) sinnvoll für Suchschlüssel mit hoher "**Selektivität**" (also geringem Anteil passender Sätze an allen Sätzen einer Datei)
  - Faustregel:
    - Grenztrefferrate liegt bei ca. 5%.
    - Höhere Trefferraten lohnen bereits den Aufwand für einen Indexzugriff nicht mehr.

Grundlegende Idee der Bitmap-Indexstruktur

- **Idee**
  - In die Jahre gekommen ...
    - Eingesetzt schon in den 60er Jahren in Model 204 der Computer Corporation of America
  - Legt **für jeden Schlüsselwert eine Bitliste** an.
  - Jedem Satz der Datei ist ein Bit in der Bitliste zugeordnet.
    - Dafür notwendig: beliebige, aber feste Reihenfolge der Sätze
  - Bitwert 1 heißt:
    Der Schlüssel hat im Satz den Wert, zu dem die Liste gehört;
    0 heißt:
    Er hat einen anderen Wert.

ID	Geschlecht	Wohnort	Alter
1	M	Erlangen	jung
2	M	Erlangen	mittel
3	F	Forchheim	alt
4	M	Eckental	alt
5	F	Erlangen	jung
6	F	Erlangen	jung
7	M	Bamberg	mittel
8	F	Höchstadt	mittel
9	F	Forchheim	jung
10	M	Erlangen	Jung

Geschlecht

F	M
0	1
0	1
1	0
0	1
1	0
1	0
0	1
1	0
1	0
0	1

Größe von Bitmap-Indexstrukturen

- **Indexgröße: (Anzahl der Werte) × (Anzahl der Sätze) Bits**
  - Beispiel:
    - Schlüssel Geschlecht mit zwei Werten in Datei mit 10.000 Sätzen
    - Bitmap: 2 × 10.000 Bits = 20.000 Bits = 2.500 Bytes
    - Ggf. noch TID-Liste für Reihenfolge: 4 × 10.000 Bytes = 40.000 Bytes
- **Eigenschaften**
  - Wächst mit der Anzahl der möglichen Werte
  - Besonders interessant bis zu ca. 500 verschiedenen Werten
  - Bei kleinen Wertigkeiten (z.B. Geschlecht) nur sinnvoll, wenn entsprechender Schlüssel oft in **Konjunktionen** mit anderen indizierten Attributen auftritt (z.B. Geschlecht und Wohnort)
- **Nochmal Indexgröße:**
  - Nicht so problematisch, da gerade bei höherwertigen Schlüsseln die Bitmaps sehr dünn besetzt und Kompressionsverfahren (z.B. RLE) sehr gut einsetzbar sind.

Konjunktionen auf Bitmap-Indexstrukturen

- **Hauptvorteil von Bitmap-Indexen**
  - Einfache und effiziente logische Verknüpfbarkeit
  - Beispiel: Bitmaps B1 und B2 in Konjunktion

```
for ( i=0; i < B1.length; i++ )
  B = B1[i] & B2[i];
```

F		FO		jung		
0		0		1		0
0		0		0		0
1		1		0		0
0		0		0		0
1	AND	0	AND	1	=	0
1		0		1		0
0		0		0		0
1		0		0		0
1		1		1		1
0		0		1		0

- **Beispiel "junge Frauen aus Forchheim"**
  - Selektivitätsfaktor allg.: $1/2 \times 1/5 \times 1/3 = 1/30$
  - Annahme: 10.000 Sätze mit je 200 Bytes Länge (ca. 10 Sätze pro Block bei 2KB-Blöcken)
    - Sequenzieller Zugriff: 1.000 Blöcke
    - Bitmap-Zugriff: $10.000/30 \approx 334$ Sätze und damit Blöcke (worst case)

5.3 Zugriffsoperationen B-Baum / B*-Baum / Bitmap

- **Genau die gleichen wie bei Gestreuter Speicherung!**
  (Siehe oben)

```
void KeyedRecordFile::insert ( char *RecordBuffer,
  int RecordLength, char *KeyValue );
char *KeyedRecordFile::read ( char KeyValue,
  int *RecordLength )
void KeyedRecordFile::modify-key ( char *OldKeyValue,
  char *NewKeyValue )
```

  usw.

- **D.h. entscheidend ist Zugriff über einen Schlüssel**
  - Nicht die Realisierung über Hashing oder B-Baum oder Bitmap
- **Wieder eine weitere Art von Datenunabhängigkeit,**
  - **Meist als** Datenstruktur- **oder** Speicherungsstruktur-Unabhängigkeit **bezeichnet**

5.4 Primär- und Sekundär-Organisation

- **Primär-Organisation**
  - bestimmt Speicherung der Sätze selbst:
    - Entscheidet darüber, in welchem Block ein Satz abzulegen ist
  - kann sequenziell, direkt oder über Schlüssel sein
- **Sekundär-Organisation**
  - verweist nur auf die Sätze,
    die nach beliebigen anderen Kriterien abgespeichert wurden
    (in einer Primär-Organisation)
  - ist nur möglich, wenn die Primärorganisation den Direktzugriff auf einen einzelnen Satz unterstützt
    - Satzadresse oder eindeutiger Schlüssel, "**Satzverweis**"
- **D.h. genau eine Primär-Organisation pro Satzmenge, aber mehrere Sekundär-Organisationen möglich**
  - (Invertierungen, "Indexe")

Primär- und Sekundär-Organisation (2)

- **B-Baum/B*-Baum als Sekundär-Organisation**
  - Oben bereits dargestellt:
    In Di steht anstelle eines Satzes nur ein Satzverweis.

- **Auch Gestreute Speicherung als Sekundär-Organisation einsetzbar:**
  - In den Buckets dann nur (Schlüsselwert, Satzverweis)-Paare
  - Dann allerdings mindestens zwei Blockzugriffe beim Lesen
    - Sätze dafür unabhängig speicherbar

Organisation einer Menge von Sätzen

- **Aus der Sicht des Benutzers einer Datei:**
  - Unter den ***n*** Feldern der Sätze sind:
    - $p \in \{0, 1\}$ Felder, die für die Primär-Organisation benutzt werden
    - $k \in \{0, 1, \ldots, n\}$ Felder, über denen jeweils eine Sekundär-Organisation verwaltet wird
    - $n - k - p$ Felder, über denen kein Index verwaltet wird
- **Realisierung:**
  - Eine Datei für die Sätze selbst
    - Direktzugriff ($p = 0$) oder Schlüsselzugriff ($p = 1$)
  - Für jede Sekundär-Organisation eine eigene Datei
    - Schlüsselzugriff
    - Als "Sätze" nur Satzverweise gespeichert (mit genau dem Schlüsselwert, der in dem adressierten Satz zum Index gehört – Konsistenzregel!)
  - D.h. ***k* + 1 Dateien** für eine "Menge von Sätzen"

Einfügen und Suchen von Sätzen

- **Einfügen:**
  1. In die Datei der Sätze
     - Liefert Satzverweis
  2. In jede Index-Datei
     - das Paar (Schlüsselwert, Satzverweis)
- **Suchen:**
  1. **read** in der Datei der Sätze
  2. Über Index:
     - **read** im Index, liefert Satzverweise
     - In einer Schleife über alle Treffer: **read** in der Satzdatei
  3. Über nicht "indexierte" Felder:
     - **read-first** und Schleife mit **read-next** in der Datei der Sätze (Scan)

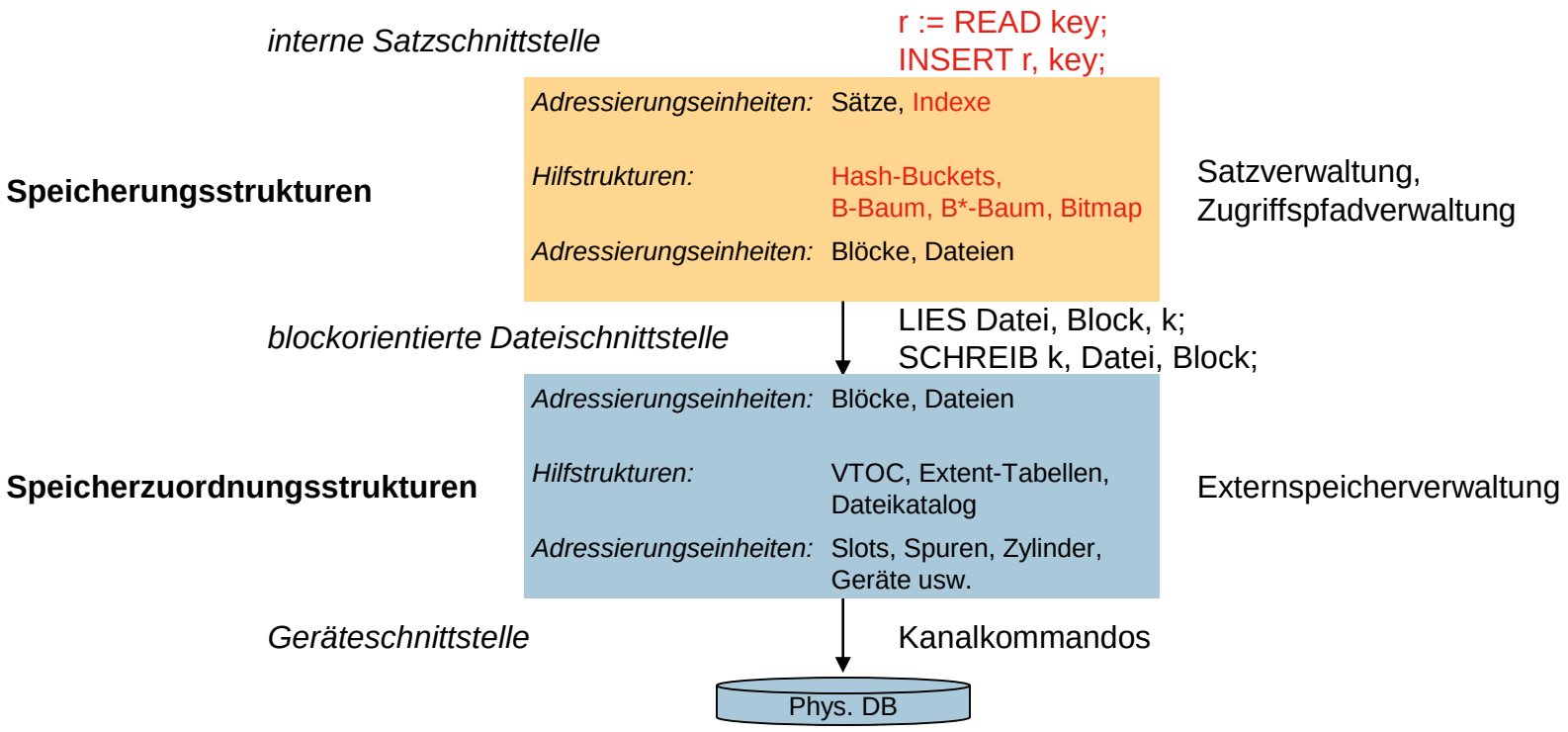
Eindeutige Schlüssel

- **Primärschlüssel**
  - Schlüssel, bei dem jeder Wert *in höchstens einem Satz* vorkommen darf
    - Kontonummer, Kundennummer, Auftragsnummer, Bibliothekssignatur u.v.a.
  - Eindeutigkeit kann bei Benutzung einer Schlüsselzugriffs-Datei (primär oder sekundär) einfach überprüft werden – sonst nur durch Sortierung!
  - **Sekundärschlüssel**: nicht eindeutig, darf in mehreren Sätzen gleich sein
- **Deshalb in Datenverwaltungssystemen oft zwei Varianten von jeder Satzorganisation verfügbar:**
  - Mit Duplikaten (Sekundärschl.) – ohne Duplikate (Primärschl.)
  - Falls ohne Duplikate:
    - Operation **read** liefert nur noch einen Satz, nicht ein Feld von Sätzen
    - Neuer Fehlerfall bei **insert**: Schlüsselwert schon vorhanden
- **Vorsicht:**
  - Index über Primärschlüssel muss nicht die Primär-Organisation sein!
  - Analog Sekundärschlüssel – Sekundär-Organisation

Weitere Möglichkeiten

- **Domänen-orientierte Indexstruktur**
  - Schlüssel mehrerer Satzmengen zusammen in einer Struktur
  - Schneller Zugriff bei Verknüpfungen (Join, siehe unten)
- **Mehrdimensionale Indexstrukturen**
  - Sätze über zwei oder mehr Schlüssel gleichzeitig zugreifbar
  - Soll auch helfen, wenn nicht alle davon als Suchschlüssel verwendet werden (partial-match queries)
  - Quadranten-Baum, Mehrschlüssel-Hashing, k-d-Baum, UB-Baum, Gridfile, R-Baum, …
    - Letztes Kapitel der Vorlesung "Multimedia-Datenbanken"

Resultierende Schicht



Vorlesung Implementierung von Datenbanksystemen

6. Puffer

Prof. Dr. Klaus Meyer-Wegener

Wintersemester 2019/20

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

6. Pufferverwaltung

- **Neben sequenziellem Zugriff nun auch direkter Zugriff auf Sätze**
  - Über Satzadresse oder Schlüssel
- **Im Hauptspeicher Platz für *n* Blöcke (*n* > 1)**
  - Hängt ab von Größe des Hauptspeichers, Zahl der Prozesse usw.
  - Größenordnung heute 10^4 – 10^6
- **(Buffer-) Frame**
  - Für Aufnahme eines Blocks vorgesehener Abschnitt des (virtuellen) Hauptspeichers
  - Deutsch: Pufferrahmen oder Kachel

6.1 Blockzugriffe

- **Zugriff auf Block *i***
  - Sog. **logischer Zugriff** (manchmal auch: "logische Referenz" des Blocks)
- **Zwei Fälle:**
  - Block bereits im Puffer
  - Block muss von Platte eingelesen werden (**physischer Zugriff**)
    - Auch **Einlagern** eines Blocks (in den Puffer) genannt
      - Eine E/A-Operation
- **Einlagern verdrängt einen anderen Block aus dem Puffer**
  - War geändert worden: muss auf Platte zurückgeschrieben werden
    - Noch eine E/A-Operation
  - War nicht geändert worden: kann direkt überschrieben werden

6.2 Ersetzungsverfahren

Welcher der *n* Blöcke im Puffer wird verdrängt?

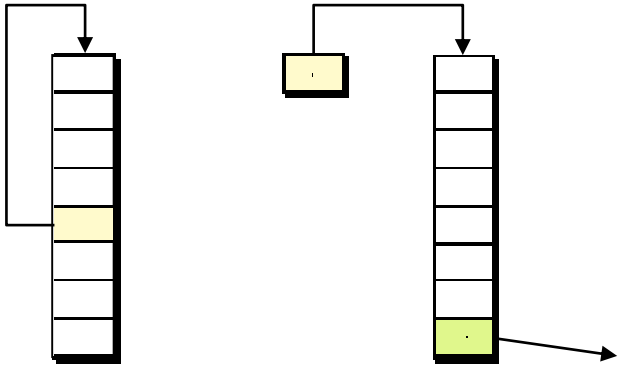
- **Ziel:**
  - Minimierung der Zahl der physischen Zugriffe bei gegebener Zahl von logischen Zugriffen
- **Deshalb:**
  - Häufig benutzte Blöcke (z.B. Wurzel B-Baum) im Puffer halten
- **Ersetzungsstrategie**
  - wählt den zu verdrängenden Block aus
- **Kriterien:**
  - "**Alter**" eines Blocks
    - Zahl der logischen Zugriffe (auf beliebige Blöcke) seit dem Einlagern
      - D.h. der logische Zugriff ist die Zeiteinheit
  - **Benutzungshäufigkeit** eines Blocks
    - Zahl der logischen Zugriffe auf diesen Block

Ersetzungsverfahren (2)

- **First in, first out (FIFO)**
  - **bewertet nur das Alter**:
    - Der Block, der am längsten im Puffer ist, wird ersetzt.
  - Ungünstig beim Direktzugriff:
    - Häufig benutzte Blöcke sollen ja gerade im Puffer bleiben – und dort "alt" werden.
- **Least frequently used (LFU)**
  - **bewertet nur die Häufigkeit**:
    - Der Block, auf den am seltensten zugegriffen wurde, wird ersetzt.
  - Sequenzielles Lesen heißt: Auf jeden Block wird genau einmal zugegriffen.
    - Kriterium nicht anwendbar!
  - Direktzugriff: Häufig genutzte Blöcke werden nun gehalten.
    - Aber: hat ein Block einmal eine hohe Häufigkeit auf sich versammelt, bleibt die, auch wenn inzwischen seit Minuten (Stunden?) nicht mehr auf ihn zugegriffen wurde!

Ersetzungsverfahren (3)

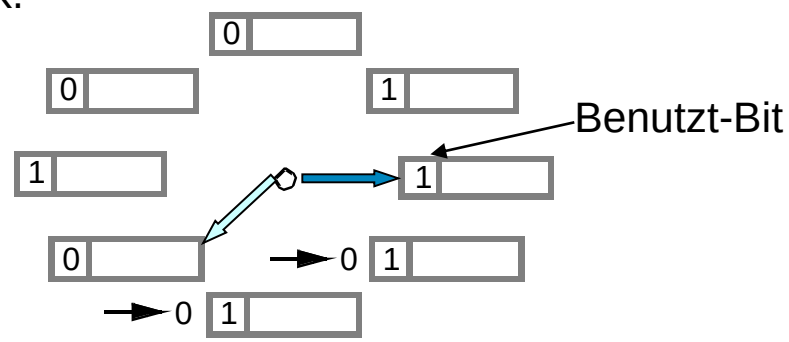
- **Least recently used (LRU)**
  - **bewertet das Alter seit dem letzten Zugriff**, nicht seit dem Einlagern
    - und damit indirekt auch die Häufigkeit
  - Wie? Quasi eine **verkettete Liste** aller Blöcke im Puffer:
    - Beim Zugriff auf einen Block im Puffer wird dieser Block ausgekettet und als erster Block wieder eingehängt.
    - Bei Verdrängung wird der letzte Block der Kette ersetzt (also der, auf den am längsten nicht zugegriffen wurde).
    - Ein eingelagerter Block kommt an den Anfang der Kette.
  - Veranschaulichung (Stapel):
  - Sehr gut, aber aufwändig!



1. Block im Puffer 2. Block einlagern

Ersetzungsverfahren (4)

- **CLOCK (Second Chance)**
  - Idee: LRU-Verhalten mit einfacher Implementierung erreichen
- **Prinzip:**
  - **Benutzt-Bit** eines Blocks im Puffer wird bei Zugriff auf 1 gesetzt.
  - Bei Verdrängung zyklische Suche mit dem **Auswahlzeiger**:
    - Falls Benutzt-Bit = 1, wird es auf 0 gesetzt.
    - Falls Benutzt-Bit = 0, wird Block ersetzt.
    - Zeiger wandert zum nächsten Block.



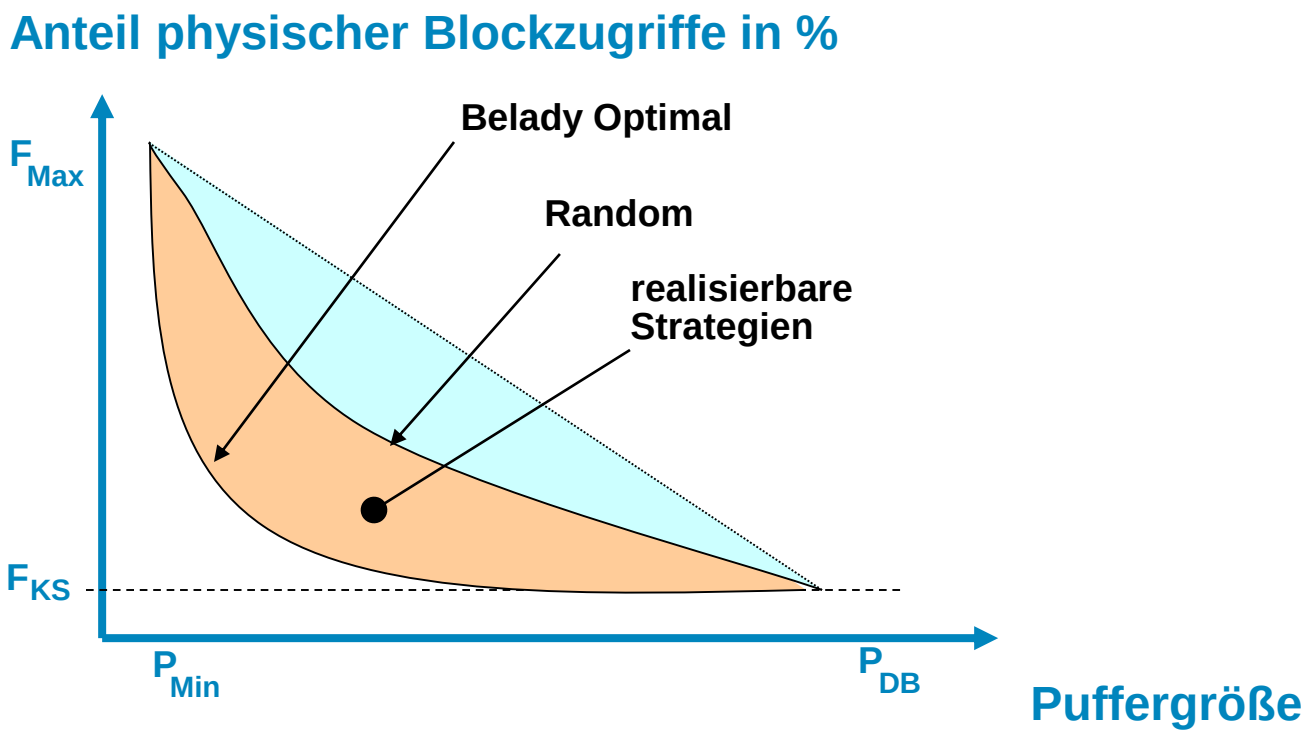
- **Charakterisierung:**
  - Jeder Block "überlebt" mindestens zwei Zeigerumläufe.

Ersetzungsverfahren (5)

- **Weitere Verfahren:**
  - G-CLOCK, LRD usw.
  - Versuchen ebenfalls, LRU-Qualität mit weniger Aufwand zu erreichen
  - Noch mehr Kriterien als nur ein Benutzt-Bit

- **Wichtige Restriktion:**
  - Blöcke, die noch benutzt (gelesen, geändert) werden, sind nicht ersetzbar.
    - Unterschied zum Paging in Betriebssystemen:
      - DBVS kennt die einzelnen Zugriffe auf einen Block im Puffer nicht.

Ersetzungsverfahren (6)



P_{min} = minimale Größe des Puffers (= 1)
P_{DB} = Datenbankgröße
F_{KS} = physische Blockzugriffe bei Kaltstart
(Jeder angesprochene Block muss zumindest einmal eingelesen werden.)

6.3 Schnittstelle einer Pufferverwaltung

- **Entscheidung für bestimmte Ersetzungsstrategie und Einbringstrategie**
  - vor Benutzern (höheren Software-Schichten) verbergen!
  - Dadurch ggf. änderbar
- **Einkapselung der Pufferverwaltung**
  - Zugänglich nur über folgende Operationen:

```
char *Buffer::fix ( BlockFile File, int BlockNo, char Mode );
```

    - Logischer Zugriff: stellt Block im Puffer zur Verfügung und liefert Anfangsadresse zurück
    - **Mode** gibt an, ob Block nur gelesen oder auch geändert werden soll
    - Name **fix** macht deutlich, dass Block vor Verdrängung geschützt ist, bis Bearbeitung abgeschlossen (einzelne Zugriffe auf Block im Puffer für Pufferverwaltung nicht wahrnehmbar)

```
void Buffer::unfix ( char *BufferAddress )
```

    - gibt Block im Puffer zur Ersetzung frei

Schnittstelle einer Pufferverwaltung (2)

- **Änderungen in der Satzverwaltung:**

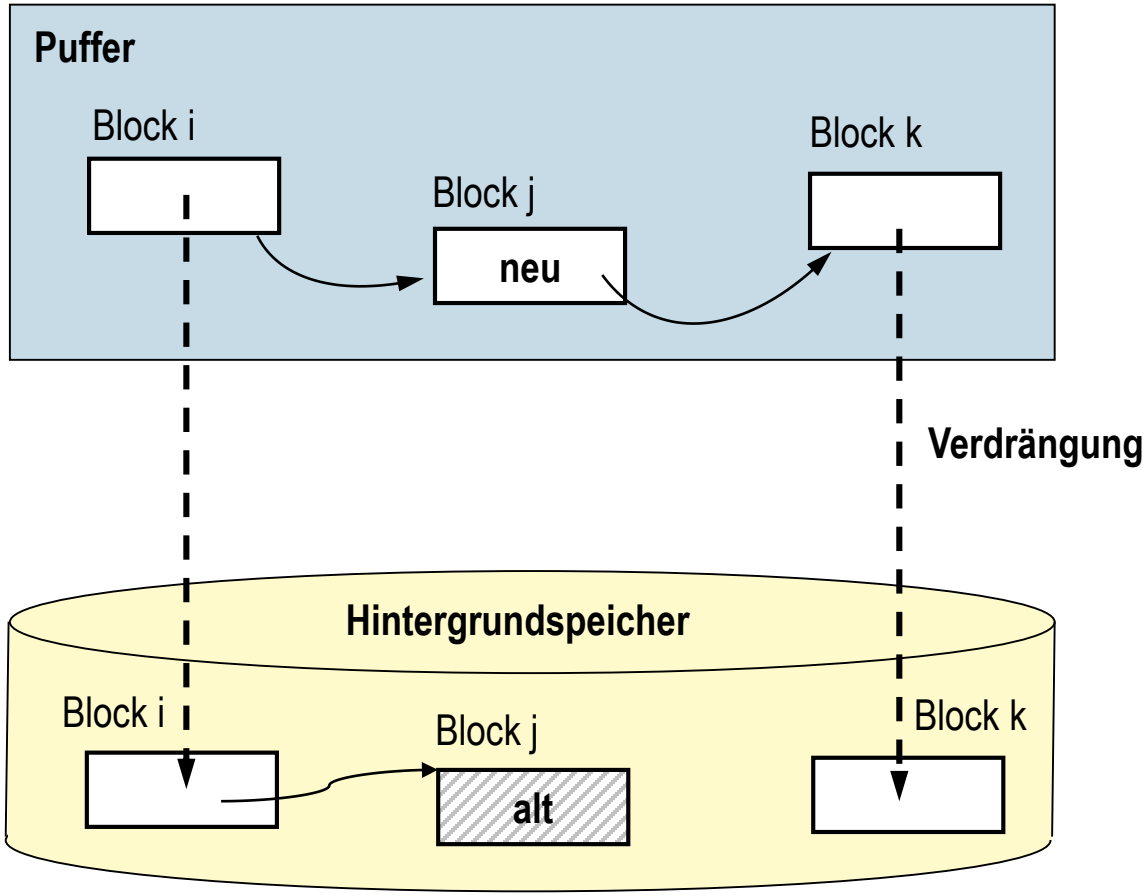
  (in allen Implementierungen von Zugriffsoperationen
  für sequenzielle, direkte und Schlüsselzugriffs-Dateien):
  - **read**(-Block)-Aufrufe
    - durch **fix**-Aufrufe ersetzen
  - **unfix-**Aufrufe
    - an geeigneter Stelle einfügen (wenn man mit der Bearbeitung eines Blocks fertig ist, z.B. vor dem Lesen des nächsten Blocks)
  - **write**(-Block)-Aufrufe
    - weglassen oder durch **unfix**-Aufrufe ersetzen
    - Block muss beim **fix** mit **Mode** = "w" geladen worden sein, wird dann von Pufferverwaltung bei Verdrängung geschrieben.
  - Auch ein leerer Block muss mit **fix** im Puffer bereitgestellt werden.
    - Pufferverwaltung liest ihn natürlich nicht von Platte ein, reserviert aber Kachel und führt ggf. auch Verdrängung durch.

6.4 Fehlersituationen

- **Betriebssystem-Absturz, Hardware-Fehler, Stromausfall:**
  - Hauptspeicher-Inhalt verloren – einschl. Puffer!
  - Ersetzungsstrategie hat entschieden,
    was auf der Platte steht (und damit vorhanden ist) und
    was noch im Puffer war (und damit verschwunden ist).
  - Blöcke auf der Platte wahrscheinlich inkonsistent!
    (alt und neu passen nicht zusammen)
- **Beispiel: B-Baum nach Splitt**
  - Neuer Knoten schon auf Platte geschrieben,
    aber übergeordneter Knoten dort noch der alte.
  - Neuer Knoten wird nicht beachtet!
    Seine Hälfte der Sätze fehlt plötzlich.
- **Muss von Hand mit hohem Aufwand bereinigt werden.**
- **Pufferverwaltung sollte Unterstützung bieten**

Fehlersituationen (2)

- **Veranschaulichung:**



Vorbeugemaßnahmen

- **Einbringstrategie:**
  - Bei Verdrängung
    alte Blockinhalte auf der Platte nicht überschreiben,
    sondern neue Blockinhalte in andere Slots schreiben.
  - Erst am Schluss
    (wenn alle neuen Blockinhalte auf der Platte sind)
    auf einen Schlag (ununterbrechbar) umschalten von alt auf neu.
    - Slots mit alten Blockinhalten dann freigeben und beim nächsten Mal selbst wieder für neue Blockinhalte verwenden.
- **Bei Fehler dann:**
  - Neue Blöcke ignorieren,
    dadurch alten (konsistenten) Zustand wiederherstellen und
    Programm nochmal laufen lassen.

6.5 Einbringstrategien

- **Einbringstrategien für Änderungen**
  - **Einbringen** = Ablegen auf einem nicht-flüchtigen Speicher derart, dass es nach einem Ausfall verwendet werden kann
  - Grundlegende Idee:
    - Speichern (beim Verdrängen aus dem Puffer) muss nicht notwendigerweise sofort ein Einbringen in den Datenbestand sein !
    - Einbringen kann auch verzögert stattfinden.
      - Nämlich erst dann, wenn auch noch andere, dazu gehörende Blöcke gespeichert worden sind.
- **Begriffliche Trennung in Block und Seite**
  - Seite = Block im Puffer (d.h. genau so groß)
  - Anwender (höhere Software-Schicht) arbeitet nur noch mit Seiten.
  - Flexible Zuordnung zu Blöcken
  - Insbesondere: *mehrere* Blöcke für eine Seite
    - Z.B. eben alter und neuer Inhalt derselben Seite

Segment

- **Linearer, logischer, potenziell unendlicher, tatsächlich aber endlicher Adressraum mit sichtbaren Seitengrenzen**
- **Entspricht der Datei**
  - Folge von Seiten
  - Einheit des Sperrens, der Wiederherstellung und der Zugriffskontrolle
  - Zuordnung zu Dateien: systemabhängig

    1:1 – alle Seiten eines Segments (und nur sie) in den Blöcken einer Datei gespeichert

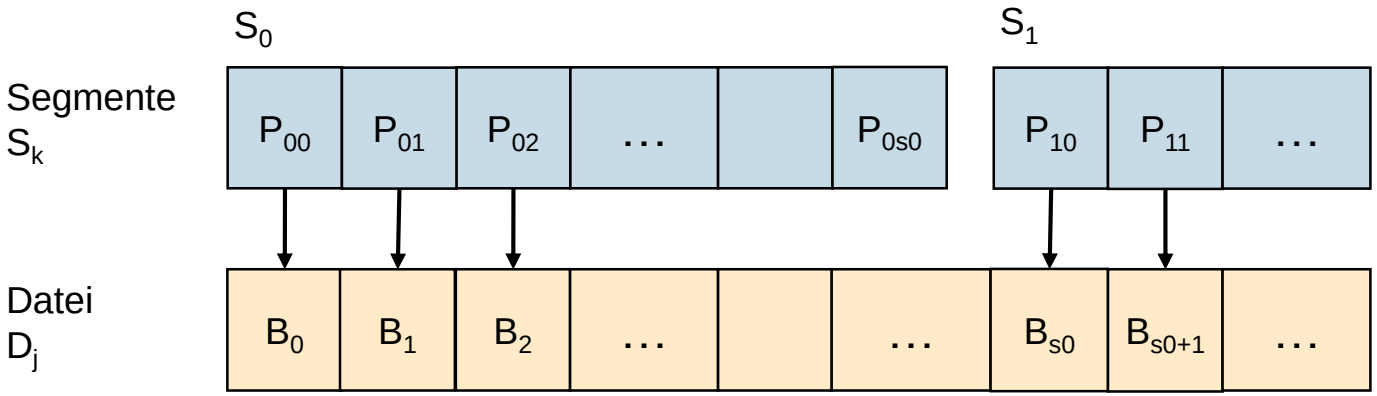
    N:1 – Seiten mehrerer Segmente zusammen in den Blöcken einer Datei gespeichert

    1:N – Seiten eines Segments in Blöcken unterschiedlicher Dateien gespeichert
- **Segmenttypen**
  - Selektive Einführung zusätzlicher Attribute
    - Katalog, Schema-Informationen, Log, alle gemeinsam benutzbaren Daten
    - Daten, die für bestimmte Benutzer oder Benutzergruppen reserviert sind
    - Private Kopien von Daten für einzelne Benutzer (Snapshots)
    - Hilfsdateien für Benutzerprogramme
    - Temporärer Speicher, z.B. für Sortierprogramme

Seitenzuordnung

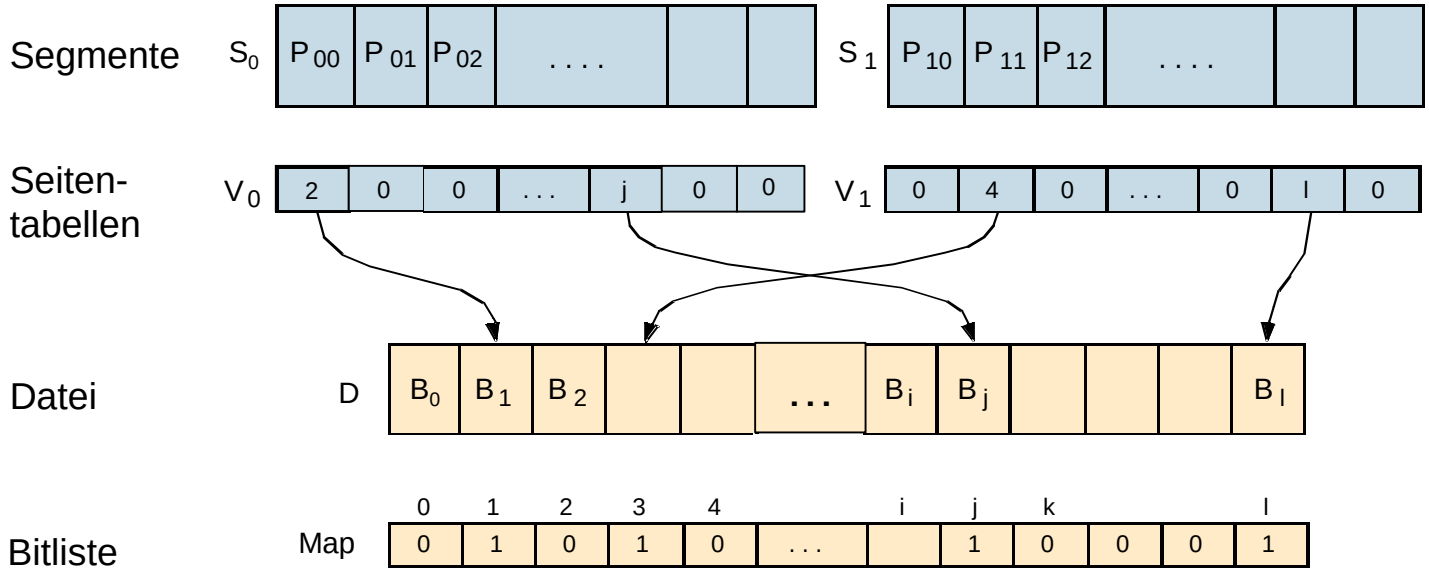
- **Aufgabe:**
  - Welche Blöcke für eine Seite?
- **Möglichkeiten:**
  - **Direkte** Seitenzuordnung
    - Aufeinander folgende Seiten werden auf aufeinander folgende Blöcke einer Datei abgebildet
    - Keine Hilfsstruktur benötigt, nur erste Blocknummer merken
  - **Indirekte** Seitenzuordnung
    - Mehr Flexibilität bei der Zuordnung zu Blöcken
    - Erfordert Hilfsstruktur (Array) mit Blocknummer zu jeder Seite

Direkte Seitenzuordnung



- **Eigenschaften**
  - Keine Fragmentierungsprobleme wegen der geforderten Übereinstimmung von Seiten- und Blockgröße ($L_k == L_j$)
  - Jeder Seite $P_{ki} \in S_k$ genau ein Block $B_{jl} \in D_j$ zugeordnet
  - Funktioniert natürlich nur für N:1- oder 1:1-Verhältnis von Segment (Datenbanksystem) und Datei (Betriebssystem)

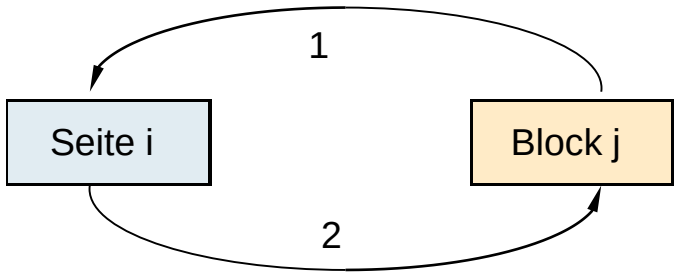
Indirekte Seitenzuordnung



- Für jedes Segment S_k existiert eine **Seitentabelle** V_k, die für jede Seite einen Eintrag (4 Bytes) mit der aktuellen Blockzuordnung enthält.

- Für die Datei D existiert eine **Bitliste** "Map", die ihre aktuelle Belegung beschreibt, d.h. die für jeden Block angibt, ob er momentan eine Seite enthält oder nicht (Freispeicher-verwaltung).

Direkte Seiteneinbringung



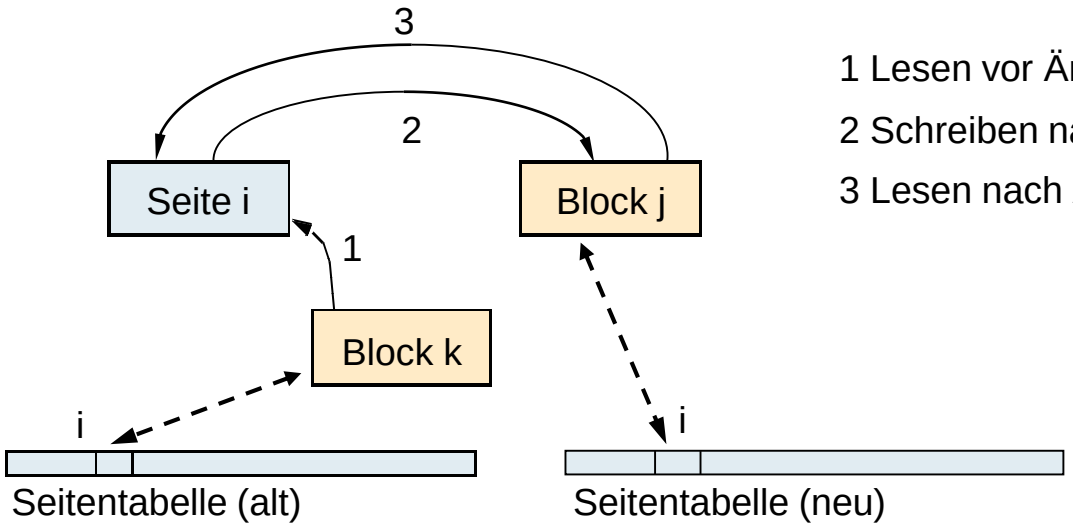
1 Lesen vor Änderung

2 Schreiben nach Änderung

- Beim Verdrängen aus dem Puffer *ersetzt* eine Seite genau den Block, aus dem sie beim Einlagern in den Puffer gelesen wurde ("**update in place**").
- **Vorteil:** Einfachheit; immer nur ein Block pro Seite
- **Nachteil:** Keine Unterstützung der Wiederherstellung nach Ausfällen
  - Der alte Zustand der Seite muss als Protokollinformation *vor* (!) dem Einbringen der geänderten Seite (und damit dem Vernichten des alten Seiteninhalts) auf einen sicheren Speicher geschrieben werden ("**W**rite-**A**head **L**og", **WAL**-Prinzip, s. unten).

Indirekte Seiteneinbringung

- Beim Verdrängen aus dem Puffer wird eine Seite *in einen freien Block* geschrieben. Der ursprüngliche Block bleibt unverändert.
- Auch nach einem Hauptspeicherverlust ist eine konsistente Datenbank in den alten Blöcken verfügbar!



1 Lesen vor Änderung

2 Schreiben nach Änderung

3 Lesen nach Änderung

Indirekte Seiteneinbringung (2)

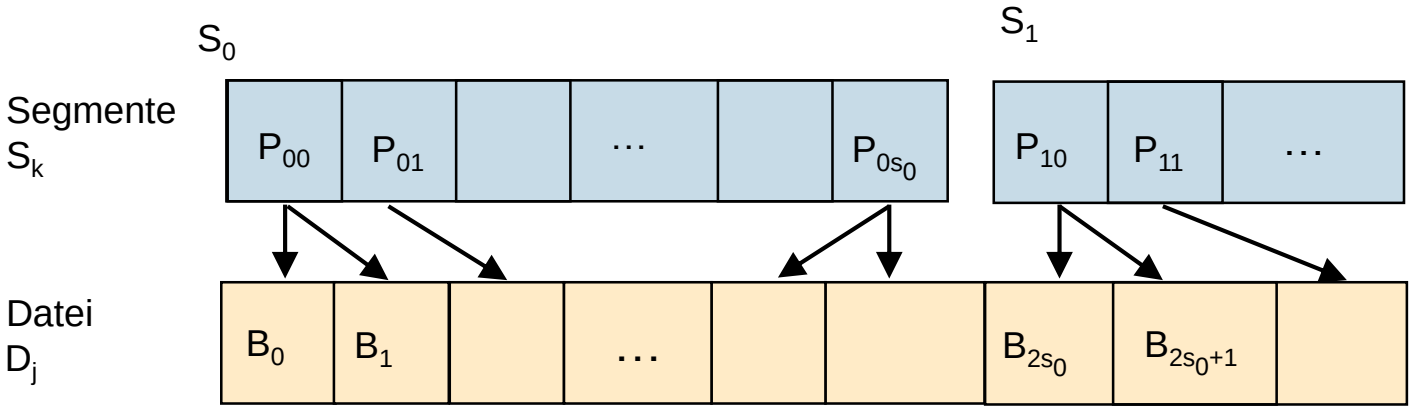
- **Noch zu klären: Wann und wie wird man die alten Blöcke los?**
  - Man muss wissen, wann alle zusammengehörenden Blöcke gespeichert wurden.
    - Beim Herunterfahren – falls es das gibt
    - Wenn gerade mal kein Programm läuft – falls es das gibt
    - Sicherungspunkt

- **Techniken:**
  - Schattenspeicher
  - Twin Slots
  - (Zusatzdatei)

Schattenspeicher – Prinzip

- **Benötigt indirekte Seitenzuordnung**
- **Inhalte aller Seiten eines Segments werden in einem Sicherungsintervall Δt in einem konsistenten Zustand unverändert gehalten.**
  - Sicherungspunkte (Endpunkte der Sicherungsintervalle) sind segmentorientiert,
    - wirken für alle Nutzer des Segments gemeinsam.
  - Sicherungspunkt besteht aus: belegten Seiten, Seitentabelle V_k, Bitliste M_j auf stabilem Speicher
  - Im Fehlerfall: segmentorientiertes Zurückgehen auf letzten Sicherungspunkt
- **In einem Segment S_k mit den Seiten P_{ki} $(0 \leq i \leq s_k - 1)$ sind $h_k \leq s_k$ Seiten belegt.**
  - Speicherbedarf z: $h_k \leq z \leq 2\, h_k$
- **Details im Anhang**

Twin Slots – Prinzip



- **Eigenschaften**
  - Direkte Seitenzuordnung
  - Doppelter Speicherplatzbedarf
  - Seite muss eine Art Versionsnummer enthalten, damit man weiß, welche die aktuellere ist.
  - Immer beide Blöcke lesen!
  - Dann bei Änderungen den älteren überschreiben

6.6 Zusammenfassung

- **Zweistufige Abbildung**
  - von Segment/Seite auf Datei/Block
    erlaubt Einführung von Abbildungsredundanz
    durch verzögertes Einbringen
- **Verzögertes (indirektes) Einbringen**
  - Teurer als direktes, besitzt jedoch implizite Fehlertoleranz
  - Geringe Kosten für Protokollieren und Wiederherstellung
  - Belastet den Normalbetrieb zugunsten der Wiederherstellung
    (Verwaltung der Seitentabellen)
  - Schwierige Implementierung
- **Direktes Einbringen ("update in place")**
  - Einfach zu implementieren
  - Keine zusätzlichen Kosten zur Ausführungszeit für die Seitenzuordnung
  - Fehlertoleranz nur durch explizite Protokollierung und Wiederherstellung
- **Verallgemeinerung: Transaktionskonzept (s. unten)**

Resultierende Schicht

interne Satzschnittstelle

r := READ <Satzadr>;
<Satzadr> := INSERT r;

Speicherungsstrukturen

Adressierungseinheiten:	Sätze, Indexe
Hilfsstrukturen:	Positionsindex, Freisp., Hash-Buckets usw.
Adressierungseinheiten:	Seiten, Segmente

Satzverwaltung, Zugriffspfadverwaltung

seitenorientierte Pufferschnittstelle

BEREITSTELLEN Segment, Seite;
FREIGEBEN Seite;

Seitenzuordnungsstrukturen

Adressierungseinheiten:	Seiten, Segmente
Hilfsstrukturen:	Seitentabelle, Blocktabellen
Adressierungseinheiten:	Blöcke, Dateien

Pufferverwaltung, Einbringstrategien

blockorientierte Dateischnittstelle

LIES Datei, Block, k;
SCHREIB k, Datei, Block;

Speicherzuordnungsstrukturen

Adressierungseinheiten:	Blöcke, Dateien
Hilfsstrukturen:	VTOC, Extent-Tabellen, Dateikatalog
Adressierungseinheiten:	Slots, Spuren, Zylinder, Geräte usw.

Externspeicherverwaltung

Geräteschnittstelle

Kanalkommandos

Phys. DB

Anhang

- **Lokalität**
- **Schattenspeicher im Detail**

Lokalität

- **Puffer kann nur wirken, wenn Lokalität bei den Seitenzugriffen (logischen Referenzen) vorliegt.**
  - Z.B. 80-20-Regel: 80% der Zugriffe entfallen auf 20% der Seiten.
- **Ungleichverteilung (Schiefe, Schräge, Skew)**
- **Kann man erwarten, weil einige Seiten (z.B. Wurzel eines B-Baums) häufiger referenziert werden müssten.**

- **Kann man es für einen gegebenen Seitenreferenz-String auch prüfen?**

Bestimmung der Lokalität

- **Working-Set-Strategie**
  - **Working-Set** ***W*(*t*, *n*)** := Menge der verschiedenen Seiten, die von einer Transaktion innerhalb der ***n*** letzten Referenzen, vom Zeitpunkt ***t*** aus rückwärts gerechnet, angesprochen wurden.
    - ***n*** = Fenstergröße
  - Working-Set-Größe ***w*(*t*, *n*)** := |*W*(*t*, *n*)|
- **Beispiel:**

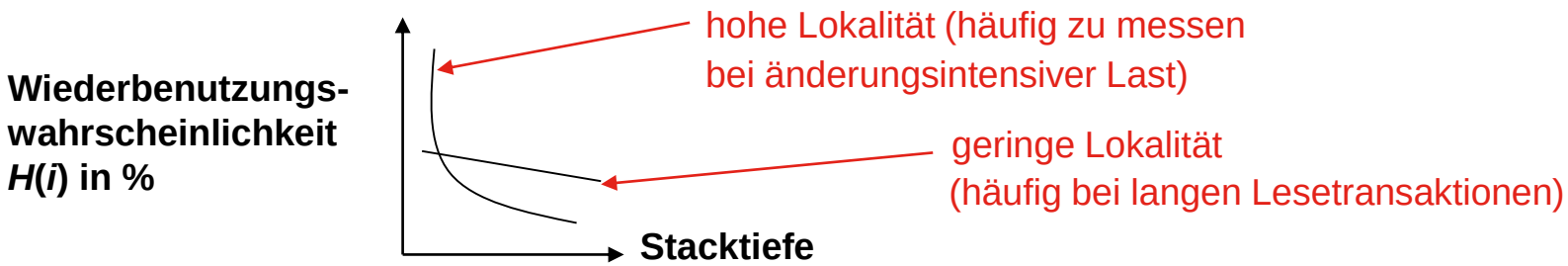
TA1:	A	A		C			A	A			A	G	H	
TA2:			B		D	E			E	F				F
								t1		t2			t3	

TA1: $W(t1, 5) = \{A, C\}$ $w(t1, 5) = 2$
TA2: $W(t2, 5) = \{B, D, E, F\}$ $w(t2, 5) = 4$
TA1: $W(t3, 5) = \{A, G, H\}$ $w(t3, 5) = 3$

- **Mittlere Working-Set-Größe *w*(*n*)**
- **Mittlere Working-Set-Lokalität *L*(*n*) = 1 – *w*(*n*) / *n* mit *n* > 1**

Bestimmung der Lokalität (2)

- **LRU-Stacktiefenverteilung**
  - LRU-Stack enthält alle im Intervall [1, *t*] angesprochenen Seiten einer Transaktion in der Reihenfolge ihres Alters
    - D.h. die zuletzt angesprochene ganz oben, darunter die unmittelbar vorher angesprochene usw.
  - Ermittlung der LRU-Stacktiefenverteilung (Wiederbenutzungshäufigkeit):
    - Für jede Position des LRU-Stacks wird ein **Zähler** geführt.
    - Wird die angesprochene Seite in der Stacktiefe *i* gefunden, so wird der *i*-te Zähler um 1 erhöht.
  - Nach Bearbeitung des Referenz-Strings gibt der *i*-te Zähler die Häufigkeit *H(i)* an, mit der Seite in Stacktiefe *i* wiederbenutzt wird.



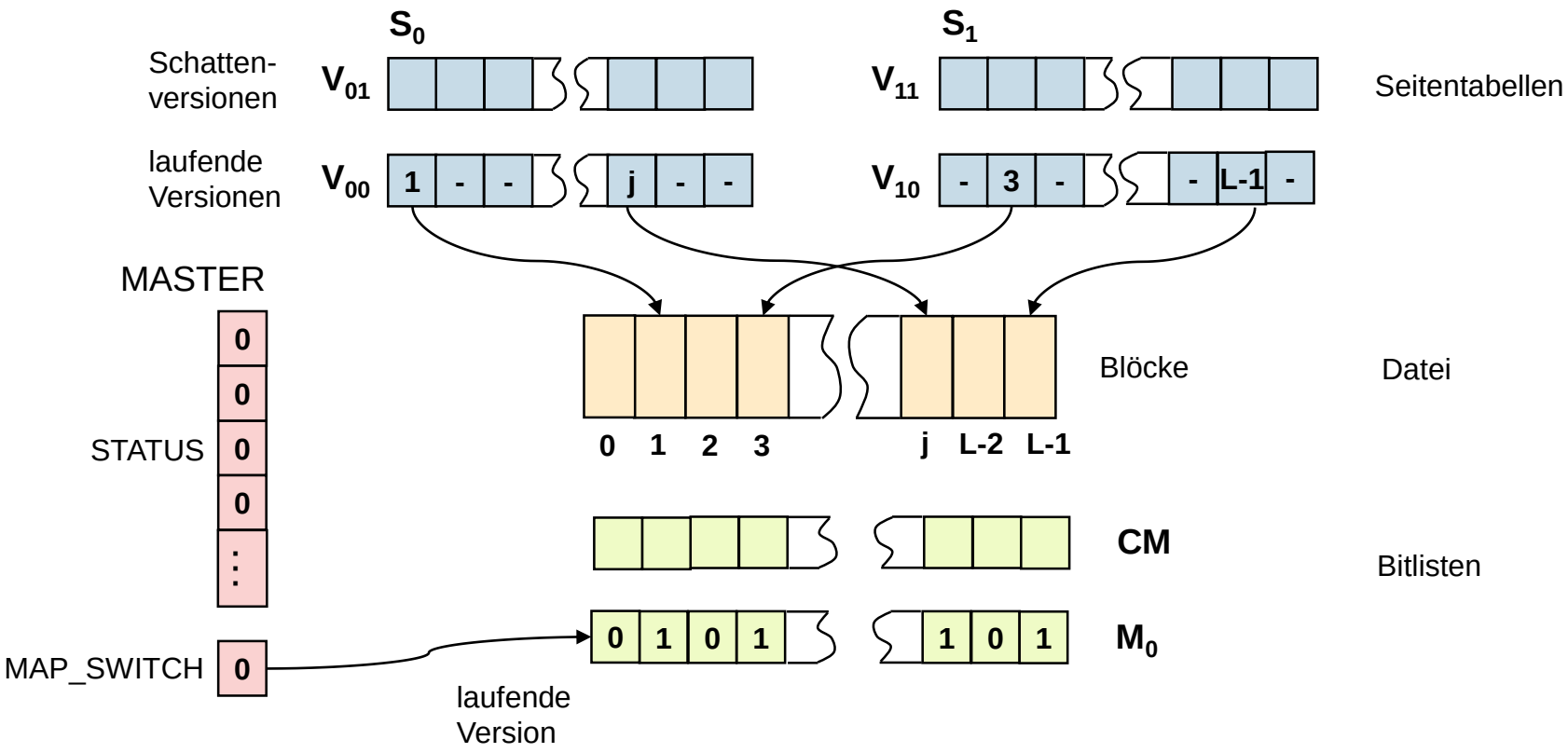
Bestimmung der Lokalität (3)

- **Page-Fault-Frequency-Strategie**
  - Basiert auf LRU-Stacktiefenverteilung
- **Verfahren:**
  - Wenn auch eine LRU-Ersetzungsstrategie benutzt wird, dann lässt sich aus *H(i)* die zu erwartende **Fehlseitenrate** *F* angeben.
  - Fehlseitenrate
    - *N* = Puffergröße bzw. Partitionsgröße
    - *D* = Zahl verschiedener Seiten im Referenzstring

$$F = \sum_{i=N+1}^{D} H(i)$$

Schattenspeicher – operational

- **Ausgangszustand: Segmente S_0 und S_1 geschlossen**



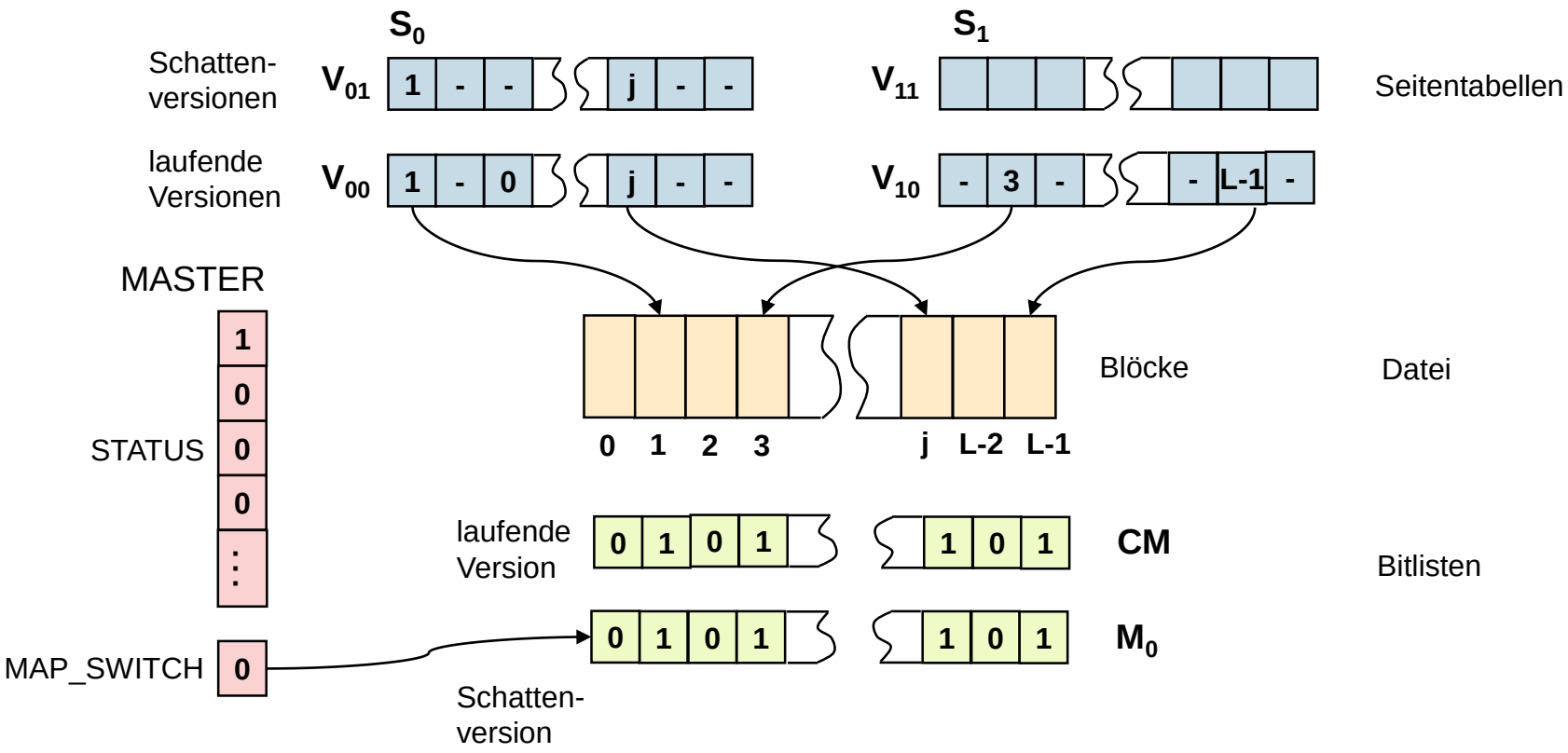
Schattenspeicher – operational (2)

- **Öffnen eines Segments S_k**
  - Kopieren von V_{k0} nach V_{k1} (auf der Platte)
  - STATUS[k] := 1
  - MASTER ununterbrechbar auf die Platte schreiben (z.B. doppelt)
  - Erzeugen von CM als Kopie von M_j ($j \in \{0, 1\}$ je nach MAP_SWITCH)

- **Nach Verlust des Hauptspeicherinhalts zeigt STATUS[k], dass V_{k1} gültig ist.**
  - V_{k0} dagegen i.Allg. unvollständig

Schattenspeicher – operational (3)

- **Segment S_0 für Änderungen geöffnet**

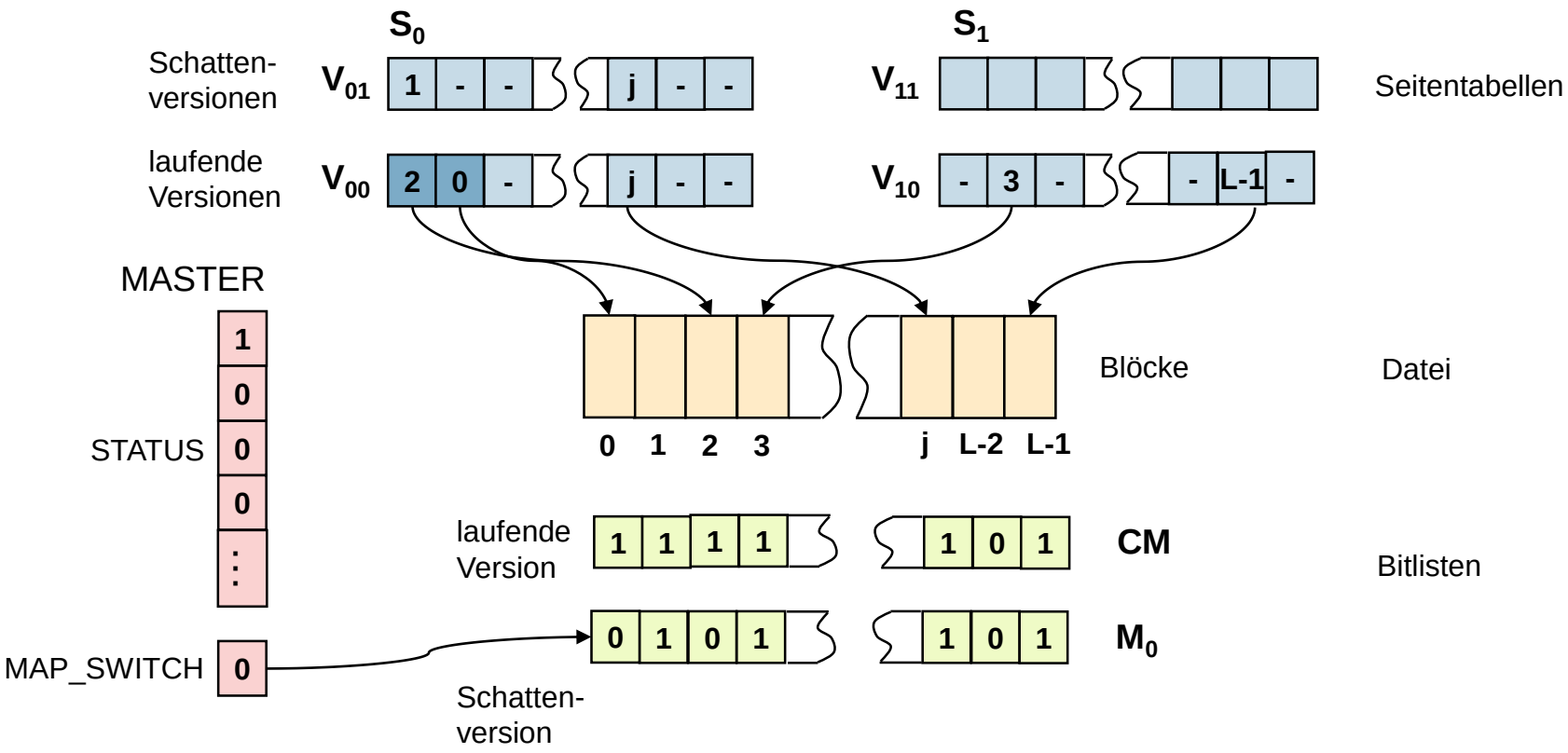


Schattenspeicher – operational (4)

- **Modifikation einer Seite:**
  Ausschreiben der Seite P_{ki} mit bisherigem Block $b = V_{k0}[i]$
  - Für P_{ki} muss ein neuer Block b' gefunden werden.
  - Suchen von b' mit $CM[b'] = 0$ (unbenutzter Block)
  - $CM[b'] := 1$
  - Setze $V_{k0}[i] := b'$
  - P_{ki} nach b' schreiben
  - Schattenbit in $V_{k0}[i]$ setzen
- **Block b mit altem Zustand der Seite P_{ki}**
  bleibt als "Schatten" über V_{k1} erreichbar.
- **Bei weiteren Änderungen wird P_{ki} immer wieder in Block b' geschrieben.**

Schattenspeicher – operational (5)

- **Seiten P_{01} und P_{02} geändert:**



Schattenspeicher – operational (6)

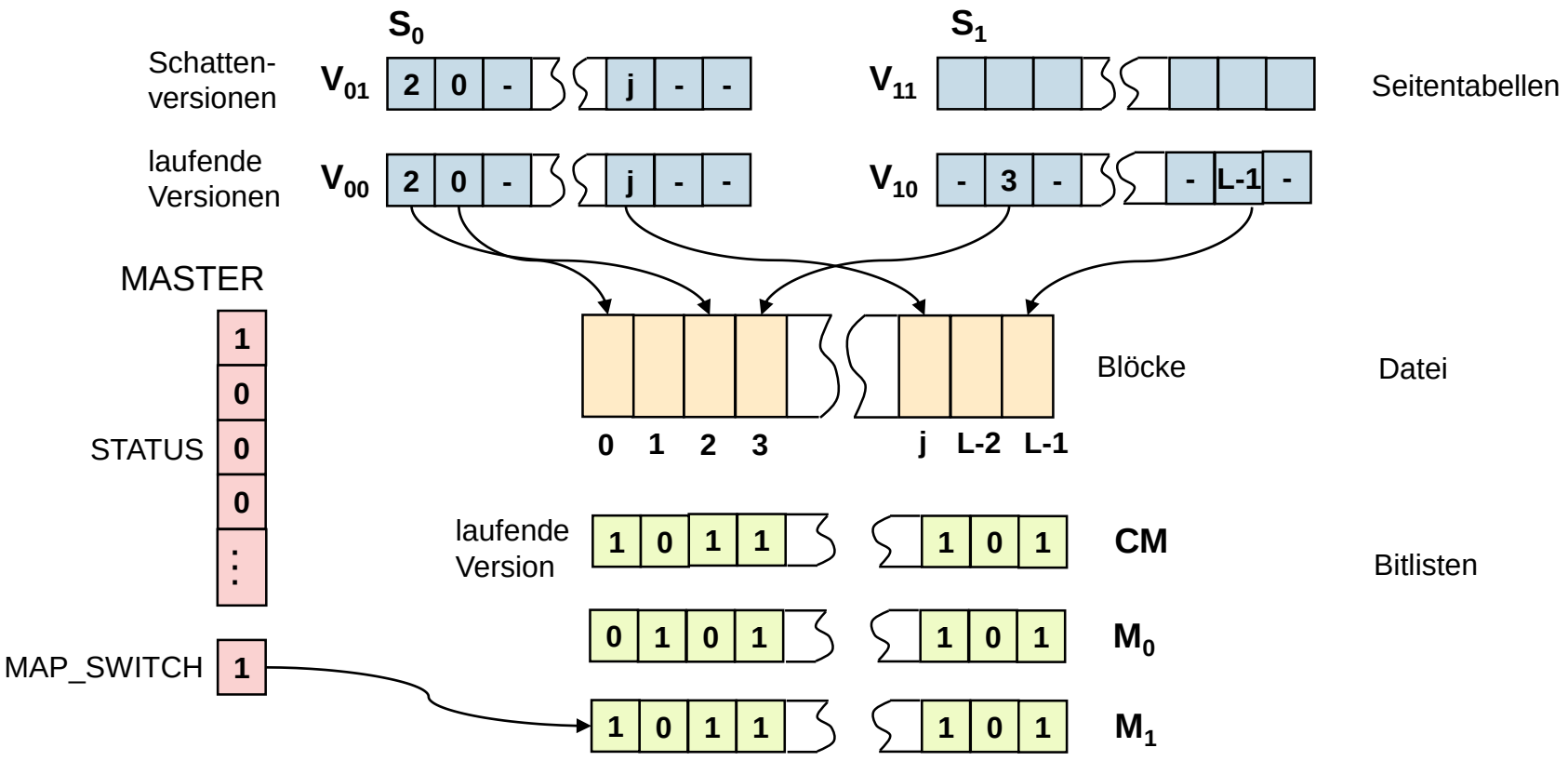
- **Sicherungspunkt für Segment S_k:**
  Nach Ablauf von Δt werden geänderte Seiten übernommen und Schattenseiten freigegeben.
  - M_1 als Kopie von M_0 erzeugen
    (oder umgekehrt, je nach MAP_SWITCH)
  - Für alle $V_{k0}[i] = b'$ mit gesetztem Schattenbit und $V_{k1}[i] = b$:
    - $M_1[b] := 0$; $M_1[b'] := 1$
    - V_{k0} und M_1 auf die Platte schreiben
    - STATUS[k] := 0
    - MAP_SWITCH := 1
    - MASTER ununterbrechbar auf die Platte schreiben (z.B. doppelt)
  - Segment S_k ist damit geschlossen.
- **Auf der Platte sind nun V_{k0} und M_1 als gültig verzeichnet.**

Schattenspeicher – operational (7)

- **Sicherungspunkt für Segment S_k (Forts.):**
  - Anschließend wird S_k gleich wieder geöffnet.
  - Für alle $V_{k0}[i] = b'$ mit gesetztem Schattenbit und $V_{k1}[i] = b$:
    - $CM[b] := 0$
    - Schattenbit löschen
    - V_{k0} nach V_{k1} kopieren (auf der Platte)
    - STATUS[k] = 1
    - MASTER ununterbrechbar auf die Platte schreiben (z.B. doppelt)

Schattenspeicher – operational (8)

- **Sicherungspunkt für S_0:**



Schattenspeicher – operational (9)

- **Zurücksetzen auf letzten Sicherungspunkt (bei laufendem System):**
  - Für alle $V_{k0}[i] = b$ mit gesetztem Schattenbit:
    - $CM[b] := 0$
    - $V_{k0}[i] := V_{k1}[i]$
- **Reparatur einer Datei nach Systemausfall:**
  - Für alle Segmente S_k mit STATUS[k] = 0 (geschlossen):
    - keine Maßnahme erforderlich
  - Für alle Segmente S_k mit STATUS[k] = 1 (offen):
    - V_{k1} nach V_{k0} kopieren
      - Zurück zur alten, konsistenten Version
    - STATUS[k] := 0
  - MASTER ununterbrechbar auf die Platte schreiben

Schattenspeicher – Bewertung

- **Rücksetzen auf einen konsistenten Zustand ist billig.**
- **WAL (s. unten) kann vermieden werden, um flexibleres Schreiben der Protokolldatei zuzulassen.**
- **Cluster-Eigenschaften von Blöcken gehen verloren.**
- **Bei großen Datenbanken werden Hilfsstrukturen (Seitentabellen V_k, Bittabellen M_j) leider zu aufwändig.**
  - Sie müssen in Blöcke zerlegt und durch einen speziellen Ersetzungsalgorithmus in einem eigenen Puffer verwaltet werden.
- **Konzept geeignet für kleinere DB; direktes Einbringen besser für größere DB**
  - Also eigentlich für alle …

Vorlesung Implementierung von Datenbanksystemen

7. Zugriff auf Datenbanken aus einem Programm

Prof. Dr. Klaus Meyer-Wegener

Wintersemester 2019/20

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

7.1 Relationale Datenbanken

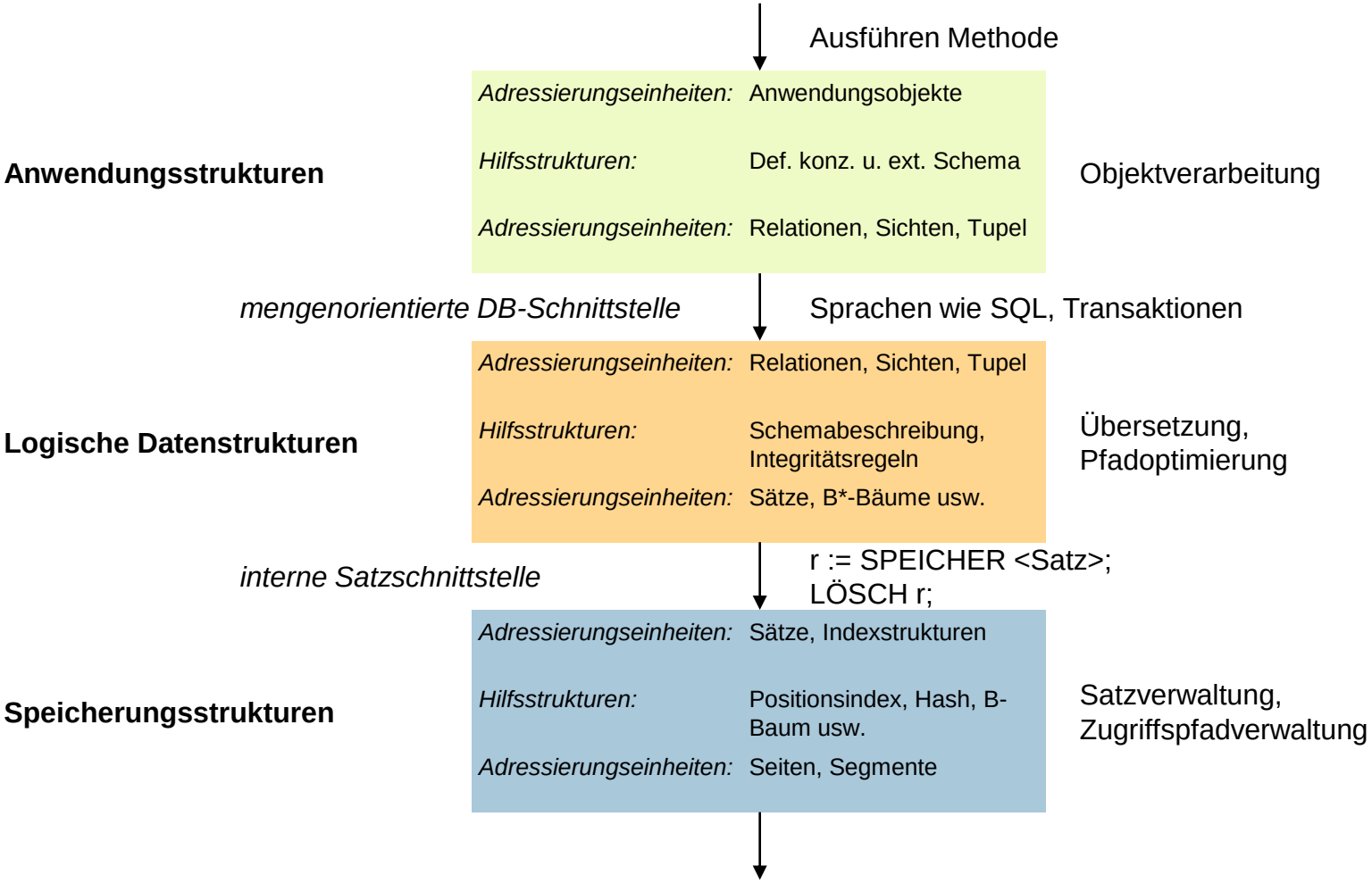
- **Bisher im Schichtenmodell erreicht:**
  Interne Satzschnittstelle
  - Benutzer (Programmierer) arbeitet beim Ablegen von Daten auf dem Hintergrundspeicher mit den "virtuellen" Objekten Segment (Datei) und Satz
  - **Segment** = Menge von Sätzen,
    Satz = variabel lange Folge von Bytes, evtl. mit Schlüssel

- **Operationen (Dienste):**
  - **Sequenzielles Lesen** aller Sätze (eines Typs, z.B. alle Kunden)
    - Reihenfolge durch Speicherungssystem festgelegt
  - **Direktzugriff über Satzadresse**
    - Wird vom Speicherungssystem vergeben
  - **Direktzugriff über Schlüssel** (= Teile der Sätze, Felder)
    - Wenn entsprechende Primär- oder Sekundär-Organisation vorhanden

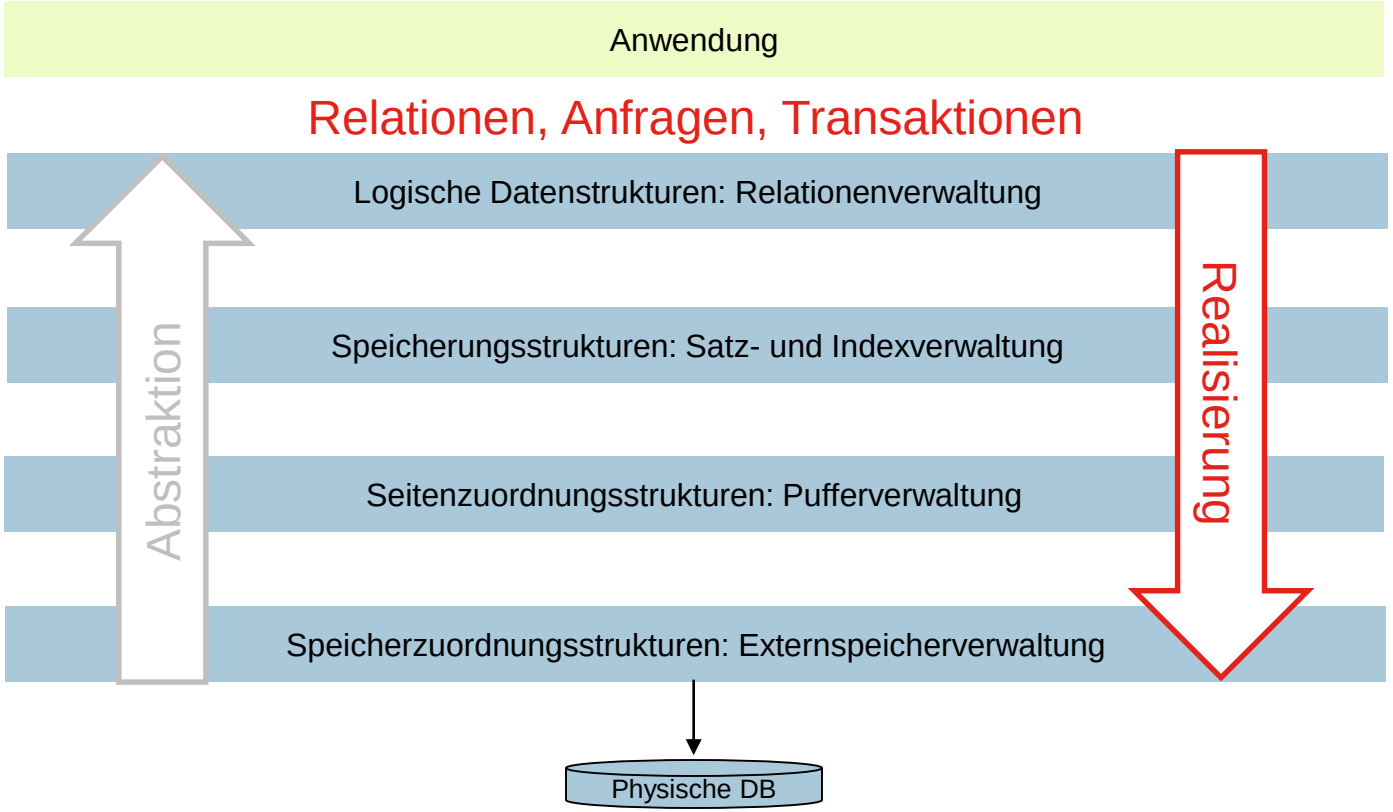
Relationale Datenbanken (2)

- **Nächster Schritt:**
  - Unabhängigkeit der Programme (Anwendungen)
    auch noch von Organisationsformen (Segmenttypen) und Indexen!
    - Sehen **nur noch Satzmengen**
    - **Direktzugriff über beliebigen Schlüssel immer möglich**
      - Speicherungssystem wählt intern Index aus oder sucht sequenziell
  - Noch mächtigere Operationen zur Auswertung:
    - Boolesche Ausdrücke zur Auswahl von Sätzen
    - Verknüpfung von Sätzen verschiedener Dateien
      (über gleiche Schlüsselwerte; Join)
- **Bedeutet weitere Abstraktion:**
  - Von Dateien zu **Relationen** (oder Klassen)
  - Von Sätzen zu **Tupeln** (oder Objekten)
  - Von Feldern (Schlüsseln) zu **Attributen**

Resultierende Schicht



Weitere Vorgehensweise



7.2 Programmzugriff

- **Ausführung von SQL-Anweisungen (Anfragen) in Programmen**
  (Hier am Beispiel von C)
  - Nicht Bestandteil der Programmiersprache
    - Durch **Vorübersetzer** (Precompiler) in normales C (hauptsächlich Funktionsaufrufe) transformiert
  - Daher spezielle Kennzeichnung im Programmtext:

```
EXEC SQL <SQL-Anweisung> ;
```

  - Ergebnisse einer SELECT-Anweisung müssen in **Programmvariablen** abgelegt werden:
    - INTO-Klausel
    - Kennzeichnung der Variablendeklaration und -verwendung für den Vorübersetzer (Typ-Überprüfung!)
    - Programmvariablen werden zur Unterscheidung von Attributen durch vorangestellten **Doppelpunkt** gekennzeichnet.
    - Auch in der WHERE-Klausel dürfen Programmvariablen verwendet werden!

Programmzugriff (2)


```
EXEC SQL BEGIN DECLARE SECTION;
    int anr;
    char nachname[31];  /* Platz für \0 */
EXEC SQL END DECLARE SECTION;

EXEC SQL SELECT Nachname, ANr
        INTO :nachname, :anr
        FROM Personen
        WHERE PNr = 3498;
```

- Zu Beginn des Programms:

```
EXEC SQL INCLUDE SQLCA;
```

  - Einkopieren einer Struktur-Definition für die **"SQL Communication Area":** Speicherbereich u.a. für Fehlermeldungen von SQL

Cursor

- SELECT liefert i.Allg. Menge, möglicherweise sehr groß
  - Kann nicht auf einen Schlag ins Programm übernommen werden
- Lösung: Tupel für Tupel abrufen

- **Definition eines Tupelzeigers (Cursors) für eine SELECT-Anweisung:**

```
EXEC SQL DECLARE c1 CURSOR FOR
     SELECT Nachname, ANr
     FROM Personen
     WHERE Gehalt < :limit;
```

  - Wird erst beim Öffnen des Cursors ausgeführt:

```
EXEC SQL OPEN c1;
```

Cursor (2)

- Abruf der einzelnen Tupel mit FETCH-Anweisung:

```
EXEC SQL FETCH c1 INTO :nachname, :anr;
```

  - Füllt die angegebenen Programmvariablen
    (die wie gehabt in einer DECLARE SECTION aufgeführt sein müssen)
- Nach letztem Tupel
  - führt weiteres FETCH zum Fehlercode 100
    in der Komponente SQLCODE der SQLCA.
- Cursor schließen:

```
EXEC SQL CLOSE c1;
```

  - Anschließend ggf. neu öffnen (z.B. mit neuem Wert in **limit**)

Beispielprogramm


```
main () {
   exec sql include sqlca;
   exec sql begin declare section;
        char pnr[8];
        char nachname[31];
        int anr;
   exec sql end declare section;
   exec sql declare c1 cursor for
        select PNr, Nachname, ANr from Personen;
   exec sql open c1;
   printf ("%-3s  %-20s  %-4s\n", "PNr", "Name", "Abt");
   for (;;) {
        exec sql fetch c1 into :pnr, :nachname, :nr;
        if (sqlca.sqlcode == 100) break;
        printf ("%-3s  %-20s  %4d\n", pnr, nachname, anr);
   }
   exec sql close c1;
}
```

Einschätzung

- **Genormt**
  - Portabilität der Anwendungsprogramme
- **Relativ kompakt zu programmieren**
  - Vergleiche nächsten Abschnitt …
- **Erlaubt die Erzeugung von Zugriffsmodulen**
  - SQL-Anweisungen schon zur **Übersetzungszeit** durch Vorübersetzer an DBS übergeben
  - Analysieren, optimieren, in Zwischencode überführen, im DBS ablegen
  - Zur Laufzeit dann sehr effizient:
    Nur noch ausführen
    - Aufruf über Namen:

```
call dbs (zugriffsmodul43, parameter);
```

- **Keine SQL Injection möglich**

7.3 Unterprogrammaufruf

- **Neben Spracherweiterung**
  - Wie eben gesehen
  - Wird auch "Eingebettetes SQL" (Embedded SQL) genannt
  - Durch Vorübersetzer (precompiler) in normales C transformiert
- **Auch noch Unterprogramm-Schnittstelle:**
  - **Call-level interface (CLI)** eines DBS
    - Bzw. Application-programming Interface (API) eines DBS
  - Prinzip:

```
call DBS ("select ... from ... where ... ");
```

  - Einfacher zu realisieren für den DBS-Hersteller,
    mehr Arbeit für den DB-Anwender

Beispiel JDBC

- **Call-level Interface für die Programmiersprache Java**
- **Angelehnt an ODBC**
  - "Open Data Base Connectivity";
    Microsoft-"Standard" für Datenbank-Zugriff
  - **Aber**: "JDBC" ≠ "Java Data Base Connectivity",
    kein Akronym, sondern geschützter Name ...
  - Übersichtlicher, einfacher (Nutzung von Klassen, streng typisiert)
- **Package java.sql**

java.sql

- **java.sql.DriverManager**
  - Treiber registrieren
    - implementieren eine gemeinsame Schnittstelle (abstrakte Klasse) auf jeweils einer bestimmten Datenbank
  - Verbindungen zur Datenbank aufbauen
- **java.sql.Connection**
  - verwaltet dann bestehende Verbindung zu einer Datenbank
- **java.sql.Statement**
  - Ausführung einer SQL-Anweisung über eine bestehende Verbindung
- **java.sql.ResultSet**
  - verwaltet Ergebnis einer Anfrage in Form einer Relation
  - Zugriff auf einzelne Attribute

JDBC am Beispiel

- **Prinzipieller Ablauf einer JDBC-Datenbankanwendung:**
  - Aufbau einer Verbindung zur Datenbank
  - Senden einer SQL-Anweisung
  - Verarbeiten der Anfrageergebnisse
- **Zuerst Laden des Treibers:**
  - `Class.forName ("oracle.jdbc.driver.OracleDriver");`
- **Verbindungsaufbau:**
  - Methode **getConnection** der Klasse **java.sql.DriverManager**
  - Argumente:
    - URL mit Verbindungsmechanismus und Treiber (muss geladen sein)
    - Benutzername und Passwort

```
String url = "jdbc:oracle:thin:@rechner1:1521:db1";
Connection con =
      DriverManager.getConnection (url, "jupp", "1%Ueg&");
```

JDBC am Beispiel (2)

- **SQL-Anweisung:**

```
String anfrage = "SELECT PNr, Nachname, ANr FROM Personen";
```

  - Anmerkung: SELECT * ist hier keine gute Idee – was käme da zurück?

```
Statement anweisung = con.createStatement ();
ResultSet ergebnis = anweisung.executeQuery (anfrage);
```

- **Verarbeitung des Ergebnisses:**
  - Die meisten Programmiersprachen (so auch Java) können Mengen nicht direkt verarbeiten.
  - Deshalb **Iteration** (Schleife):
    - Nacheinander jedes einzelne Tupel der Ergebnismenge
    - Methode **next** der Klasse **ResultSet** positioniert auf das nächste Tupel.
  - Zugriff auf Spalten (Attribute einer Relation) mit typspezifischer **get-**Methode

```
while (ergebnis.next ()) {
  int pnr = ergebnis.getInt (1);
  String name = ergebnis.getString (2);
  int anr = ergebnis.getInt (3);
  System.out.println (pnr + " " + name + " " anr); }
```

JDBC am Beispiel (3)

- **Am Schluss Ressourcen freigeben:**

```
ergebnis.close ();
anweisung.close ();
```

  - Optional
    - Freigabe auch durch automatische Speicherbereinigung von Java
  - Allerdings guter Programmierstil
- **Fehler**
  - werden in JDBC als Ausnahme der Klasse **SQLException** signalisiert
  - Mit einem **try-catch**-Block abfangen
  - Details zum Fehler mit **getMessage** ermitteln

```
try {
  // Aufruf von JDBC-Anweisungen
  } catch (SQLException exc) {
          System.out.println ("SQLException: " +
                                  exc.getMessage () );
  }
```

Ausführen von Anweisungen

- **Schnittstelle "Statement"**
  - Drei **execute**-Methoden:

```
ResultSet executeQuery (String sql)
    throws SQLException;
```

    - Siehe oben
    - Lesende Anfrage (SELECT)

```
int executeUpdate (String sql)
    throws SQLException;
```

    - Ändernde Anweisung (CREATE TABLE, INSERT, DELETE, UPDATE)
    - Liefert die Anzahl der geänderten Tupel zurück

```
boolean execute (String sql)
    throws SQLException;
```

    - Für Anweisungen, die mehrere Ergebnisse liefern (gespeicherte Prozeduren)

Vorübersetzte Anfragen

- **Mehrfache Ausführung von Anfragen**
  - Evtl. mit unterschiedlichen Parametern (in Programmvariablen)
- **Schnittstelle PreparedStatement**
  - Von **Statement** abgeleitet
  - Schon beim Erzeugen SQL-Anweisung angeben:

```
PreparedStatement einfuegen =
  con.prepareStatement ("INSERT INTO Personen " +
     "VALUES (?, ?, ?, ?, ?, ?, ?)");
```

  - Wird sofort analysiert, übersetzt, optimiert, in Zwischencode überführt, ...
  - Vor Ausführung dann noch Parameterwerte (für die Fragezeichen) setzen:

```
\\ java.sql.PreparedStatement
void setBoolean (int paramIdx, boolean b)
  throws SQLException;
void setString (int paramIdx, String s)
  throws SQLException;
```

Vorübersetzte Anfragen (2)

- **Schnittstelle PreparedStatement (Forts.)**
  - Parameter von 1 an durchnummeriert

```
einfuegen.setInt (1, 8967);
einfuegen.setString (2, "Karl Numpf");
einfuegen.setString (3, "Wunsdorf");
einfuegen.setInt (4, 57);
...
```

  - Anschließend wie gewohnt ausführen

```
einfuegen.executeUpdate ();
```

  - Alle Methoden von **Statement** verfügbar, allerdings überschrieben: ohne Parameter
  - Dann Parameter neu setzen und nochmal ausführen

- **Wichtige Optimierung**
  - Erheblicher Anteil der Verarbeitung nur einmal beim Erzeugen

Gespeicherte Prozeduren

- **Weitere Optimierung**
  - Vorübersetzte Anfragen müssen immer noch bei jedem Programmlauf neu analysiert, übersetzt und optimiert werden.
  - Idee: Anfrage in der Datenbank ablegen, mit einem Namen und Parametern versehen
    - Unabhängig von den Anwendungsprogrammen
    - Die rufen die Anfrage nur noch unter ihrem Namen auf.
  - Analyse und Optimierung dann nur noch einmalig
  - "Stored Procedures"
- **Schnittstelle CallableStatement**
  - Abgeleitet von PreparedStatement
  - Objekt erzeugen über Methode **prepareCall** von **Connection**

```
CallableStatement call =
      con.prepareCall ("{ call prozedur1 }");
```

  - Sog. Escape-Syntax (geschweifte Klammern); wird von JDBC-Treiber übersetzt in die spezielle Syntax eines Datenbanksystems

Gespeicherte Prozeduren (2)

- **Parameter**
  - Wieder mit Fragezeichen als Platzhalter:

```
{ call prozedur2 (?, ?) }
{ ? = prozedur3 (?) }
```

  - Alle **set**-Methoden von **PreparedStatement** verfügbar; nutzbar für **in**-Parameter
  - Für **out**- **und inout-**Parameter vor der Ausführung JDBC-Typ festlegen:

```
void registerOutParameter (int paramIdx, int jdbcType)
   throws SQLException;
```

  - Beispiel:

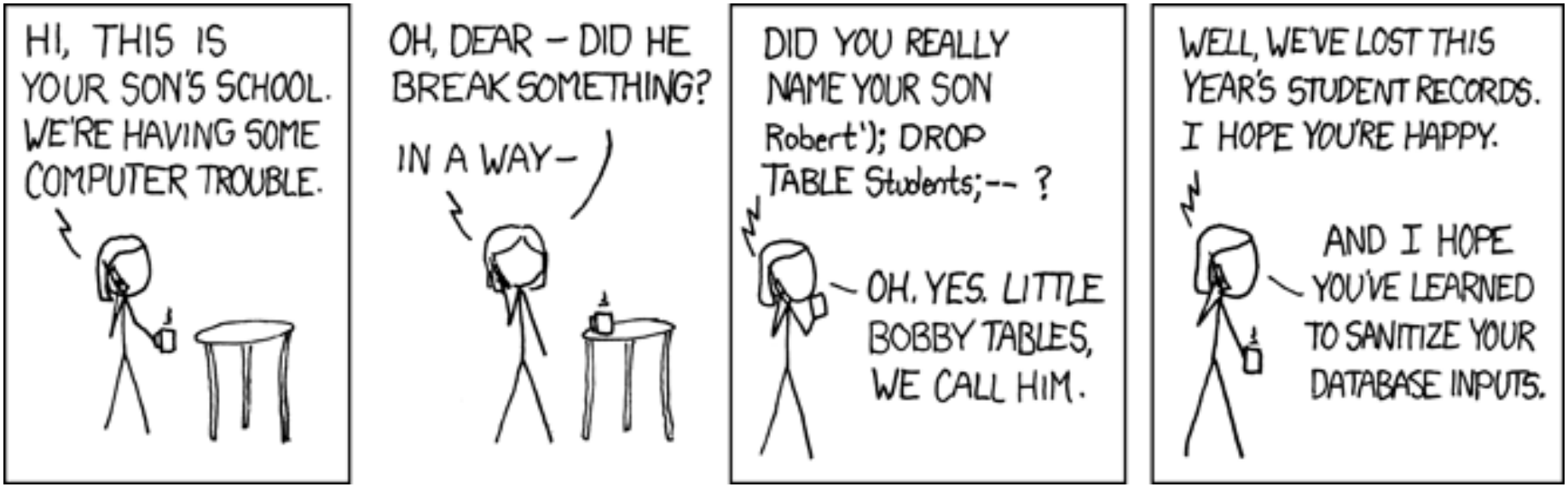
```
CallableStatement aufruf =
   con.prepareCall ("{ call Test (?, ?) }");
aufruf.setDouble (2, 42.0);
aufruf.registerOutParameter (1, java.sql.Types.VARCHAR);
aufurf.registerOutParameter (2, java.sql.Types.FLOAT);
```

  - Lesen der **out**- und **inout**-Parameter mit **get**-Methoden

Einschätzung

- **Relativ elementar**
- **Programmierer muss viel selbst machen**
- **Optimierung explizit:**
  - Direkter Aufruf oder
  - vorübersetzte Anfrage oder
  - Nutzung einer gespeicherten Prozedur
  - Wechsel nur durch Umprogrammierung!
- **Vorsicht mit SQL Injection!**
  - Wo kommt die Zeichenkette mit der SQL-Anfrage her? Direkt von der Eingabe?

SQL Injections



Quelle: http://xkcd.com/327/

7.4 Vergleich

- **Eingebettetes SQL**
  - Erweiterung der Programmiersprache: Vorübersetzer oder Extra-Compiler
  - Bequemer für Programmierer (kompakter)
  - Typüberprüfung und Optimierung schon zur Übersetzungszeit
    - Und das heißt: nur einmal
- **Unterprogrammaufruf**
  - Programmiersprache (auch "Wirtssprache" – host language – genannt) bleibt unverändert
  - Parameter für die Unterprogramme:
    alle Teile der SQL-Anweisungen einzeln
    oder SQL-Anweisung insgesamt als Zeichenkette – umständlich!
  - Typüberprüfung und Optimierung erst zur Ausführungszeit
    - Und das heißt: immer wieder

7.5 O/R-Mapping

- **Besonderheit für objektorientierte Programmiersprachen**
- **Objekte**
  - nicht selbst "zerlegen" und in Tupeln abspeichern.
  - Durch Werkzeuge erledigen lassen!
- **Konfigurationsdateien (in XML)**
  - sagen, welche Teile eines Objekts in welcher Relation gespeichert werden
- **Transfer der Daten zwischen Anwendung und Datenbank**
  - erfolgt automatisch,
  - d.h. ohne dass Anwendungscode dafür zu schreiben wäre!

Beispiel Java Persistence API

- **EJB 3.0**
- **Annotationen**

```
package  … ;
import javax.persistence.*;
@Entity
public class Personen implements java.io.Serializable
{
     private int PNr;
     private String Nachname;
     private int ANr;
     @Id
     public int getId( ) { return PNr; }
     public void setId(int PNr) { this.PNr = PNr; }
     …
}
```

Beispiel Java Persistence API (2)

- **Generiertes SQL:**

```
CREATE TABLE Personen (
     PNr            INT primary key,
     Nachname       VARCHAR(255),
     ANr            INT
);
```

  - sowie INSERT, UPDATE, DELETE usw.

- **Mehr dazu**
  - in der Vorlesung "Objektorientierte Datenbanken"

Vorlesung Implementierung von Datenbanksystemen

8. Transaktionen

Prof. Dr. Klaus Meyer-Wegener

Wintersemester 2019/20

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

- **Umgang mit Fehlern und Ausfällen**
- **Programmfehler – Datenbank-Anwendungsprogramm stürzt ab:**
  - Ist fehlerhaft:
    - Division durch Null, unzulässige Adressierung (Exception), ...
  - Wird von außen abgebrochen:
    - Überschreiten einer CPU-Zeit-Vorgabe
    - Sättigungszustand im System
    - Eingriff der Operator (cancel, kill)
- **Ergebnis:**
  - Hauptspeicherinhalte des Programms (Programmvariablen = Programmzustand) verschwunden
  - Daten im Puffer und auf Platte, die das Programm zu ändern begonnen hatte, in irgendeinem unbekannten Zwischenzustand (unvollständig)

Programmfehler

- **Notwendige Maßnahmen:**
  - Daten des Programms, die noch im Puffer sind, wegwerfen
  - Datenbestand auf Platte muss von Hand bereinigt werden, bevor das nächste Programm darauf zugreifen darf.
    - Programmierer müssen Programmlauf nachvollziehen und Änderungen der Daten einzeln rückgängig machen!

- **Aufwand! Zeit! Daten blockiert für andere!**

Systemfehler

- **Systemfehler:**
  Datenbankverwaltungssystem (DBVS) fällt aus oder
  Betriebssystem (BS) fällt aus oder
  Hardware-Fehler oder
  Stromausfall oder ...
  - Programm ist (meistens) unschuldig.
    - War evtl. sogar schon fertig,
      d.h. alle Änderungen vollständig ausgeführt – aber leider bisher nur im Puffer des BS oder DBVS abgelegt
- **Ergebnis:**
  - Pufferinhalte sind verlorengegangen.
  - Daten auf Platte in unbekanntem Zustand
    - Muss vor Neustart irgendwelcher Programme bereinigt werden

Systemfehler (2)

- **Notwendige Maßnahmen:**
  - *Alle* Programme, die gerade liefen oder kurz zuvor fertig geworden waren, von Hand nachvollziehen!
  - Klären, welche Programm ihre Ergebnisse vollständig auf die Platte bringen konnten (also abgehakt werden können) und welche nach einer Bereinigung wiederholt werden müssen.

- **Noch viel größerer Aufwand!**

Gerätefehler

- **Gerätefehler: Platte defekt**
  - Z.B. Headcrash; nichts mehr lesbar! Oder Diebstahl ...
- **Ergebnis:**
  - Programme, DBVS und BS können nicht mehr weitermachen.
  - Änderungen auf Platte verloren
    - Die ggf. noch im Hauptspeicher verfügbaren daher sinnlos (ohne Basis)
- **Notwendige Maßnahmen:**
  - Neue Platte formatieren
  - Sicherungskopien (Backup) einspielen
  - Alle Änderungen seit Anlegen der Sicherungskopie wiederholen
    - Also seit gestern Abend, letzter Woche, Anfang des Monats, ...
  - Alle Programme mit gleichen Eingabedaten (!) und in der gleichen Reihenfolge (!) nochmal ablaufen lassen
- **Wiederum: Aufwand, Zeit, Daten blockiert**

Erwünschte Zustände der Daten auf Platte

- **Physische Konsistenz**:
  - Korrektheit der Speicherungsstrukturen
  - Alle Verweise und Adressen (TIDs) stimmen.
  - Alle Indexe sind vollständig und stimmen mit den Primärdaten überein.
    - Kein Schlüsselwert eines neuen oder geänderten Satzes fehlt.
    - Kein Schlüsselwert eines gelöschten oder geänderten Satzes ist übriggeblieben.
- **Logische Konsistenz**:
  - Korrektheit der Dateninhalte
    - Stellen einen (möglichen) Zustand der realen Welt dar
  - Alle Bedingungen des Datenmodells (Primärschlüsseleigenschaft, Referenzielle Integrität) und alle benutzerdefinierten Integritätsbedingungen (Assertions) sind erfüllt.

Erwünschte Zustände der Daten auf Platte (2)

- **Annahmen:**
  - Alle vollständig ausgeführten **Datenbank-Operationen** (z.B. SQL-Anweisungen) hinterlassen einen **physisch konsistenten** Zustand.
    - Soll heißen: Das DBVS selbst ist nicht fehlerhaft.
  - Alle vollständig ausgeführten **Anwendungsprogramme** hinterlassen einen **logisch konsistenten** Zustand.
    - Soll heißen: Programmiererin hat alles richtig gemacht.
    - Dies wird im folgenden noch differenziert; es kann auch nach einem *Teil* eines Anwendungsprogramms schon wieder ein konsistenter Zustand erreicht sein.
- **Nach einem Fehler:**
  - Die Daten sind i.Allg. **weder physisch noch logisch konsistent.**

Ziel

- **Systemunterstützung**
  - zur **Wiederherstellung (Recovery)**
    eines logisch und physisch konsistenten Zustands der Daten
    nach einem Fehlerfall
    - Am besten automatisch
- **Der herzustellende konsistente Zustand kann sein:**
  - Entweder der vor Beginn der Änderungen eines unvollständig ausgeführten Programms herrschende
    - **Rückgängigmachen** der bereits ausgeführten Änderungen
      (**Backward Recovery**)
  - Oder der nach Abschluss aller Änderungen eines Programms erreichte
    - Komplettieren der unvollständigen Änderungen
      bzw. **Wiederholen** verlorengegangener Änderungen
      (**Forward Recovery**)

Ziel (2)

- **Voraussetzung:**
  - Geeignete **Sicherungs- und Protokollierungsmaßnahmen** im laufenden Betrieb (**Logging**)
    - Gezieltes Führen von (Sicherungs-) Kopien der Daten für die Wiederherstellung im Fehlerfall
    - Also Kosten: Speicher, E/A, CPU

8.1 Die Transaktion

- **Transaktion (TA):**
  - **Folge von DB-Operationen**,
    die, von einem logisch konsistenten Zustand ausgehend,
    die Datenbank wieder in einen logisch konsistenten Zustand überführt.
  - Muss vom Anwender definiert werden
- **Bei Fehler *vor* dem Ende der Transaktion:**
  - Alle Änderungen, die eine Transaktion bereits ausgeführt hatte,
    werden automatisch so **rückgängig gemacht**,
    dass sich die Daten wieder in dem Zustand befinden,
    in dem sie zu Beginn der Transaktion waren (Systemgarantie).
  - Der Datenbestand sieht dann aus, als ob die TA nie gestartet worden wäre.
  - Anwender kann selbst die TA neu starten (Programmlauf wiederholen).
- **Bei Fehler *nach* dem Ende der Transaktion:**
  - Alle Ergebnisse der Transaktion,
    die durch den Fehler verlorengegangen waren,
    werden automatisch **wiederhergestellt**.
  - Anwender bemerkt den Fehler nicht.

Die Transaktion (2)

- **Aufgabe des Anwenders dann nur noch: Transaktion definieren**
  - (Anfang und) Ende
    - Anfang meist implizit (Beginn des Programms, erste DB-Operation)
- **DB-Aufruf commit**
  - Anwendungsprogramm teilt dem DBVS mit,
    - dass es durch eine Reihe von DB-Änderungsoperationen wieder einen logisch konsistenten Zustand hergestellt hat oder
    - dass es nach einer Reihe von lesenden Zugriffen so weit fertig ist, dass andere die gelesenen Daten auch wieder ändern dürfen.
- **DB-Aufruf abort oder rollback**
  - Rückkehr in den Anfangszustand auf Wunsch des Anwenders

Beispiel

- **Programm zur Ausführung von Banküberweisungen:**

```
...
for ( ;; )
{
    scanf("%d %d %d", &von_konto, &auf_konto, &betrag);
    if ( ... )
          break;
    exec sql      update Kontos
                  set Stand = Stand - :betrag
                  where Kontonr = :von_konto;
    exec sql      update Kontos
                  set Stand = Stand + :betrag
                  where Kontonr = :auf_konto;
    exec sql commit;
}
```

  - Es darf weder Geld verschwinden noch welches aus dem Nichts entstehen!

Beispiel (2)

- **Andere Reihenfolge hilft auch nicht:**

Variante 1

```
update Kontos
set Stand = Stand - 10
where Kontonr = 14235623;
```

Systemausfall

```
update Kontos
set Stand = Stand + 10
where Kontonr = 31232124;
```

Geld wurde vernichtet !

Variante 2

```
update Kontos
set Stand = Stand + 10
where Kontonr = 31232124;
```

```
update Kontos
set Stand = Stand - 10
where Kontonr = 14235623;
```

Geld wurde erzeugt !

- Einzelne DB-Operationen können die von der Anwendung geforderten Verarbeitungseinheiten nicht realisieren.

Charakterisierung von Transaktionen

- **Charakterisierung**
  - Datenbanktransaktion stellt logische **Arbeitseinheit** dar.
    - Englisch deshalb auch: Logical Unit of Work
  - Zusammenfassung von aufeinander folgenden DB-Operationen, die eine Datenbank von einem konsistenten Zustand in einen neuen konsistenten Zustand überführen (physische und logische Konsistenz)
    - DB-Operationen hier gleichzusetzen mit SQL-Befehlen
  - Innerhalb einer Transaktion können vorübergehend logisch inkonsistente Zustände auftreten.

- **Der Transaktionsmanager des DBVS garantiert**
  - entweder vollständige Ausführung einer Transaktion
  - oder Wirkungslosigkeit der gesamten Transaktion (und damit aller in ihr enthaltenen Operationen).

Atomarität

Eine Transaktion ist atomar:

- **Unteilbar:**
  - Alle zu einer Transaktion gehörenden DB-Operationen bilden eine Einheit.
- **Ununterbrechbar:**
  - Aus der Sicht der Benutzer wird die Transaktion entweder vollständig ausgeführt oder gar nicht.
- **Zusicherung des DBVS an den Anwender:**
  Alles oder nichts

Konsistenzerhaltung

Transaktion

- setzt bei ihrem Start einen **konsistenten Zustand** des Systems voraus und
- hinterlässt nach ihrem erfolgreichen Abschluss wieder einen konsistenten Zustand

- **Zusicherung der Programmierer an das DBVS**
  - Kann nicht immer überprüft werden (vgl. Banküberweisung)
  - Was geprüft werden kann (z.B. Referenzielle Integrität), wird aber geprüft.

- **Transaktion muss vollständig sein**
  - D.h. alle erforderlichen Änderungen sind enthalten
  - **So lang wie nötig**

Isolation

- **Mehrbenutzerbetrieb:**
  - Auf demselben Datenbestand arbeiten gleichzeitig mehrere Programme (Transaktionen).
  - Die müssen sich so abstimmen (bzw. durch das DBVS so "synchronisiert" werden), dass keine Inkonsistenzen entstehen.
- **Ziele:**
  - Solange eine Transaktion noch läuft, darf keine andere Transaktion von ihr bereits durchgeführte Änderungen lesen und benutzen, weil
    - die noch unvollständig und damit inkonsistent sind
    - die Transaktion noch scheitern kann
      und diese Änderungen dann wieder verschwinden
      - Die andere Transaktion hätte sonst ein "Phantom" gesehen.
  - Transaktionen laufen **isoliert** voneinander ab.

Isolation (2)

- **Ziele (Forts.):**
  - Eine Transaktion T1 darf entweder
    nur den Zustand *vor* Ausführung einer anderen Transaktion T2 sehen
    oder
    nur den Zustand *nach* Ausführung von T2.
    - Sieht sie Datenobjekt 1 vor der Änderung durch T2
      und Datenobjekt 2 nach der Änderung,
      kann dies kein konsistenter Zustand sein,
      sonst wären die beiden Änderungen ja nicht in T2 zusammengefasst worden.

Isolation (3)

- **Synchronisation der Transaktionen**
  - muss so erfolgen, dass die Wirkung auf den Datenbestand dieselbe ist wie bei einer strikt seriellen Ausführung der Transaktionen (eine nach der anderen)
  - **Fiktiver Einbenutzerbetrieb** für jede Transaktion
    - Optimal für Programmierung: nichts zu tun für gegenseitigen Ausschluss

- **Auch für rein lesende Zugriffe:**
  - Sollen nur einen konsistenten Zustand sehen
  - und keine inkonsistenten Zwischenzustände gleichzeitig arbeitender Änderungstransaktionen

- **Realisiert durch Blockierung von Daten für andere**
  - Deshalb Transaktion **so kurz wie möglich** machen

Dauerhaftigkeit

- **Eine dem Benutzer als erfolgreich abgeschlossen gemeldete Transaktion ist dauerhaft ("persistent").**

- **Zusicherung des DBVS an die Benutzer:**
  - Ihre Änderungen im Datenbestand gehen nicht durch spätere Fehler (Verlust des Puffers, Gerätefehler) verloren.
  - Sie bleiben unverändert erhalten
    (bis eine nachfolgende Transaktion sie wieder überschreibt).

Zusammenfassung: "ACID"-Eigenschaften

- **Atomarität (atomicity)**
  - Unteilbarkeit durch Transaktionsdefinition (Begin/BoT – End/EoT)
  - Alles-oder-Nichts-Prinzip
- **Konsistenzerhaltung (consistency)**
  - Eine erfolgreiche Transaktion garantiert, dass alle Konsistenzbedingungen (Integritätsbedingungen) eingehalten werden.
- **Isolation (isolation)**
  - Mehrere Transaktionen laufen voneinander isoliert ab und benutzen keine (inkonsistenten) Zwischenergebnisse anderer Transaktionen.
- **Dauerhaftigkeit (durability)**
  - Alle Ergebnisse erfolgreicher Transaktionen müssen persistent gemacht worden sein, bevor der Erfolg an die Anwendung gemeldet werden darf.

8.2 Programmierung von Transaktionen

- **DB-Zugriffe nur noch innerhalb von Transaktionen erlaubt**
  - Eine Transaktion beginnt automatisch mit der ersten ausführbaren SQL-Anweisung im Programm.
  - Die Anweisungen

    ```
    exec sql commit;
    exec sql rollback;
    ```

    beenden die Transaktion.
  - Mit der nächsten ausführbaren SQL-Anweisung beginnt automatisch eine neue.
- **Wichtiges Prinzip:**
  - Transaktionen sollten so kurz wie möglich und so lang wie nötig sein! (siehe oben)
    - Möglichst keine Interaktion mit dem Benutzer innerhalb einer Transaktion!
    - Eingabedaten (soweit möglich) *vor* dem ersten Zugriff auf die Datenbank prüfen!

Programmierung von Transaktionen (2)

- **Wenn ein Programm *nur eine* Transaktion ausführt:**
  - Scheitert das Programm, kann es (ggf. nach Korrektur) mit den gleichen Eingabedaten noch einmal ausgeführt werden.
- **Wenn ein Programm *zwei oder mehr* Transaktionen hintereinander ausführt:**
  - Scheitert das Programm, muss seine erneute Ausführung so modifiziert werden, dass bereits erfolgreich abgeschlossene Transaktionen nicht wiederholt werden!
    - Von den Eingabedaten müssen die weggelassen (übergangen) werden, die bereits zu einer erfolgreichen Änderung des Datenbestands geführt haben (vgl. Banküberweisung).
    - Der Programmierer muss den Fall der erneuten Ausführung explizit vorsehen,
      - d.h. den Programmzustand (Inhalte wichtiger Variablen) sichern und beim Wiederholen in Abhängigkeit davon Programmteile überspringen.

Beispiel

Relation zur Verwaltung eines Lagerbestands:

Vorräte (Teilenr, Lager, Menge, StandVom)

Einlesen und Verbuchen des Tagesumsatzes
in mehreren aufeinander folgenden Transaktionen,
damit der Datenbestand nicht zu lange blockiert ist
und nach Fehler und Rücksetzen nicht ganz von vorn begonnen werden muss:

```
#include <stdio.h>
#define EOF ...

main () {
  exec sql begin declare section;
       char datum[11], stand_vom[11], lager[21];
       unsigned int teilenr, entnahme, nachlieferung;
  exec sql end declare section;

  char letztes_datum[11] = " ";
```

Beispiel (2)


```
while (! EOF) {
   scanf ("%10s %u %20s %u %u",
            datum, &teilenr, lager, &entnahme, &nachlieferung);

   if (nach (letztes_datum, datum)) {
      printf ("Eingabe nicht nach Datum sortiert!\n");
      exit (1);
   }

   strcpy (letztes_datum, datum);

   exec sql  select stand_vom
             into :stand_vom
             from Vorrat
             where Teilenr = :teilenr
             and Lager = :lager;
```

Beispiel (3)


```
    if (nach (datum, stand_vom)) {
       exec   sql  update Vorrat
                   set Menge = Menge
                        + :nachlieferung
                        - :entnahme,
                       stand_vom = :datum
                   where Teilenr = :teilenr
                   and Lager = :lager;
    }

    exec sql commit;

  } /* while */

  }
```

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)

Ausführen Methode

Anwendungsstrukturen

Adressierungseinheiten:	Anwendungsobjekte
Hilfsstrukturen:	Def. konz. u. ext. Schema
Adressierungseinheiten:	Relationen, Sichten, Tupel

Objektverarbeitung

mengenorientierte DB-Schnittstelle

Sprachen wie SQL, Transaktionen

Logische Datenstrukturen

Adressierungseinheiten:	Relationen, Sichten, Tupel
Hilfsstrukturen:	Schemabeschreibung, Integritätsregeln
Adressierungseinheiten:	Sätze, Indexe

Übersetzung, Pfadoptimierung

interne Satzschnittstelle

r := SPEICHER <Satz>;
LÖSCH r;

Speicherungsstrukturen

Adressierungseinheiten:	Sätze, Indexe
Hilfsstrukturen:	Positionsindex, Freisp., Hash-Buckets usw.
Adressierungseinheiten:	Seiten, Segmente

Satzverwaltung, Zugriffspfadverwaltung

Schichtenmodell (2)

interne Satzschnittstelle

r := READ <Satzadr>;
<Satzadr> := INSERT r;

Speicherungsstrukturen

Adressierungseinheiten:	Sätze, Indexe
Hilfsstrukturen:	Positionsindex, Freisp., Hash-Buckets usw.
Adressierungseinheiten:	Seiten, Segmente

Satzverwaltung, Zugriffspfadverwaltung

seitenorientierte Pufferschnittstelle

BEREITSTELLEN Segment, Seite;
FREIGEBEN Seite;

Seitenzuordnungsstrukturen

Adressierungseinheiten:	Seiten, Segmente
Hilfsstrukturen:	Seitentabelle, Blocktabellen
Adressierungseinheiten:	Blöcke, Dateien

Pufferverwaltung, Einbringstrategien

blockorientierte Dateischnittstelle

LIES Datei, Block, k;
SCHREIB k, Datei, Block;

Speicherzuordnungsstrukturen

Adressierungseinheiten:	Blöcke, Dateien
Hilfsstrukturen:	VTOC, Extent-Tabellen, Dateikatalog
Adressierungseinheiten:	Slots, Spuren, Zylinder, Geräte usw.

Externspeicherverwaltung

Geräteschnittstelle

Kanalkommandos

Phys. DB

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)

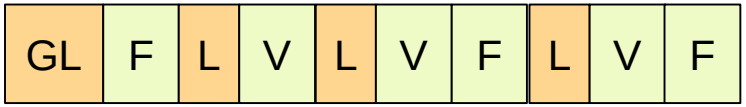
- **Satztyp**
  - Menge von Sätzen mit gleicher Struktur
    - Z.B. Tupel derselben Relation
    - Einmalige Beschreibung im Systemkatalog
  - Jedem Satz wird beim Abspeichern ein Satztyp zugeordnet.
  - Zuordnung zu Segmenten:
    - Typischerweise n:1, manchmal n:m
  - Länge der Sätze eines Satztyps
    - 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

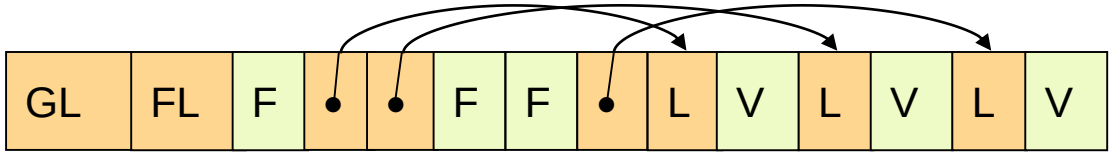
- **Konkatenation von Feldern fester Länge?**
  - Zu speicheraufwändig (immer Maximallänge)
- **Zeiger im Vorspann?**
  - Unflexibel (beim Hinzufügen von Feldern)
- **Eingebettete Längenfelder**
  - Dynamische Erweiterung möglich
  - Keine direkte Berechnung der satzinternen Adresse eines Felds aus Katalogdaten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

Vorlesung Implementierung von Datenbanksystemen

11. Relationale Operatoren

Prof. Dr. Klaus Meyer-Wegener

Wintersemester 2019/20

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

11.1 Transformation

- **Grundsätzliche Aufgabe:**
  - Ersetzen der Logischen Operatoren (SEL(), PROJ(), JOIN(), …) durch (ausführbare) **Planoperatoren**
- **Teilprobleme**
  - Gruppierung von direkt benachbarten Operatoren zur Auswertung durch einen einzigen Planoperator
    - Beispiel: Verbund mit Selektionen und/oder Projektionen auf den beteiligten Relationen durch einen speziellen Planoperator gemeinsam ausführen.
  - Bestimmung der Reihenfolge bei Verbundoperationen
    - Ziel: minimale Kosten für die Operationsfolge
    - Heuristik: Minimierung der Größe der Zwischenergebnisse, d.h. die kleinsten (Zwischen-)Relationen immer zuerst verknüpfen
  - Erkennen gemeinsamer Teilbäume
    - Einmalige Berechnung
    - Dafür nötig: Zwischenspeicherung der Ergebnisrelation

Relationale Planoperatoren

- **Ausführbar**
  - Unterprogramm / Komponente des DBMS
- **Parameter u.a.:**
  - Eingabe-Relationen
  - Zu verwendende Indexstrukturen
  - Bedingungen (Prädikate)
- **Voraussetzungen:**
  - Vorhandensein bestimmter Speicherungsstrukturen (vor allem: Indexe)
- **Ergebnis (Ausgabe):**
  - Ganze Relation oder
  - Nächstes Tupel (Pipelining) oder
  - Nächste *n* Tupel
- **Kosten der Verarbeitung**
  - Zeit, Speicher, CPU, …

Planoperatoren (2)

- **SQL erlaubt komplexe Anfragen über k Relationen.**

- **Ein-Variablen-Ausdrücke**
  - beschreiben Bedingungen für die Auswahl von Elementen aus einer Relation.
- **Zwei-Variablen-Ausdrücke**
  - beschreiben Bedingungen für die Kombination von Elementen aus zwei Relationen.
- **k-Variablen-Ausdrücke**
  - werden typischerweise in Ein- und Zwei-Variablen-Ausdrücke zerlegt und durch entsprechende Planoperatoren ausgewertet.

Planoperatoren (3)

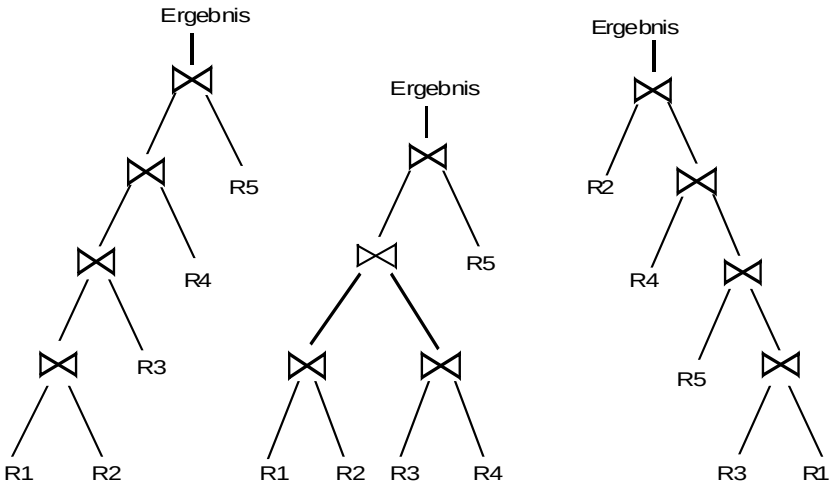
- **Selektion**
  - Mehrere Planoperatoren zur Auswahl
  - Nutzung eines **Scan**-Operators
    - Wahlweise mit Definition von Start- und Stopp-Bedingung und von einfachen Suchargumenten (z.B. Attribut = Wert)
  - Relationen-Scan
    - Sequenzielles Lesen aller Tupel einer Relation
  - Index-Scan
    - Direkt auf erstes passendes Tupel springen, dann ggf. sequenziell die weiteren
    - Nach Auswahl des kostengünstigsten Index
- **Projektion**
  - Typischerweise mit im Planoperator von Sortierung, Selektion oder Verbund durchgeführt
    - Denn die bearbeiten sowieso jedes Tupel.
  - Zusätzlich auch noch als eigener Planoperator

Planoperatoren (4)

- **Sortierung**
  - Erforderlich bei ORDER BY
    - Aber auch zur Beschleunigung von Joins
      - Siehe unten
    - Und für Gruppierung
    - Und zur Duplikateliminierung (DISTINCT, Mengenoperationen)
  - Deshalb eigener Planoperator
  - Leider "blockierend":
    - Muss das letzte Eingabetupel abwarten,
      bevor die erste Ausgabe erzeugt werden kann
    - Kein Pipelining möglich
  - I.Allg. extern, d.h. sortierte Teilergebnisse müssen auf Hintergrundspeicher ausgelagert werden (sog. Runs)
    - Am Schluss zum Gesamtergebnis zusammenmischen (Merge)

Planoperatoren (5)

- **Join über mehreren Relationen (n-Wege-Verbunde)**
  - Zerlegung in *n* – 1 Zwei-Wege-Verbunde
  - Anzahl der Verbundreihenfolgen abhängig von den Verbundattributen
    - *n*! verschiedene Reihenfolgen möglich
  - Optimale Auswertungsreihenfolge abhängig von
    - Planoperatoren,
    - "passenden" Sortierordnungen für Verbundattribute,
    - Größe der Operanden usw.
  - Verschiedene Verbundreihenfolgen mit Zwei-Wege-Verbunden (*n* = 5):



Planoperatoren für den Verbund

- **Eigenschaften der Verbundoperation:**
  - Teuer und häufig → Optimierungskandidat !!!
  - Typisch: Gleichverbund; allgemeines Verbundprädikat eher selten
  - Standardszenario:

```
SELECT * FROM R, S
WHERE R.VA Θ S.VA          // Verbundprädikat,
                           // Θ ∈ {=, >, <, ≠, ≤, ≥}
AND P(R.SA)                // lokale Selektionen
AND P(S.SA);               // VA = Verbundattribut,
                           // SA = Selektionsattribut
```

- **Mögliche Zugriffspfade:**
  - Relationen-Scan über R und S
  - Scans über Indexe I_R(VA) und I_S(VA), falls vorhanden
    - Dabei besonders interessant: Sortierreihenfolge nach R.VA und S.VA !!!
  - Scans über Indexe I_R(SA) und/oder I_S(SA), falls vorhanden
    - Schnelle Selektion für R.SA und S.SA !!!
  - Beliebige andere Kombinationen

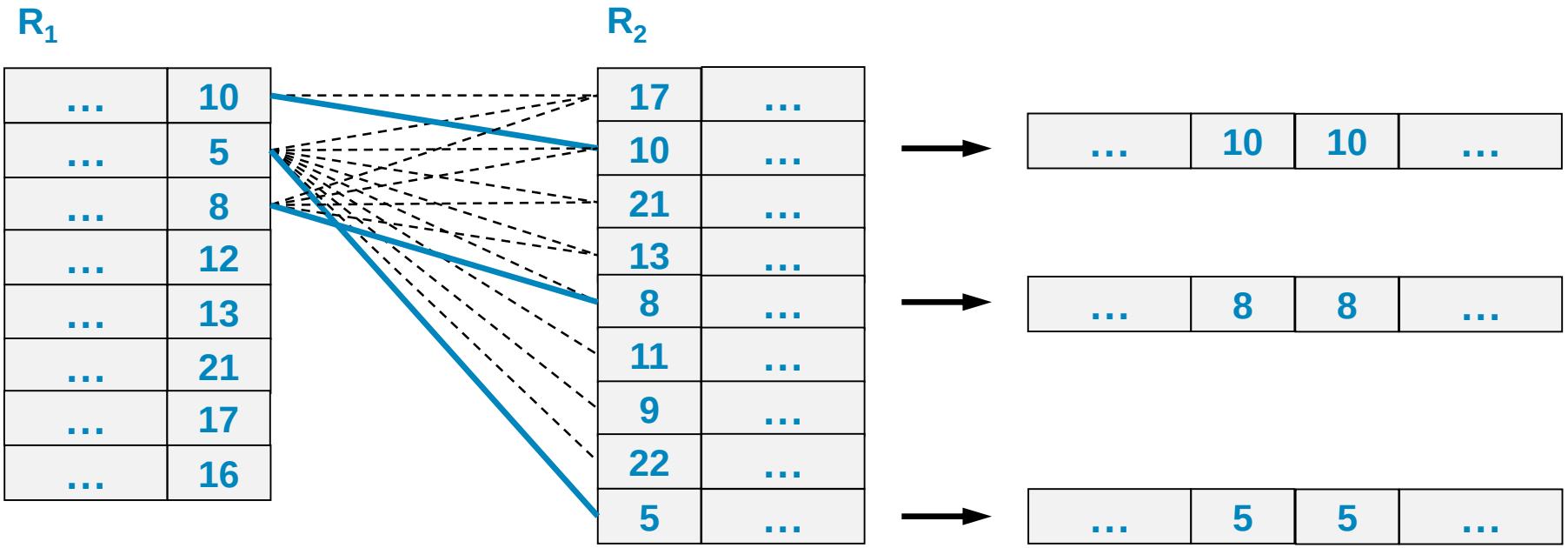
Nested-Loop-Verbund

- **Annahmen:**
  - Sätze in R und S sind nicht nach den Verbundattributen geordnet.
  - Es sind keine Indexstrukturen $I_R(VA)$ und $I_S(VA)$ vorhanden.
- **Algorithmus für Θ-Verbund:**

```
Scan über S;                    // äußere Schleife
für jeden Satz s, für den P(s.SA) gilt:
     Scan über R;               // innere Schleife
     für jeden Satz r,
     für den P(r.SA) AND (r.VA Θ s.VA) gilt:
            übernimm kombinierten Satz (r, s)
            in das Ergebnis;
```

- **Komplexität:**
  - $O(N^2)$

Beispiel zum Nested-Loop-Verbund



- **Merke:**
  - Falls das Verbundattribut der zweiten Relation die Eigenschaft UNIQUE aufweist, kann die Suche abgebrochen werden, nachdem der erste Verbundpartner gefunden wurde.

Nested-Loop-Verbund mit Indexzugriff

- **Annahme:**
  - Es sind Indexstrukturen I_R(VA) und I_S(VA) vorhanden.
- **Algorithmus für Gleichverbund mit Indexzugriff:**

```
Scan über S;
für jeden Satz s, für den P(s.SA) gilt:
   ermittle über I_R(R.VA) alle (TIDs der) Sätze mit r.VA = s.VA;
   ggf. für jedes TID: hol Satz r;
   für jeden Satz r, für den P(r.SA) gilt:
        übernimm kombinierten Satz (r, s) in das Ergebnis;
```

- **Merke:**
  - Eigentlich wird seitenweise vorgegangen.
    - Mehrfachen Zugriff auf dieselbe Seite vermeiden

Sort-Merge-Verbund

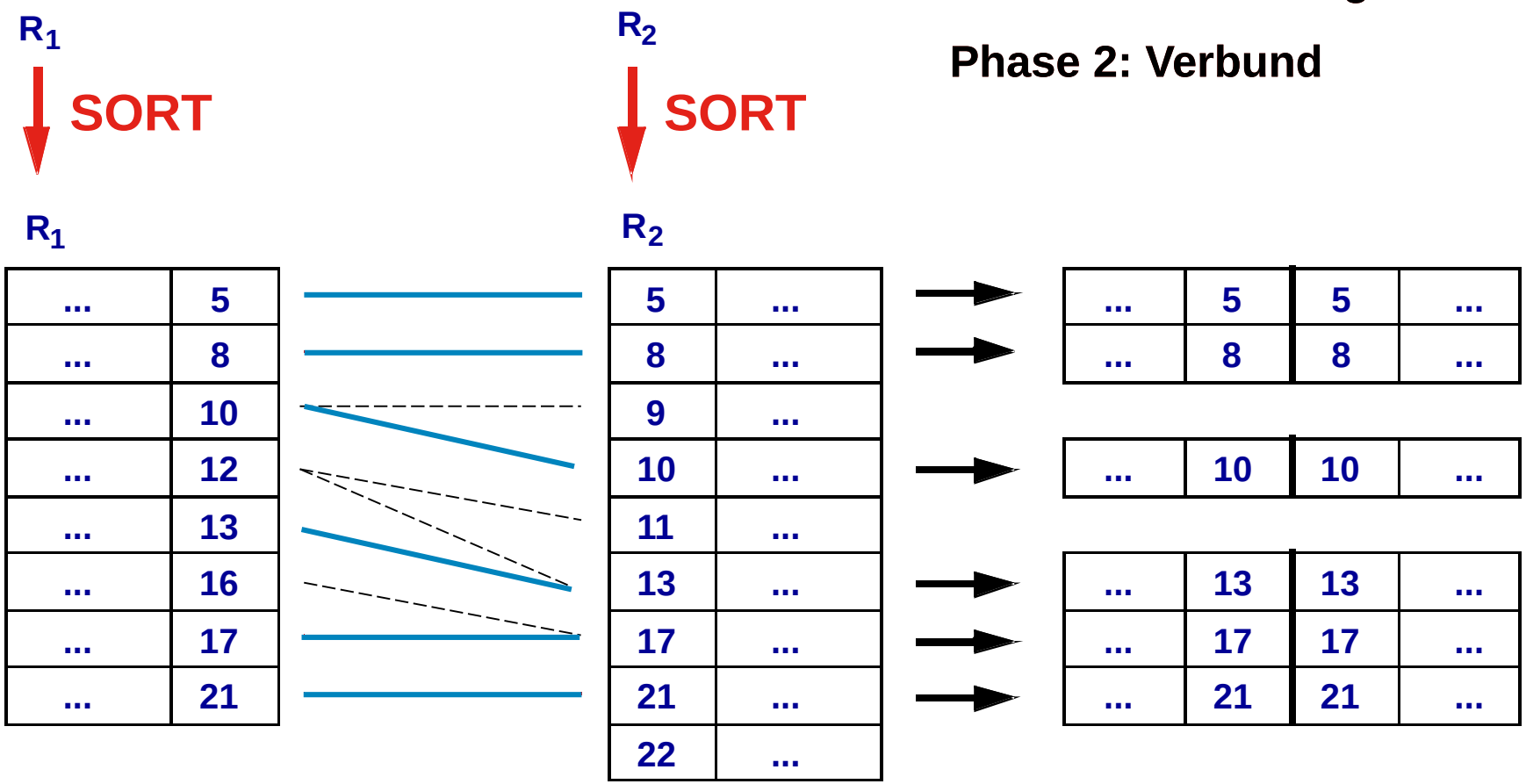
- **Zweiphasiger Algorithmus**
  - Phase 1:
    - Sortierung von R und S nach R.VA und S.VA (falls nicht bereits vorhanden), dabei frühzeitige Eliminierung nicht benötigter Tupel (durch Überprüfung von P(R.SA) bzw. P(S.SA))
  - Phase 2:
    - Schritthaltende Scans über sortierte Relationen R und S mit Durchführung des Verbunds bei r.VA = s.VA (d.h. auch Gleichverbund)
  - Komplexität: O(N log N)
- **Ausnutzen von Indexstrukturen I_R(VA) und I_S(VA) mit Sortierordnung:**

```
Schritthaltende Scans über I_R(R.VA) und I_S(S.VA);
für jeweils zwei Schlüssel aus I_R(R.VA) und I_S(S.VA),
für die r.VA = s.VA gilt:
   hol mit den zugehörigen TIDs die Tupel;
   für jedes Paar (r, s), für das P(r.SA) und P(s.SA) gelten:
       übernimm kombinierten Satz (r, s) in das Ergebnis;
```

Beispiel zum Sort-Merge-Verbund

Phase 1: Sortierung

Phase 2: Verbund



Hash-Verbund

- **Immer größere Hauptspeicher verfügbar**
  - Ausnutzen für Zwischenergebnisse
- **Gleichverbund**
  - Sehr häufiger Fall
- **Idee:**
  - Tupel der einen Relation so im Hauptspeicher ablegen,
    dass sie über Verbundattribut schnell gefunden werden können
  - Tupel der anderen Relation sequenziell durchlaufen
    und mit Wert des Verbundattributs
    die passenden Verbundpartner im Hauptspeicher aufsuchen
- **Organisation der Tupel im Hauptspeicher?**
  - Naheliegend: über Hashing

Hash-Verbund (2)

- **Einfachster Fall ("Classic Hashing")**
  - Äußere Schleife:
    - Abschnittweises Lesen der (kleineren) Relation **R**
    - Aufteilen in p Abschnitte R_i ($1 \leq i \leq p$) derart, dass
      - jeder der p Abschnitte in den verfügbaren Hauptspeicher passt;
      - jeder Satz, der gehasht wird, P(R.SA) erfüllt.
    - Aufbau einer Hash-Tabelle mit $\mathbf{h_A}$(r.VA) nach Werten von R(VA)
  - Innere Schleife (für jeden Abschnitt R_i):
    - Überprüfung ("Probing") für jeden Satz von **S**, der P(S.SA) erfüllt:
      - Ebenfalls Hashing h_A(s.VA)
        - Verbundpartner (falls vorhanden) muss sich an dieser Adresse befinden
    - Im Erfolgsfall Durchführung des Verbunds
- **Merke:**
  - Komplexität: $O(p \times N)$
  - Idealfall: R passt ganz in den Hauptspeicher, d.h. $p = 1$

Hash-Verbund (3)

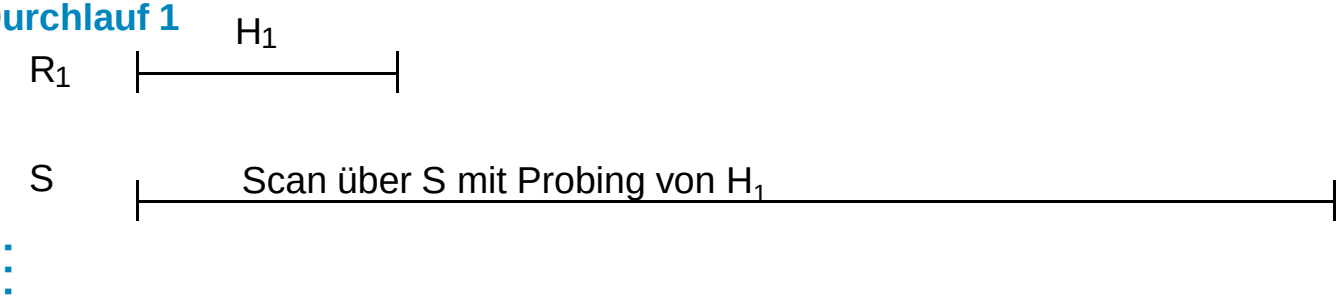
- **Pseudocode:**

```
Berechne Anzahl p der Abschnitte der kleineren Relation R,
so dass jeder Abschnitt in den Hauptspeicher passt;
Scan über R;
für jeden Abschnitt Ri, 1 <= i <= p:
    für jeden Satz r, der P(r.SA) erfüllt:
            Hash über r.VA, Ablage im Hauptspeicher;
    Scan über S;
    für jeden Satz s, der P(s.SA) erfüllt:
            Hash über s.VA, "Probing" im Hauptspeicher;
            bei Erfolg (Satz r mit r.VA = s.VA):
                    übernimm kombinierten Satz (r, s)
                    in das Ergebnis;
    Hauptspeicherinhalt (Hash-Tabelle) leeren;
```

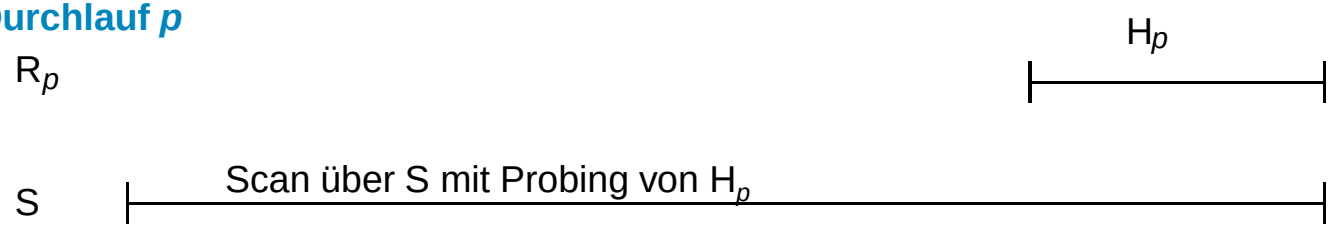
Aufbau der Hash-Tabelle

- **Aufbau der Hash-Tabelle und Probing**
  - Hash-Tabellen H_i ($1 \leq i \leq p$) werden schrittweise im Hauptspeicher aufgebaut.
  - Nach jedem Durchlauf von S wird die Hash-Tabelle wieder gelöscht.

Durchlauf 1



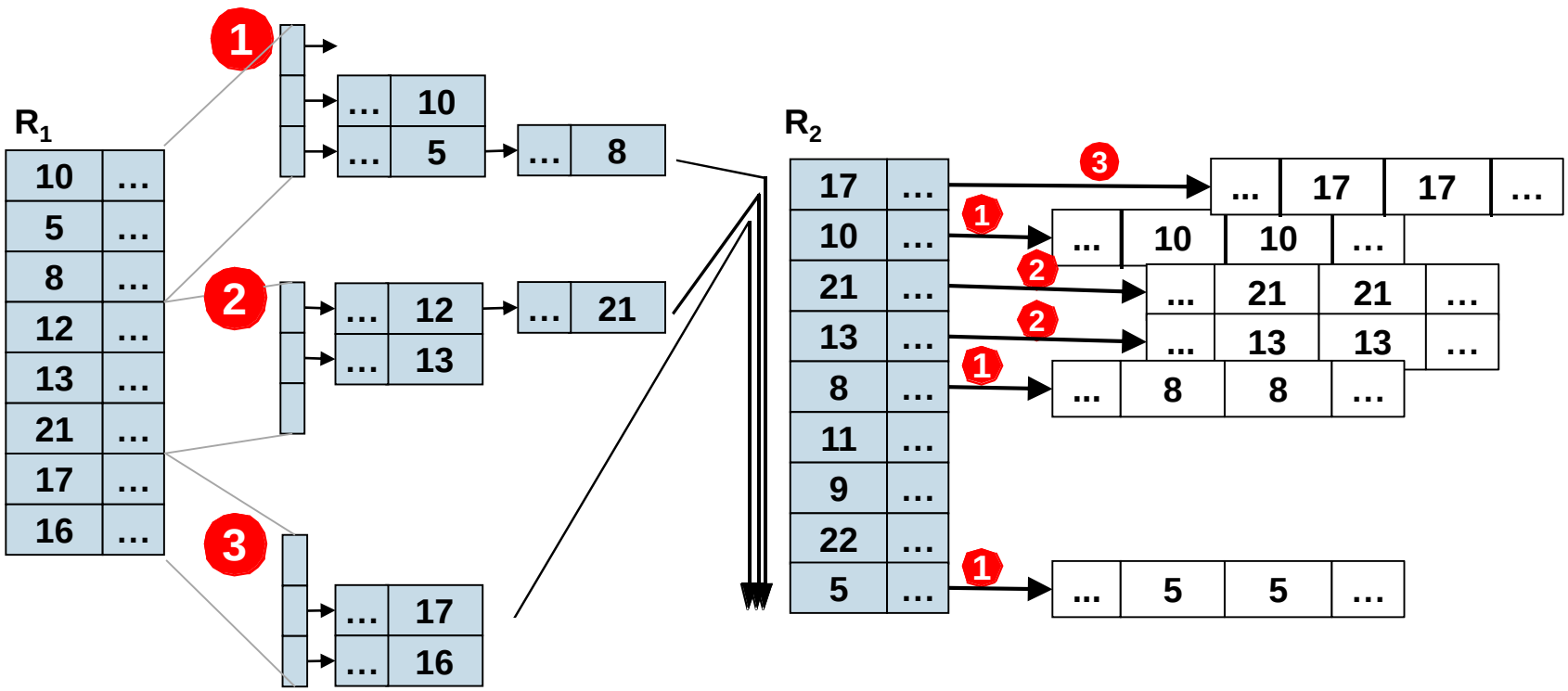
Durchlauf *p*



Beispiel zum Hash-Verbund

Annahme: Hauptspeicherkapazität = 3 Tupel

Hashing von R_1 mit h(x) = x mod 3



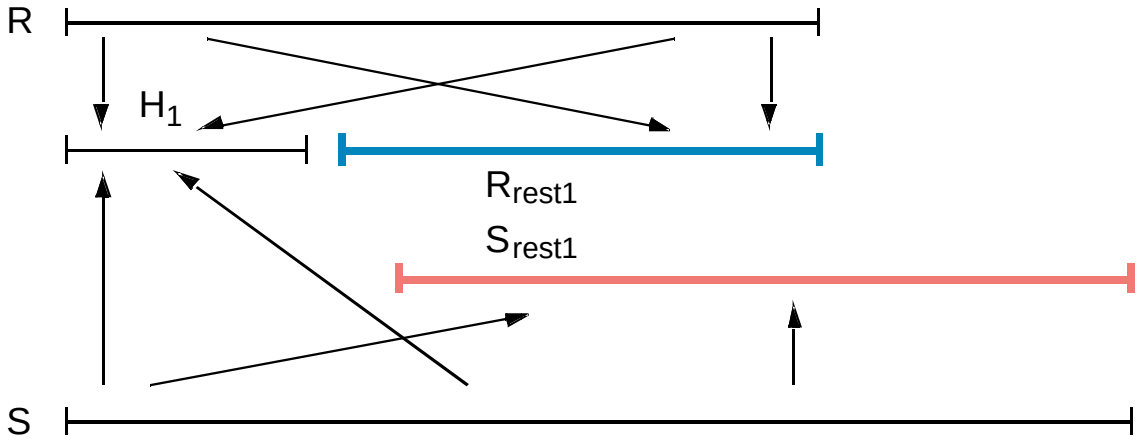
Verbesserungen am Verfahren

- **Nachteil des Hashing:**
  - Verbundpartner S muss *p*-mal gelesen werden.
- **Idee:**
  - Warum nicht auch S (analog zu R) partitionieren?
  - Hashing von R nicht nach der Reihenfolge der Tupel, sondern **wertemäßig** durchführen

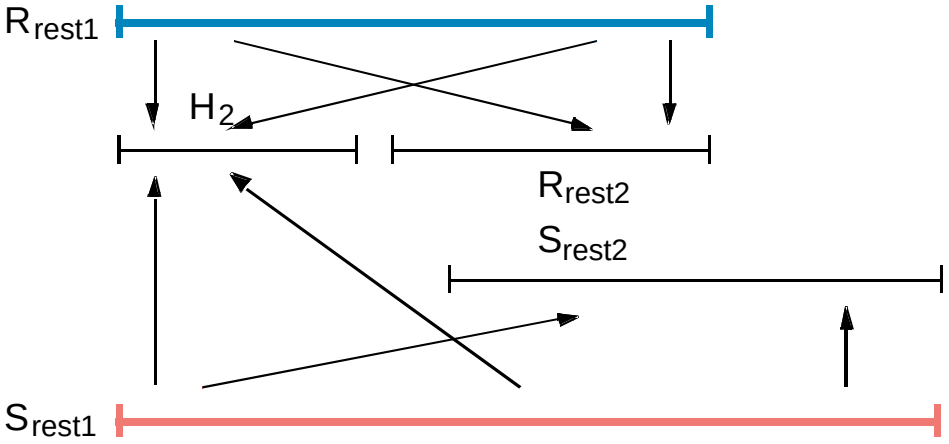
    $\rightarrow$ Partitionierung von R nach Werten von R.VA
  - Aber:
    - Nicht einfach, da üblicherweise keine Gleichverteilung der Werte vorliegt
    - Heranziehen von Statistiken (insbes. Histogramme !)
  - Zum Probing auch S nach den gleichen Kriterien partitionieren
- **Vielzahl unterschiedlicher Hash-Verfahren**
  - "Simple Hashing"

Partitionierungsprinzip beim Simple-Hash-Verbund

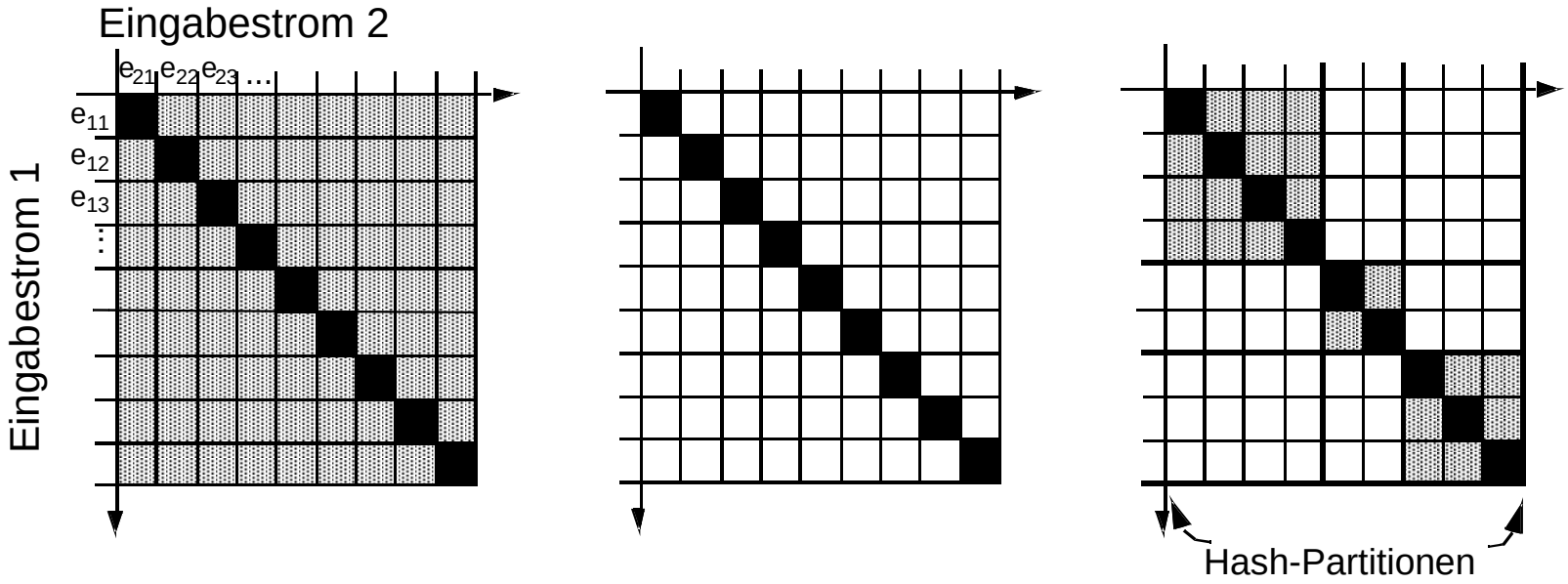
Durchlauf 1



Durchlauf 2



Vergleich der Verbundalgorithmen



Schleifeniteration **Mischverbund** **Hash-Verbund**

Elementvergleich

Elementvergleich, der zu einem Tupelverbund führt

Weitere Planoperatoren

- **Duplikat-Eliminierung:**
  - Klassisch durch Sortierung
    und "Gruppierung" im anschließenden Scan
  - Möglich aber auch Hashing:
    - Duplikate werden zwangsläufig auf gleiche Adresse abgebildet.
- **Gruppierung:**
  - Klassisch durch Sortierung
    und Scan mit Aggregation pro Gruppe
  - Aber ebenfalls über Hashing möglich:
    - Abbildung auf Zähler, bisherige Summe, bisheriges Minimum oder Maximum
    - Für Durchschnitt Kombination von Zähler und Summe

11.2 Anfrageoptimierung

- **Ziel:**
  - Von der Anfrage (WAS?) zur Auswertung (WIE?)
  - Ermittlung des *kostengünstigsten* Auswertungswegs
- **Zentrales Problem:**
  - Globale Optimierung ist im allgemeinen zu aufwändig.
    - Fehlen von exakten statistischen Informationen
    - Sehr großer Suchraum
- **Deshalb Einsatz von Heuristiken**
- **Optimierungsziel**
  - Entweder Maximierung des Outputs bei gegebenen Ressourcen
    - Durchsatzmaximierung (in Transaktionen pro Sekunde)
  - Oder Minimierung der Ressourcennutzung für gegebenen Output
    - Antwortzeitminimierung für eine gegebene Anfragesprache, einen Mix von Anfragen verschiedenen Typs und eine gegebene Systemumgebung

Kostenarten bei der Anfrageoptimierung

- **Berücksichtigung unterschiedlicher Kostenarten**
  - **Berechnungskosten**
    - CPU-Kosten
    - Pfadlängen (Anzahl Maschineninstruktionen)
  - **E/A-Kosten**
    - Anzahl der physischen Referenzen
  - **Speicherungskosten**
    - Temporäre Speicherbelegung im DB-Puffer und auf Externspeichern
  - Kommunikationskosten (bei verteilten DBS)
    - Anzahl der Nachrichten
    - Menge der zu übertragenden Daten
- **Merke:**
    - Kostenarten sind nicht unabhängig voneinander.
    - In DBVS oft gewichtete Funktion von Berechnungs- und E/A-Kosten

Erstellung und Auswahl von Ausführungsplänen

- **Eingabe:**
  - Algebraisch optimierter Anfragebaum
  - Existierende Speicherungsstrukturen und Zugriffspfade
  - Kostenmodell
- **Ausgabe:**
  - Optimaler (eher: guter) Ausführungsplan
- **Zwei oft notwendige Annahmen:**
  - Alle Datenelemente und alle Attributwerte sind gleichverteilt.
  - Suchprädikate in Anfragen sind unabhängig.
- **Beide Annahmen sind (im Allgemeinen) falsch!**
  - Beispiel:
    (Gehalt ≥ 100.000) AND (Alter BETWEEN 20 AND 30)
    mit Gehalt: [10K .. 1M] und Alter: [20 .. 65]
    - Lineare Interpolation und Multiplikation von Wahrscheinlichkeiten eher nicht angemessen …

Erstellung und Auswahl von Ausführungsplänen (2)

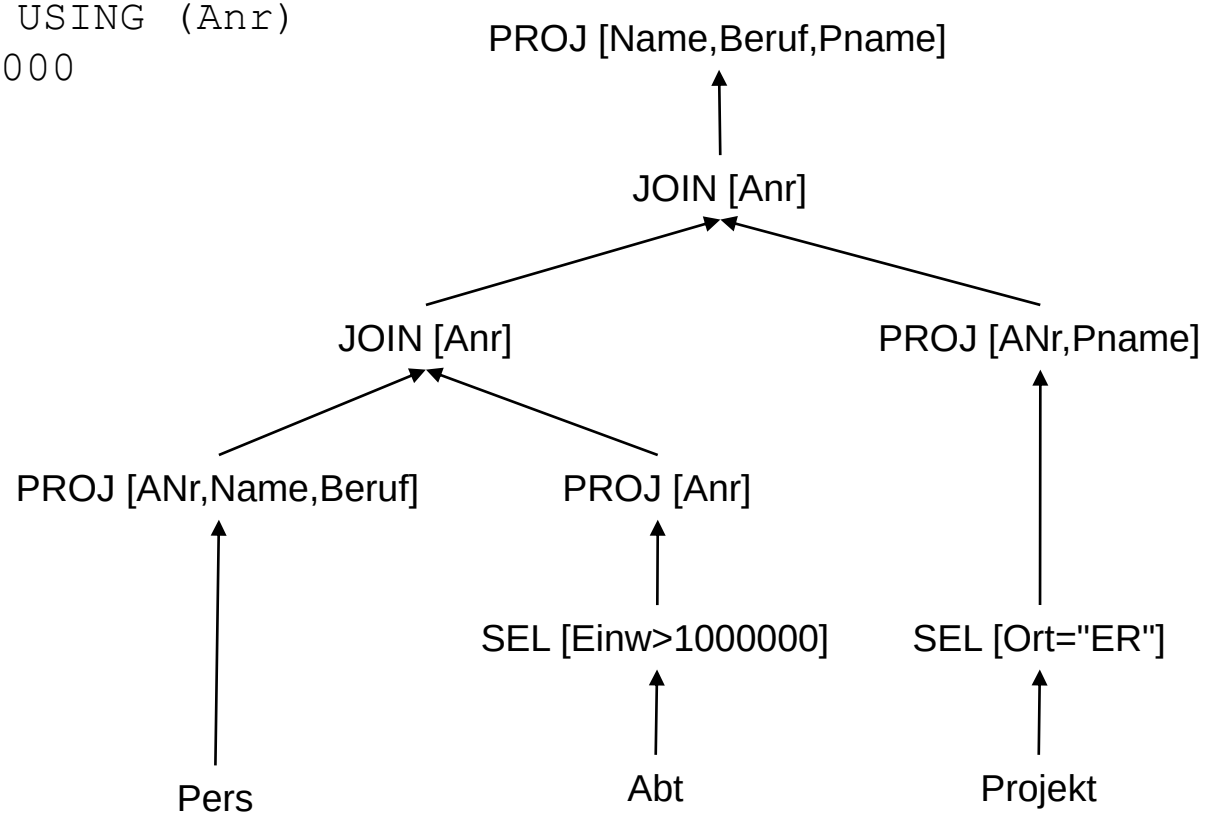
- **Vorgehensweise (vereinfacht):**
  - Generieren aller "vernünftigen" logischen Ausführungspläne zur Auswertung der vorliegenden Anfrage
  - Vervollständigen der Ausführungspläne mit Einzelheiten aus der physischen Datenrepräsentation (Sortierreihenfolge, Zugriffspfadmerkmale, statistische Information)
  - Bewertung der generierten Alternativen und Auswahl des billigsten Ausführungsplans gemäß dem vorgegebenen Kostenmodell
- **Merke:**
  - Wie entstehen alternative Ausführungspläne ???
    - Planoperatoren liegen in verschiedenen Implementierungen vor.
    - Operationsreihenfolgen (z.B. bei Mehrfachverbunden) können variiert werden.
  - Entstehung sehr großer Suchräume bei komplexen Anfragen mit allen Alternativen
    - Beispiel: ca. 10^{23} mögliche Ausführungspläne bei einer Anfrage mit 15 Verbunden
      - 15! (Reihenfolge) * 3^{15} (versch. Planoperatoren) * 2^{15} (Index- o. Table-Scan)

Suchstrategien für einen guten Ausführungsplan

- **Ziel der Plangenerierung**
  - Auffinden eines guten Plans
    - Gelingt immer und geht schnell
  - Mit einer möglichst kleinen Anzahl generierter Pläne auskommen
- **Unterschiedliche Strategieklassen:**
  - voll-enumerativ
  - beschränkt-enumerativ
  - zufallsgesteuert
    - genetische Algorithmen und die Strategien des "simulated annealing"

Beispiel zur Auswahl eines Ausführungsplans


```
SELECT P.Name, P.Beruf, J.Pname
FROM (Pers P JOIN Abt A USING (Anr))
     JOIN Projekt J USING (Anr)
WHERE A.Einw > 1000000
AND J.Ort = "ER";
```

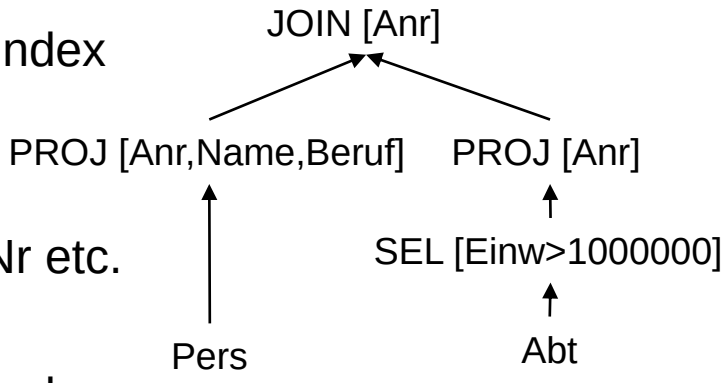


Beispiel (2)

- **Mögliche Zugriffspfade für die einzelnen Relationen**
  - Relation **Pers**:
    - Index I_P(ANr) – nicht eindeutig, für den Join mit Abt und Proj
    - Scan(P)
  - Relation **Abt**:
    - Index I_A(ANr) – eindeutig, für den Join mit Pers und Proj
    - Index I_A(Einw) – für die Selektion
    - Scan(A)
  - Relation **Proj**:
    - Index I_J(ANr) – nicht eindeutig, für den Join mit Pers und Abt
    - Index I_J(Ort) – für die Selektion
    - Scan(J)

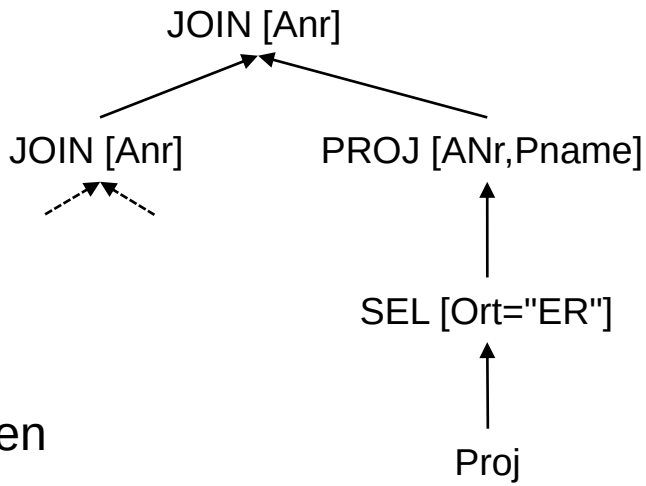
Beispiel (3)

- **Nested-Loop-Join von Pers und Abt**
  - Mit **Abt** anfangen
    - I_A(Einw) benutzen für Selektion
      - Bei jedem Treffer gleich Projektion auf ANr
    - I_P(ANr) benutzen für Join-Partner
      - Direktzugriff über ANr, kann jeder Index
      - Evtl. mehrere Join-Partner
  - Mit **Pers** anfangen
    - Scan(P) benutzen mit Projektion auf ANr etc.
    - I_A(ANr) benutzen für Join-Partner
      - Direktzugriff über ANr, kann jeder Index
      - Immer nur ein Join-Partner
      - Selektionsbedingung dann noch prüfen



Beispiel (4)

- **Zwischenergebnisse haben keinen Index.**
  - Nur Scan möglich
- **Nested-Loop-Join von (Pers, Abt) mit Proj**
  - Scan(P,A) benutzen
  - I_J(ANr) benutzen für Join-Partner
    - Direktzugriff über ANr, kann jeder Index
    - Mehrere Join-Partner
    - Selektionsbedingung prüfen
  - Alternativ I_J(Ort) benutzen – hohe Selektivität?
    - Dann das (kleine) Ergebnis
      für jedes Tupel von Scan(P,A)
      sequenziell nach Join-Partnern durchsuchen



Beispiel (5)

- **Weitere Möglichkeiten:**
  - Falls I_A(ANr) und I_P(ANr) Sortierung nach Schlüssel bieten: Sort-Merge-Join
  - Erst Join (A,J), dann Join des Ergebnisses mit P
    - Also andere Join-Reihenfolge
  - …

Berechnung der Zugriffskosten

- **Kostenvoranschlag**
  - für jeden Ausführungsplan (möglicher Lösungsweg)
- **Kostenformel**
  - Gewichtetes Maß für E/A- und CPU-Belastung:
    Kosten: #physische-Seitenzugriffe + W × #Aufrufe-des-Zugriffssystems
    - **W** ist das Verhältnis des Aufwands für einen Aufruf des Zugriffssystems (interne Satzschnittstelle) zum Aufwand für einen Seitenzugriff
  - Ziel der Gewichtung: Minimierung der Kosten abhängig vom Systemtyp
    - **CPU-bound**: höherer E/A-, geringerer CPU-Aufwand
      - Beispiel für W:
        W_{CPU} = #Instr-pro-Aufruf-des-Zugriffssystems / #Instr-pro-E/A-Vorgang
    - **I/O-bound**: geringerer E/A-, höherer CPU-Aufwand
      - Beispiel für W:
        W_{IO} = #Instr-pro-Aufruf-des-Zugriffssystems / (#Instr-pro-E/A-Vorgang + Zugriffszeit × MIPS-Rate)

Ermittlung statistischer Kenngrößen

- **Statistische Größen für Segmente**
  - M_S – Anzahl der Datenseiten des Segments S
  - L_S – Anzahl der leeren Seiten in S
- **Statistische Größen für Relationen**
  - N_R – Anzahl der Tupel in Relation R (auch card(R) oder | R |)
  - $T_{R,S}$ – Anzahl der Seiten in S mit Tupeln von R
  - C_R – Cluster-Faktor (Anzahl der Tupel von R pro Seite)
- **Statistische Größen pro Index I auf Attributen A einer Relation R**
  - j_I – Anzahl der Attributwerte (Schlüsselwerte) im Index (= card(PROJ(R,A))
  - B_I – Anzahl der Blattseiten (beim B*-Baum)
- **Merke:**
  - Statistiken müssen gesammelt und im DB-Katalog gewartet werden.

Ermittlung statistischer Kenngrößen (2)

- **Problem: Aktualisierung bei jeder Änderung sehr aufwändig**
  - Zusätzliche Schreib- und Log-Operationen
  - DB-Katalog wird zum Sperr-Engpass
- **Alternative:**
  - Initialisierung der statistischen Werte zum Lade- oder Generierungszeitpunkt von Relationen und Indexstrukturen
  - **Periodische Neubestimmung** der Statistiken durch eigenes Kommando oder Dienstprogramm
    - DB2: `runstats on table` ...
    - Oracle: `analyse table` ...

Selektivitätsabschätzungen

- **Ermittlung eines Selektivitätsfaktors SF (0 ≤ SF ≤ 1)**
  - Erwarteter Anteil an Tupeln, die ein Prädikat p erfüllen:
    SF(p) = card(SEL(R,p)) / card(R)
  - Beispiel-Heuristiken zur Abschätzung des Selektivitätsfaktors:

Prädikat	*SF*	*wenn*
$A_i = a_i$	$1 / j_{I_i}$ 1 / 10	Index I_i auf Attribut A_i sonst
$A_i = A_k$	$1 / MAX(j_{I_i}, j_{I_k})$ $1 / j_{I_i}$ $1 / j_{I_k}$ 1 / 10	Index auf beiden Attributen Index nur auf A_i Index nur auf A_k sonst
$A_i > a_i$	$(a_{max} – a_i) / (a_{max} – a_{min})$ 1 / 3	Index auf A_i und Werte interpolierbar sonst

(Großbuchstaben: Attributnamen; Kleinbuchstaben: Attributwerte)

Selektivitätsabschätzungen (2)

- Beispiel-Heuristiken zur Abschätzung des Selektivitätsfaktors (Forts.):

Prädikat	*SF*	*wenn*
A_i BETWEEN a_i AND a_j	$(a_j - a_i) / (a_{max} - a_{min})$ 1 / 4	Index auf A_i und Werte interpolierbar sonst
A_i IN $(a_1, a_2, \ldots, a_r)$	r / j_{I_i} 1 / 2	Index auf A_i und SF < 0,5 sonst

- **Berechnung von Verbundtermen im Qualifikationsterm**
  - SF(p(A) and p(B)) = SF(p(A)) · SF(p(B))
  - SF(p(A) or p(B)) = SF(p(A)) + SF(p(B)) – SF(p(A)) · SF(p(B))
  - SF(not p(A)) = 1 – SF(p(A))
- **Beachte:**
  - Wieder die Annahme der Unabhängigkeit von Prädikaten!

Beispiel zur Selektivitätsabschätzung

- **Anfrage**

```
SELECT Name, Gehalt FROM Pers
WHERE Beruf = "Programmierer" AND Gehalt >= 100000;
```

- **Vorhandene Zugriffspfade**
  - Relationen-Scan im Segment von Pers
  - I_{Pers}(Beruf)
  - I_{Pers}(Gehalt)
- **Vorhandene statistische Kennwerte**
  - **N** = Anzahl der Tupel in Relation Pers (N = 5000)
  - C_{Pers} = durchschn. Anzahl von Pers-Tupeln pro Seite
  - j_i = Index-Kardinalität (Anzahl der Attributwerte für A_i)
    - Es gibt 25 unterschiedliche Berufe.
    - Die Gehälter liegen zwischen 30.000 und 120.000.

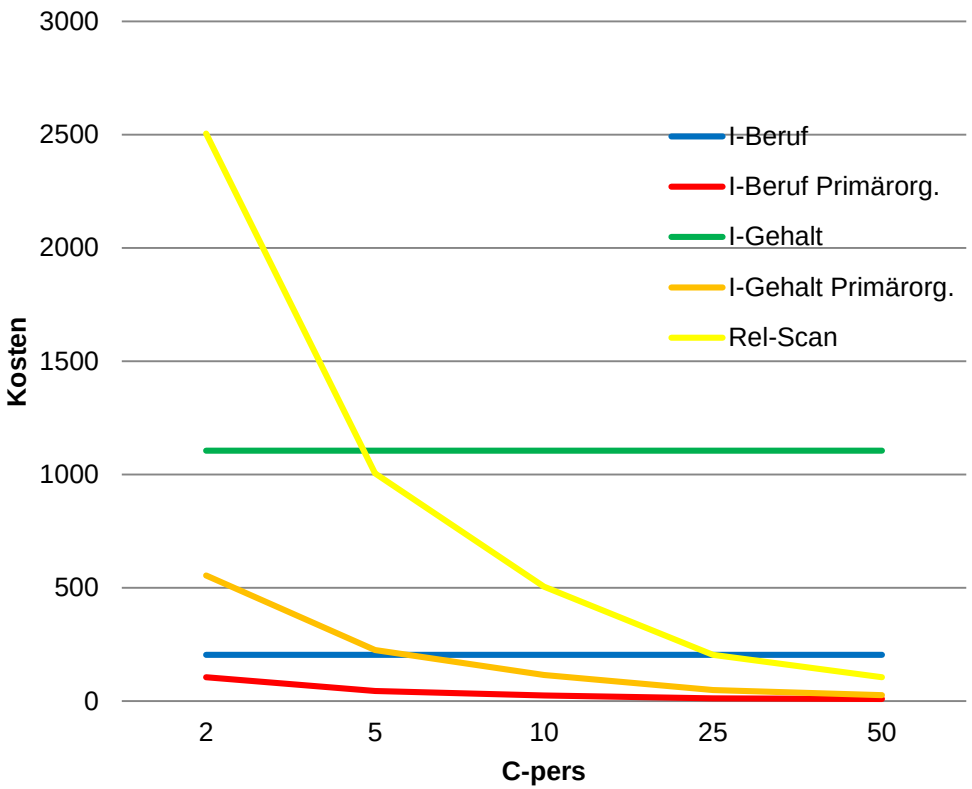
Beispiel zur Selektivitätsabschätzung (2)

- SF_B = SF(Beruf = "Programmierer") = 1 / j_{Beruf} = 1 / 25
- SF_G = SF(Gehalt >= 100.000) = (120.000 – 100.000) / (120.000 – 30.000) = 0,22

- **Methode 1: Scan über I_{Pers}(Beruf)**
  - Kosten: #physischer-Seitenzugriffe + W × #Aufrufe-des-Zugriffssystems
    $(N \times SF_B) + W \times (N \times (SF_B \times SF_G)) = N / 25 + W \times N \times 0{,}0088$
    - Index-Scan mit zusätzlichem einfachen Suchargument (Folie 11-5)
  - **Beachte**: Falls I_{Pers}(Beruf) Primärorganisation, reduziert sich die Anzahl der Seitenzugriffe um den Faktor C_{Pers} auf $(N / C_{Pers}) \times SF_B$
- **Methode 2: Scan über I_{Pers}(Gehalt)**
  - Kosten: $(N \times SF_G) + W \times (N \times (SF_B \times SF_G)) = N \times 0{,}22 + W \times N \times 0{,}0088$
  - **Beachte**: Falls I_{Pers}(Gehalt) Primärorganisation, reduziert sich die Anzahl der phys. Seitenzugriffe um den Faktor C_{Pers} auf $(N / C_{Pers}) \times SF_G$

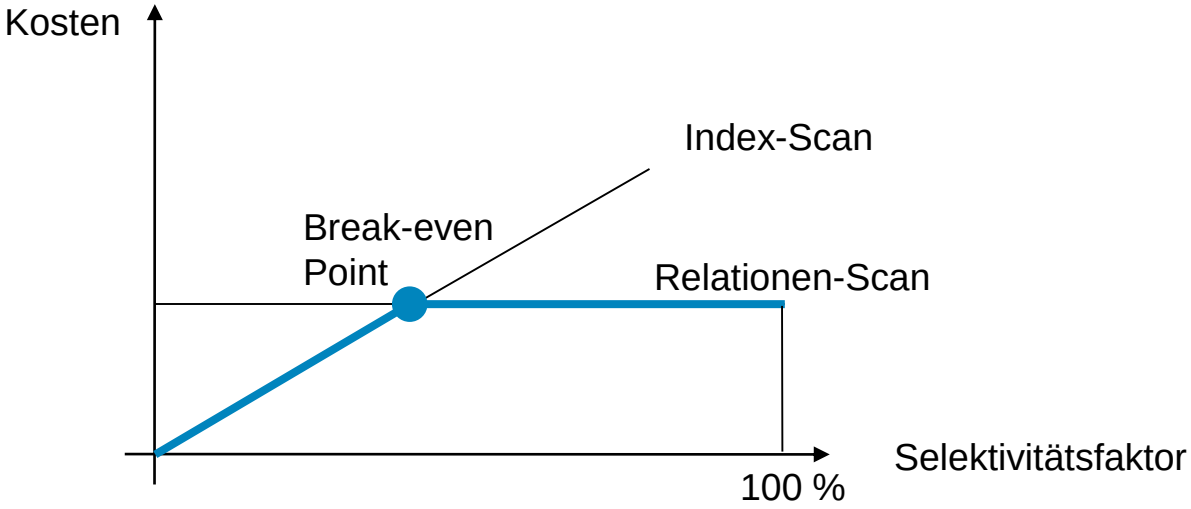
Beispiel zur Selektivitätsabschätzung (3)

- **Methode 3: Relationen-Scan**
  - Kosten: $(N / C_{Pers}) + W \times (N \times (SF_B \times SF_G)) = (N / C_{Pers}) + W \times N \times 0{,}0088$
- **Kostenvergleich:**
  - W = 0,1
    N = 5000



Betrachtung der Grenztrefferrate

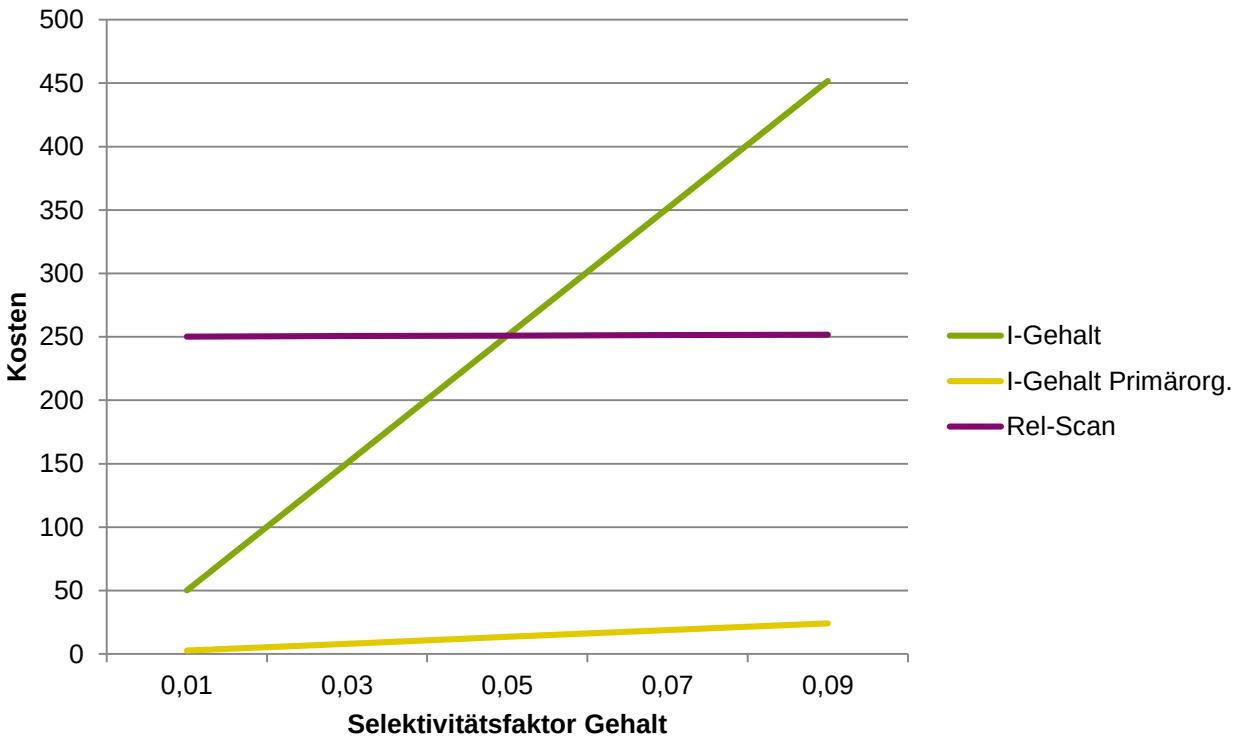
- **Folgerung:**
  - Vorhandene Indexstrukturen können die Laufzeit dramatisch reduzieren!
  - Aber nur bei sehr geringen Trefferraten (= hoher Selektivität) lohnt sich ein Index-Scan.
  - Qualitatives Zugriffsdiagramm:



  - Grenztrefferrate liegt üblicherweise bei max. 5 %.

Betrachtung der Grenztrefferrate (2)

- **Am Beispiel:**
  - C_{pers} nun fest bei 20
  - SF_G variiert



11.3 Zusammenfassung

- **Ziel der Anfrageoptimierung**
  - Ermittlung des kostengünstigsten Auswertungswegs
- **Standardisierung und Vereinfachung**
  - Umformungs-, Idempotenzregeln, …
- **Restrukturierung und Transformation**
  - Zuordnung von physischen Operatoren zu logischen
  - Reihenfolge der Operatorenausführung
- **Erstellung und Auswahl von Ausführungsplänen**
  - Sehr große Suchräume bei komplexen Anfragen
  - Strategieklassen: voll-enumerativ, beschränkt-enumerativ, zufallsgesteuert
  - Berechnung der Zugriffskosten:
    - Abschätzung von Selektivitäten
    - Nutzung vorhandener Zugriffspfade und statistischer Kennwerte

Vorlesung Implementierung von Datenbanksystemen

10. Anfrageverarbeitung

Prof. Dr. Klaus Meyer-Wegener
Demian Vöhringer

Wintersemester 2019/20

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

10.1 Einführung in die Anfrageverarbeitung

- **Realisierung eines mengenorientierten Zugriffs**
  - Nicht mehr Zugriff auf einzelne Sätze
  - Sondern inhaltliche Adressierung einer Menge von Sätzen
  - Reihenfolge der Satzzugriffe nicht vorgegeben
    - Anfrage **deskriptiv** (Was?), nicht prozedural (Wie?)

- **Abbildung von mengenorientierten Operatoren**
  - auf satzorientierte Operatoren
  - und die Benutzung von Indexstrukturen

- **SQL, Relationenalgebra**

Verarbeitung deskriptiver Anfragen

- **Verarbeitungsschritte:**
  - Überprüfung auf syntaktische Korrektheit (komplexe Syntax)
  - Überprüfung von Zugriffsberechtigung und Integritätsbedingungen
  - **Anfrageoptimierung**
    zur Erzeugung einer effizient ausführbaren Folge interner DBS-Operationen
  - Ausführung
- **Zentrale Aufgabe für RDBVS:**
  - Umsetzung deskriptiver Anfragen
    in eine "optimale" Folge
    interner DBS-Operationen (an der Satzschnittstelle)

Probleme bei deskriptiver Anfrageverarbeitung

- **Oder: Warum ist "Optimierung" so schwierig?**
- **Hohe Komplexität:**
  - Auswahlmächtigkeit an der Prädikatenlogik erster Stufe orientiert (inkl. Prädikate wie EXISTS, IS NULL, LIKE u.a.)
  - Unabhängige oder korrelierte Teilanfragen zur Bestimmung von Suchargumenten in beliebiger Schachtelungstiefe
  - Aggregations- und Sortier-Funktionen auf Partitionen der Satzmenge
- **Zusätzliche Anforderungen:**
  - Auch die Änderungsoperationen sind mengenorientiert.
  - Referenzielle Integrität ist aktiv mit Hilfe entsprechender Aktionen zu wahren.
  - Vielfältige Optionen der Datenkontrolle sind zu berücksichtigen.
- **... und was heißt überhaupt "optimal"?**
  - Maximaler Durchsatz, minimale Antwortzeit oder Einhalten von Antwortzeitschranken?

Beispiele deskriptiver SQL-Anfragen

- **Beispiel 1: Anfrage an eine einzelne Tabelle**

```
SELECT PNr, Name, Gehalt
FROM Pers
WHERE Beruf = 'Programmierer'
AND Provision > Gehalt;
```

- **Beispiel 2: Anfrage mit Korrelation**

```
SELECT P.PNr, P.Name, A.AName
FROM Pers P, Abt A
WHERE P.ANr = A.ANr
AND P.Gehalt < (SELECT MAX(Provision) FROM Pers)
AND P.Gehalt > (SELECT AVG(Provision) FROM Pers
                WHERE ANr = P.ANr);
```

  - Verbundoperation
  - Zwei Unteranfragen (**unabhängig** / **korreliert**)

Größeres Beispiel

Kunde { **KName, KAdr, Kto** }
Auftrag { **KName, Ware, Menge** }

```
SELECT Kunde.KName, Kto
FROM Kunde, Auftrag
WHERE Kunde.KName = Auftrag.KName
AND Ware = 'Kaffee';
```

Proj-Liste = `Kunde.KName, Kto`
Sel-Bed = `Kunde.KName = Auftrag.KName AND Ware = 'Kaffee'`

- Relation "Kunde": 100 Tupel; pro Seite 5 Tupel
- Relation "Auftrag": 10.000 Tupel; pro Seite 10 Tupel
- 50 Aufträge betreffen Kaffee.
- Ergebnis-Tupel der Form (KName, Kto): 50 von ihnen passen in eine Seite.
- 3 Ergebnis-Tupel von Kunde x Auftrag passen in eine Seite.
- Puffer bietet für jede Relation genau 1 Kachel.
- Keine Sätze über Seitengrenzen hinweg

Beispiel: direkte Auswertung

1. **R_1 := CROSS (Kunde, Auftrag)**

   Seitenzugriffe (*L* = lesend, *S* = schreibend):

   - *L* : (100 / 5 × 10.000 / 10) = 20.000
   - *S* : (100 × 10.000) / 3 ≈ 333.000

2. **R_2 := SELECT [Sel-Bed] (R_1)**

   - *L* : 333.000
   - *S* : 50 / 3 ≈ 17

3. **ERG := PROJECT [Proj-Liste] (R_2)**

   - *L* : 17
   - *S* : 1

- Insgesamt ca. **687.000 Seitenzugriffe**
  und ca. 333.000 Seiten zur Zwischenspeicherung

Beispiel: verbesserte Auswertung

1. **R_1 := SELECT [Ware = 'Kaffee'] (Auftrag)**
   - L : 10.000 / 10 = 1.000
   - S : 50 / 10 = 5
2. **R_2 := JOIN [KName = KName] (Kunde, R_1)**
   - L : 100 / 5 $\times$ 5 = 100
   - S : 50 / 3 = 17
3. **ERG := PROJECT [Proj-Liste] (R_2)**
   - L : 17
   - S : 1

- Ca. **1.140 Seitenzugriffe**
  - Um Faktor 500 verbessert

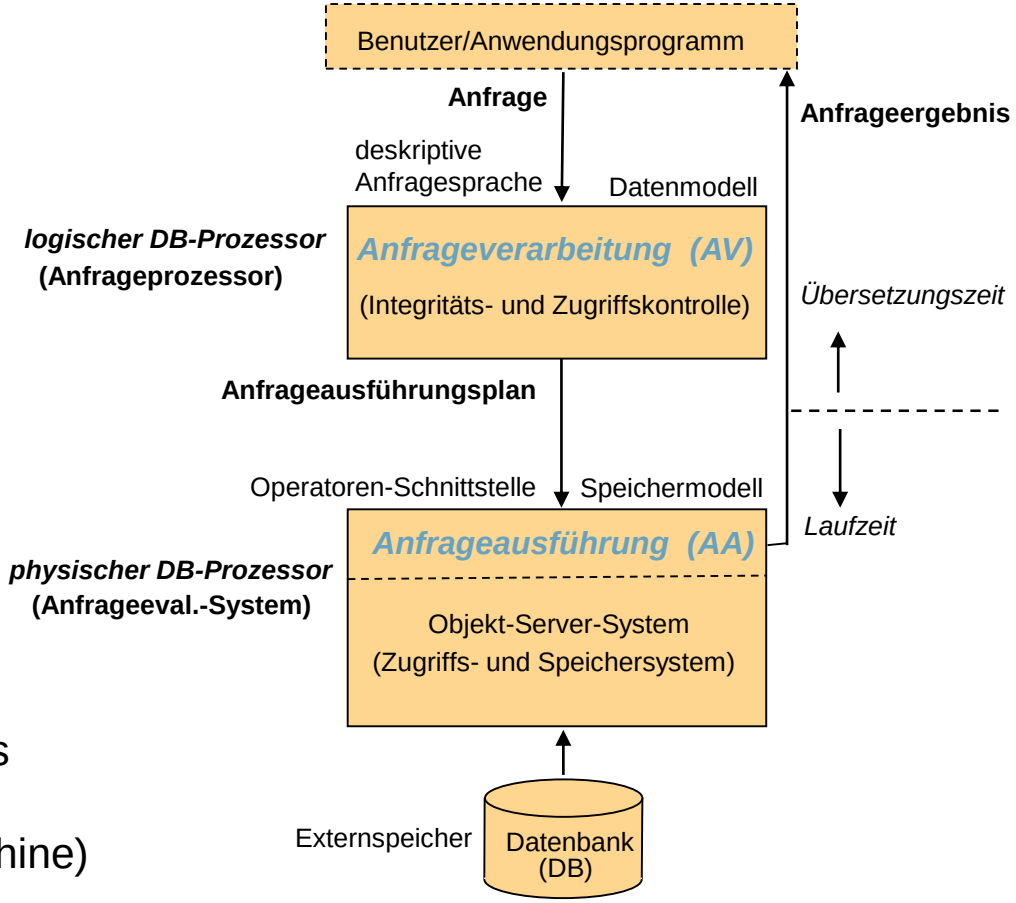
Beispiel: Indexausnutzung

Indexe `I1(Auftrag(Ware))` und `I2(Kunde(KName))`

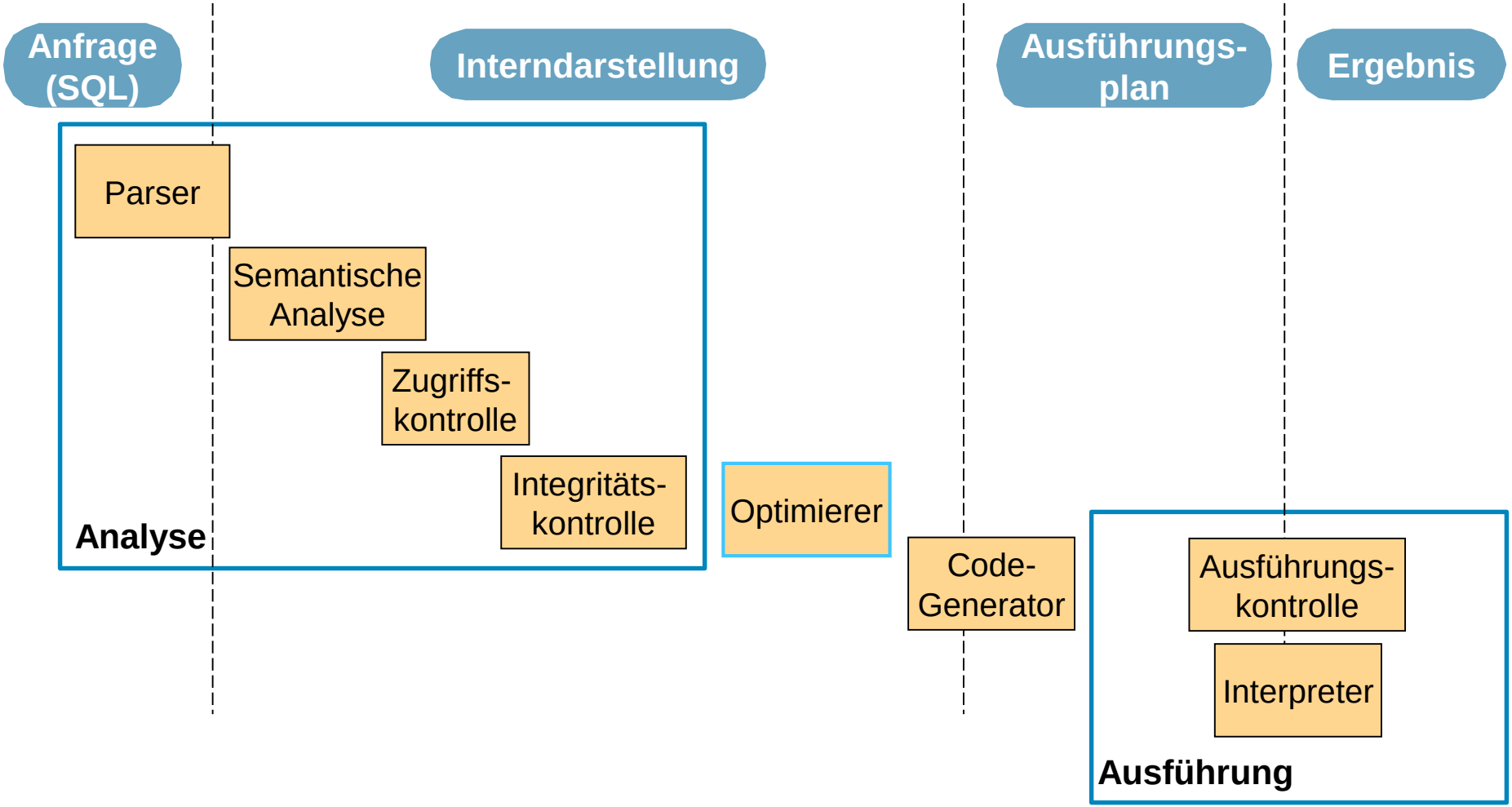
1. **R_1 := SELECT [Ware = 'Kaffee'] (Auftrag)**
   - Über `I1(Auftrag(Ware))`
     - L : minimal 5, maximal 50 je nach Index-Art
     - S : 50 / 10 = 5
2. **R_2 := sortiere R_1 nach KName**
     - $L + S$: $5 \times \log 5 \approx 15$
3. **R_3 := JOIN [KName = KName] (R_2, Kunde)**
     - L : 5 + 100 / 5 = 25
     - S : 50 / 3 = 17
4. **ERG := PROJECT [Proj-Liste] (R_3)**
     - L : 17
     - S : 1
   - Maximal ca. 130 und minimal ca. **85 Seitenzugriffe**

10.2 Abstrakte Sicht auf die Anfrageverarbeitung

- **Aufteilung der Anfrageverarbeitung (Query Processing)**
  - **Anfrageverarbeitung (AV)** (im engeren Sinne)
    - **Logischer DB-Prozessor**
    - Liefert einen **Anfrageausführungsplan** (query execution plan; QEP) zur **Übersetzungszeit**
  - **Anfrageausführung (AA)**
    - **Physischer DB-Prozessor**
    - Tatsächliche Ausführung des Anfrageausführungsplans zur **Laufzeit** (Interpretation, virtuelle Maschine)



Phasen der Anfrageverarbeitung – Überblick



Phasen der Anfrageverarbeitung

- **Lexikalische und syntaktische Analyse**
  - Überprüfung auf korrekte Syntax (Parsing)
  - Erstellen eines **Anfragebaums** für die nachfolgenden Übersetzungsschritte (d.h. Überführung in interne Darstellung)
- **Semantische Analyse**
  - Feststellen der Existenz und Gültigkeit der referenzierten Relationen und Attribute (Schema)
  - Ersetzen der externen durch interne Namen (Namensauflösung, Binden)
  - Konvertierung der Werte vom externen Format in interne Darstellung
- **Zugriffs- und Integritätskontrolle**
  - Durchführung einfacher Integritätskontrollen
    - Kontrolle von Formaten und ggf. Konvertierung von Datentypen
  - Generierung von Laufzeitaktionen für werteabhängige Kontrollen

Phasen der Anfrageverarbeitung (2)

- **Standardisierung und Vereinfachung**
  - Überführung des Anfragebaums in eine Normalform
  - Elimination von Redundanzen
- **Restrukturierung und Transformation**
  - **Algebraische** Verbesserung (**Restrukturierung**)
    - Anwendung von heuristischen Regeln
    - Zielt auf globale Verbesserung des Anfragebaums ab
  - **Nicht-algebraische** Verbesserung (**Transformation**)
    - Berücksichtigung ausführbarer Operationen (mit Kosten)
    - Ersetzen und ggf. Zusammenfassen der logischen Operatoren durch **Planoperatoren** (ausführbar)
  - Auswahl der günstigsten Planalternative
    - Meist sind mehrere Planoperatoren als Implementierung eines logischen Operators verfügbar.
    - Meist sind viele Ausführungsreihenfolgen und Zugriffspfade auswählbar.
    - Bewertung der Kosten und Auswahl des günstigsten Ausführungsplans

Phasen der Anfrageverarbeitung (3)

- **Code-Generierung**
  - Generierung eines zugeschnittenen Programms für die vorgegebene (SQL-) Anfrage
    - Zwischencode (früher sogar mal Assembler, heute auch: LLVM)
    - Enthält Aufrufe der Planoperatoren
  - Erzeugung eines ausführbaren **Zugriffsmoduls** (siehe Kap. 7)
  - Verwaltung der Zugriffsmodule in einer DBVS-Bibliothek

10.3 Interndarstellung einer Anfrage

- **Operationelle Betrachtung**
  - Problem: Wie wird eine SQL-Anfrage intern repräsentiert?
- **Relationale Algebra**
  - definiert relationale **logische Operatoren**,
    die für die interne Darstellung einer Anfrage
    in Form eines **Anfragebaums** (**Operatorbaums**) geeignet sind
    - Selektion: Auswahl von "Zeilen"
    - Projektion: Auswahl von "Spalten"
    - Kreuzprodukt: Konkatenation jedes Tupels der einen Relation mit jedem der anderen
    - Verbund: Konkatenation derjenigen Tupel aus zwei Relationen, die eine Bedingung erfüllen
    - Mengenoperatoren
  - erlaubt, Reihenfolge auszudrücken (prozedural)

Die Relationale Algebra

- **Mengenorientierte Operatoren auf einer Relationen**
  - **Selektion**: **SEL (R, pred(…))**
    - Auswahl einer mit **pred**(…) spezifizierten Teilmenge der Tupel von Relation R
    - Beispiel: **SEL (Personen, (Geburtsjahr > 1930))**
    - Wird in der WHERE-Klausel einer SQL-SELECT-Anweisung definiert:

```
SELECT ... FROM Personen
WHERE Geburtsjahr > 1930;
```

  - **Projektion**: **PROJ (R, L)** mit L = (A_1, ..., A_k)
    - Auswahl aller Tupel bezüglich einer Teilmenge L von Attributen der Relation R
    - Eliminierung von Duplikaten
    - Beispiel: **PROJ (Personen, (Vorname, Nachname))**
    - Wird in der SELECT-Klausel einer SQL-SELECT-Anweisung definiert:

```
SELECT Vorname, Nachname
FROM Personen ... ;
```

Die Relationale Algebra (2)

- **Mengenorientierte Operatoren auf zwei Relationen**
  - **Kreuzprodukt** zweier Relationen $R(A_1,...,A_n)$ und $S(B_1,...,B_m)$: **CROSS (R, S)**
    - Konkatenation jedes Tupels der Relation R mit jedem Tupel der Relation S
    - Beispiel: **CROSS (Personen, Filme)**
    - Wird in der FROM-Klausel definiert:

```
SELECT ... FROM Personen, Filme;
```

  - **Verbund** zweier Relationen $R(A_1,...,A_n)$ und $S(B_1,...,B_m)$:

    JOIN (R, S, pred) mit pred = $P(A_i, B_j)$
    - Verbinden zweier Relationen R und S gemäß eines Prädikats $P(A_i, B_j)$ über Attributen aus beiden Relationen
      - Eigentlich schon eine Optimierung, da auch durch Kreuzprodukt und Selektion darstellbar
    - Beispiel: **JOIN (Personen, Filme, (Nachname = Hauptdarstellername))**
    - Wird in der FROM-Klausel definiert:

```
SELECT ...
FROM Personen JOIN Filme
        ON Nachname = Hauptdarstellername;
```

Die Relationale Algebra (3)

- **Mengenorientierte Operatoren auf zwei Relationen (Forts.)**
  - **Mengenoperatoren:**

R ∪ S	**bzw. UNION (R, S)**
R ∩ S	**bzw. INTERSECT (R, S)**
R \ S	**bzw. EXCEPT (R, S)**
...	

- **Merke: auf logischer Ebene auch n-stellige Operatoren !**
  - Zurückführen auf eine Sequenz von binären

Die Relationale Algebra (4)

- **Multimengen-orientierte Operatoren**
  - Multimengen (engl. bags) haben Performance-Vorteile:
    - Bei Vereinigung einfach beide Operanden ausgeben
    - Bei Projektion nach Bearbeitung aller einzelnen Tupel fertig
  - Außerdem bei manchen Aggregationen (AVG, COUNT) explizit gewünscht
  - Also dafür **eigene (logische) Operatoren** – mit anderer Wirkung
    - Elemente (Tupel) kommen in den Multimengen mit einer best. Häufigkeit *n* vor; dabei ist *n* > 1 erlaubt.
    - UNION: Häufigkeiten addieren
    - INTERSECT: Minimum der Häufigkeiten
    - EXCEPT: max(0, n – m)
    - PROJECT: wie oben, nur ohne Duplikat-Eliminierung
    - SELECT, CROSS, JOIN: unverändert

Die Relationale Algebra (5)

- **Weitere Operatoren:**
  - Durch Anfragesprachen wie SQL hinzugekommen
  - **Umbenennung:**
    - **RENAME (R, [Rnew,] ((A1, A1new), (A2, A2new), …))**
  - **Duplikat-Eliminierung**:
    - **DUP-ELIM (R)**
  - **Aggregation**:
    - **SUM (R, attr), AVG (R, attr), MIN (R, attr), MAX (R, attr)**
      - Für numerische Attribute, bei MIN und MAX auch Zeichenketten
    - **COUNT (R)**
      - Anzahl der Tupel
  - **Gruppierung**:
    - **GROUP (R, L, agg)** mit Gruppierungsattributen L = (A_1, ..., A_k) und Aggregationen agg = ((AGG_1 (attr), $name_1$), (AGG_2 (attr), $name_2$), …)

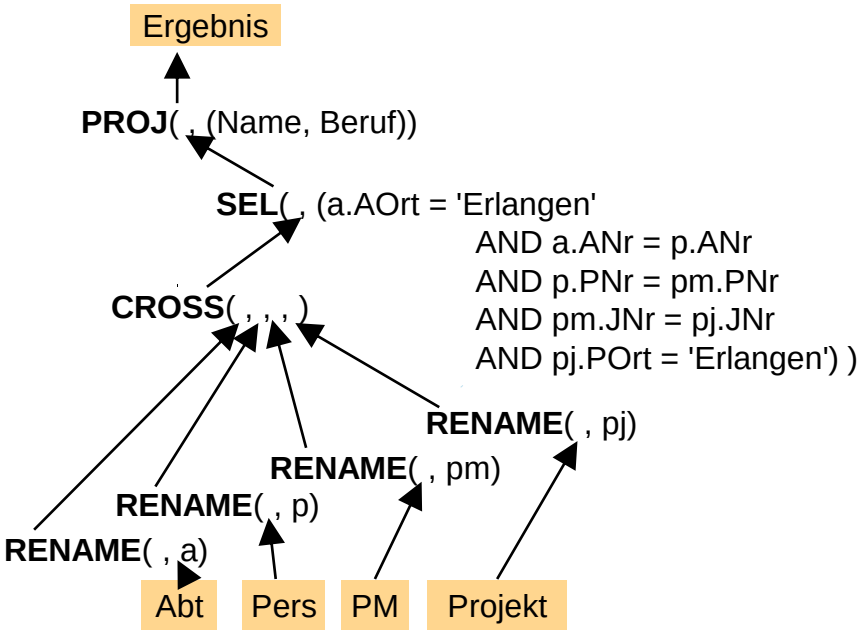
Die Relationale Algebra (6)

- **Weitere Operatoren (Forts.):**
  - **Erweiterte Projektion:**
    - **G-PROJ (R, L)** mit L = ($name_1$ = $expr_1$, $name_2$ = $expr_2$, …) Liste von Ausdrücken zur Berechnung von neuen Attributwerten
  - **Sortierung**:
    - **SORT (R, L)** mit L = (A_1, ..., A_k) Liste der Attribute, nach denen sortiert wird
    - Ergebnis ist Liste! Also nur ganz an Ende sinnvoll.
  - **Äußerer Verbund**:
    - **OUTER-JOIN (R, S, pred, case)** mit pred = P(A_i, B_j) wie beim normalen Join und case $\in$ {left, right, full}

Operatorbaum

- **Effiziente Datenstruktur mit geeigneten Zugriffsfunktionen**
  - Prozedurale Darstellung einer deskriptiven, mengenorientierten Anfrage
  - **Knoten** sind Operatoren der Relationalen Algebra.
  - **Blattknoten** sind (üblicherweise) Relationen.
  - Gerichtete **Kanten** repräsentieren den Datenfluss.
- **Beispiel**

```
SELECT Name, Beruf
FROM Abt a, Pers p,
     PM pm, Projekt pj
WHERE a.ANr = p.ANr
AND a.AOrt = 'Erlangen'
AND p.PNr = pm.PNr
AND pm.JNr = pj.JNr
AND pj.POrt = 'Erlangen';
```



10.4 Standardisierung und Vereinfachung

- **Standardisierung (der Qualifikationsbedingungen)**
  - Wahl einer Normalform
    - Konjunktive Normalform **(P_{11} OR ... OR P_{1n}) AND ... AND (P_{m1} OR ... OR P_{mp})**
    - Disjunktive Normalform **(P_{11} AND ... AND P_{1q}) OR ... OR (P_{r1} AND ... AND P_{rs})**
    - Pränex-Normalform (Verschiebung der Quantoren)
      z.B. zur Auflösung geschachtelter SELECT-Anweisungen
- **Vereinfachung**
  - Äquivalente Ausdrücke können einen unterschiedlichen Grad an Redundanz besitzen.
    - Idempotenzregeln
    - Ausdrücke mit "leeren Relationen"
  - Behandlung / Eliminierung gemeinsamer Teilausdrücke

$$(A_1 = a_{11} \text{ OR } A_1 = a_{12})$$
$$\text{AND}$$
$$(A_1 = a_{12} \text{ OR } A_1 = a_{11})$$

Standardisierung und Vereinfachung (2)

- **Vereinfachung (Forts.)**
  - Konstanten-Propagierung (allg. Hüllenbildung der Qualifikationsprädikate)

```
A <op> B AND B = const.
⇒    A <op> const.
```

  - Nicht erfüllbare Ausdrücke

```
A ≥ B AND B > C AND C ≥ A
⇒    A > A    ⇒ false
```

  - Nutzung von Information über semantische Integritätsbedingungen
    - A ist Primärschlüssel: **project**[A] → keine Duplikateliminierung erforderlich
    - Auswertung hinterlegter Regeln:

```
IF Fam-Stand = "verh." THEN Steuerklasse ≥ 3;
Fam-Stand = "verh." AND Steuerklasse = 1    ⇒ false
```

  - Umformungs- und Idempotenzregeln für Boole'sche Ausdrücke
  - Umformungsregeln für quantifizierte Ausdrücke

10.5 Restrukturierung

- **Äquivalente Umformung des Operatorbaums**
  - Man kann sich die effizientere Variante aussuchen!
- **Regeln:**

(1) **Ein n-facher Verbund kann durch eine Folge von binären Verbunden ersetzt werden und umgekehrt:**

```
JOIN (R1, R2, ..., Rn,)) =
JOIN (pred(R1, R2, ..., Rn
  ...
    JOIN (
      JOIN (
        JOIN (R1, R2, pred(R1, R2)),
        R3, pred(R1, R2, R3)
      ),
      R4, pred(R1, R2, R3, R4)
    ),
  ...,
  Rn, pred(R1, R2, ..., Rn)
)
```

Restrukturierung (2)

Nur Beispiele!
Es gibt noch viel mehr!

- **Regeln (Forts.):**
  - (2) **Verbund ist kommutativ.**
  - (3) **Verbund ist assoziativ.**
  - (4) **Selektionen können zusammengefasst werden:**
    `SEL(SEL(R,pred1),pred2) = SEL(R,(pred1 AND pred2))`
  - (5) **Projektionen können zusammengefasst werden:**
    `PROJ(PROJ(R,L1),L2) = PROJ(R,L2)`
  - (6) **Projektion dürfen (in erweiterter Form) vorgezogen werden:**
    `PROJ(SEL(R,pred(M)),L) = PROJ(SEL(PROJ(R,(L∪M)),pred(M)),L)`
  - (7) **Selektion und Verbund dürfen vertauscht werden:**
    `SEL(JOIN(R,S,pred1),pred2(R)) = JOIN(SEL(R,pred2),S,pred1)`
  - (8) **Selektion darf mit Vereinigung und Differenz vertauscht werden:**
    `SEL(UNION(R,S),pred) = UNION(SEL(R,pred),SEL(S,pred))`
  - (9) **Selektion und Kreuzprodukt können zu Verbund zusammengefasst werden:**
    `SEL(CROSS(R,S),pred) = JOIN(R,S,pred)`
  - (10) **...**

Algorithmus zur Restrukturierung

- **Ziel: Zwischenergebnisse möglichst klein halten**
  - Vor allem Kreuzprodukt vermeiden

- **Vereinfachte Vorgehensweise (Heuristik)**
  - Komplexe Verbundoperationen zerlegen in binäre Verbunde (Bilden von binären Verbunden, Regel 1)
  - Selektionen mit mehreren Prädikat-Termen separieren in Selektionen mit jeweils einem Prädikat-Term (Regel 4)
  - **Selektionen so früh wie möglich ausführen**,
    - d.h. Selektionen hinunterschieben zu den Blättern des Anfragebaums (engl. selection push-down, Regeln 7 und 8)
  - **Selektionen und Kreuzprodukt zu Verbund zusammenfassen**,
    - wenn das Selektionsprädikat Attribute aus den beiden Relationen verwendet (Regel 9)

Algorithmus zur Restrukturierung (2)

- **Vereinfachte Vorgehensweise (Forts.)**
  - Einfache Selektionen wieder zusammenfassen,
    d.h. aufeinanderfolgende Selektionen (derselben Relation) gruppieren
    (Regel 4)
  - **Projektionen so früh wie möglich ausführen**,
    - d.h. Projektionen hinunterschieben zu den Blättern des Anfragebaums
      (engl. projection push-down, Regel 6)
    - Dabei aber die teure Duplikat-Eliminierung vermeiden!

Vorlesung Implementierung von Datenbanksystemen

12. Synchronisation

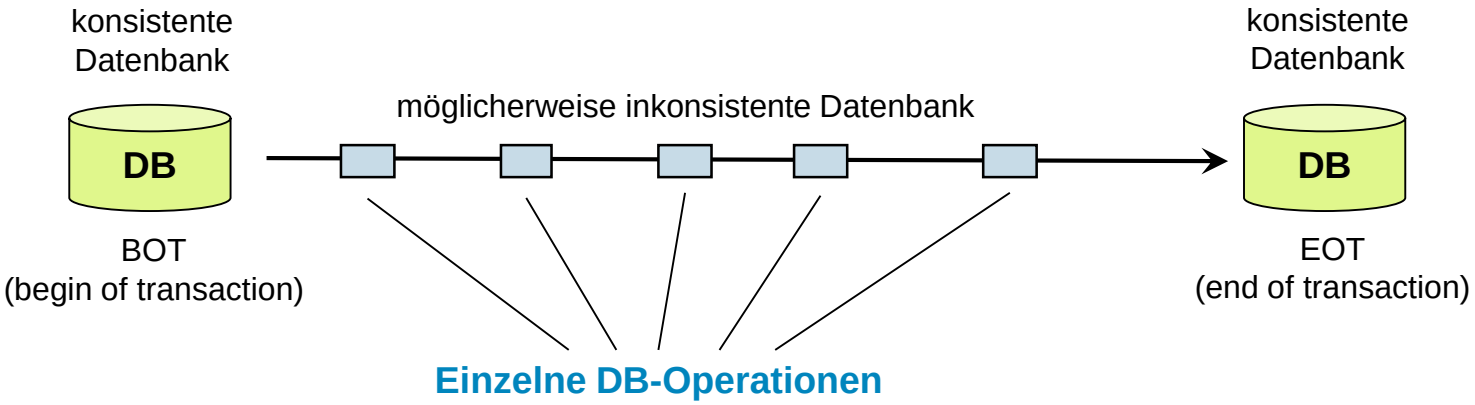
Prof. Dr. Klaus Meyer-Wegener

Wintersemester 2019/20

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

Implementierung von Transaktionen

- **Transaktion (Wiederholung)**
  - Zusammenfassung von aufeinander folgenden DB-Operationen, die eine Datenbank von einem konsistenten Zustand in einen wiederum konsistenten Zustand überführen



- **Beachte:**
  - Eine Transaktion muss nicht notwendigerweise in einem *anderen* konsistenten Zustand enden – es kann auch derselbe wie zu Anfang sein.
  - Transaktionen werden immer beendet:
    - Normal (**commit**): Änderungen sind permanent in der DB.
    - Anormal (**abort** / **rollback**): Bereits durchgeführte Änderungen werden zurückgenommen.

Anmerkungen zur Konsistenz in DBS

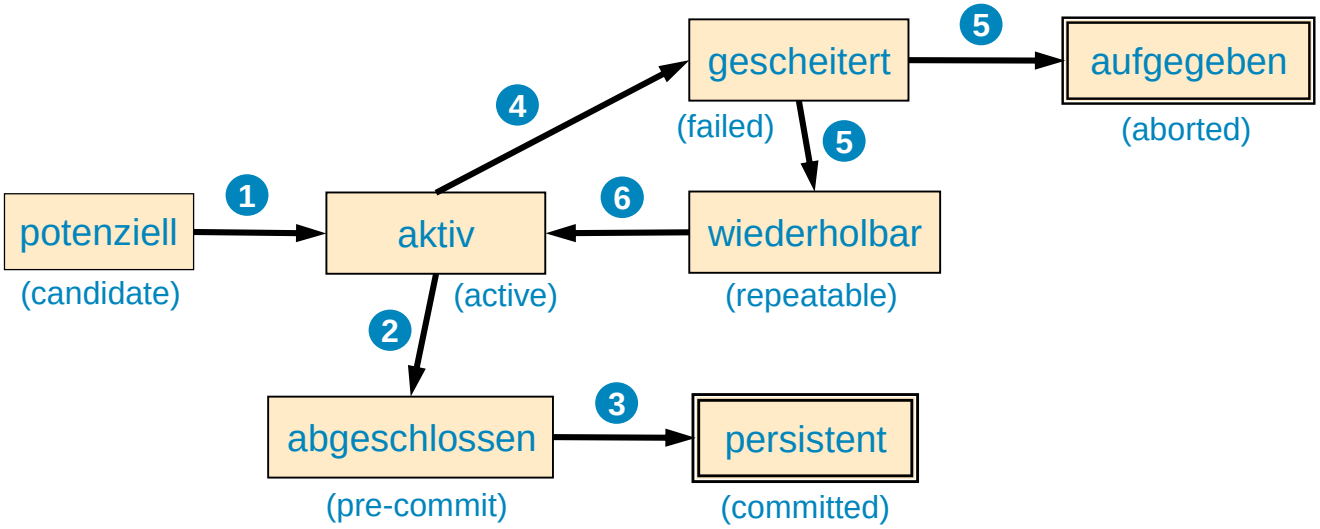
- **Beispiel**
  - Vorgegebene Konsistenzbedingung: Es muss immer x = y gelten.

```
BOT;                // Begin of Transaction; DB ist konsistent
read(x);
read(y);
x := x + 1;
write(x);           // DB ist zeitweilig inkonsistent
y := y + 1;
write(y);
EOT;                // End of Transaction; DB ist wieder konsistent
```

- **Arten von Konsistenz**
  - Datenbankkonsistenz
    - Alle (auf der DB definierten) Konsistenzbedingungen sind erfüllt.
  - **Transaktionskonsistenz** (operationelle Integrität)
    - Der nebenläufige Ablauf der Transaktionen ist korrekt.

Zustände von Transaktionen

- **Zustandsübergangsdiagramm**



1. Inkarnieren: TA ist angemeldet und wechselt in den Zustand aktiv
2. Beenden: TA ist beendet, aber Änderungen sind noch nicht permanent eingebracht
3. Festschreiben: Änderungen werden eingebracht
4. Abbrechen: TA ist fehlgeschlagen, aber noch nicht zurückgesetzt
5. Zurücksetzen: Änderungen werden rückgängig gemacht
6. Neustarten: TA wird wiederholt

ACID-Eigenschaften von Transaktionen

- **Atomarität (atomicity)**
  - Unteilbarkeit gemäß Transaktionsdefinition (Begin – End)
  - Alles-oder-Nichts-Prinzip, d.h. das DBS garantiert
    - entweder die vollständige Ausführung einer Transaktion
    - oder die Wirkungslosigkeit der Transaktion (und damit aller beteiligten Operationen).
- **Konsistenzerhaltung (consistency)**
  - Eine erfolgreiche Transaktion garantiert, dass alle Konsistenzbedingungen (Integritätsbedingungen) eingehalten worden sind.
- **Isolation (isolation)**
  - Mehrere Transaktionen laufen voneinander isoliert ab und benutzen keine (inkonsistenten) Zwischenergebnisse anderer Transaktionen.
- **Dauerhaftigkeit (durability)**
  - Alle Ergebnisse erfolgreicher Transaktionen müssen persistent gemacht werden (worden sein).

12.1 Synchronisation allgemein

- **Ziel**
  - Erhaltung der **Transaktionskonsistenz** (= operationelle Integrität) im Mehrbenutzerbetrieb
- **Gründe für den Mehrbenutzerbetrieb**
  - Verteilung der Clients generell (z.B. Geldausgabeautomat)
  - Prozessornutzung auch während systembedingter und benutzungsbedingter Transaktionsunterbrechungen (Wartezeiten, z.B. Ein-/Ausgabe)
  - Kommunikationsvorgänge in verteilten Systemen
- **Gegenstand der Synchronisation**
  - Vermeidung der gegenseitigen Beeinflussung von Lese- und Schreiboperationen
  - Verhinderung von **Anomalien** im Mehrbenutzerbetrieb

Vorgehensweise

- **Mögliche Anomalien bei fehlender Synchronisation betrachten**
  - Verlorengegangene Änderung (lost update, dirty write)
  - Abhängigkeit von nicht freigegebener Änderung (dirty read)
  - Inkonsistente Analyse (non-repeatable read)
  - (Phantom-Problem)
- **Unbefriedigende Lösung: Serialisierung**
  - Alle Transaktionen (TAen) nacheinander ausführen (Einbenutzerbetrieb)
  - Aber: führt bei langen TAen zu großen Wartezeiten für die anderen TAen (Scheduling-Fairness)
- **Statt dessen:**
  - Sicherstellen der operationellen Integrität
    durch eine "virtuelle" serielle Ausführung der TAen
    ("Logischer" Einbenutzerbetrieb! Als ob jede TA ganz allein auf der DB wäre)
- **Unterschiedliche Sichtweisen**
  - A posteriori: **Serialisierbarkeitstheorie**
  - A priori: **Sperrverfahren**

Beispiel für verlorengegangene Änderung

Gehaltsänderung TA_1

```
SELECT Gehalt INTO :gehalt
FROM Pers
WHERE PNr = 2345;

gehalt := gehalt + 2000;

UPDATE Pers
SET Gehalt = :gehalt
WHERE PNr = 2345;
```

Gehaltsänderung TA_2

```
SELECT Gehalt INTO :gehalt
FROM Pers
WHERE PNr = 2345;

gehalt := gehalt + 1000;

UPDATE Pers
SET Gehalt = :gehalt
WHERE PNr = 2345;
```

DB-Inhalt
(PNr, Gehalt)

2345	39.000
2345	41.000
2345	40.000

Zeit

- Konkurrierendes Verändern desselben (!) Datenelements
- Write-write dependency
- Lösung: exklusives Änderungsrecht für Schreiber

Beispiel für "Dirty Read"

Gehaltsänderung TA_1

```
UPDATE Pers
SET Gehalt = Gehalt + 1000
WHERE PNr = 2345;

ROLLBACK;
```

Gehaltsänderung TA_2

```
SELECT Gehalt INTO :gehalt
FROM Pers
WHERE PNr = 2345;

gehalt := gehalt * 1,05;

UPDATE Pers
SET Gehalt = :gehalt
WHERE PNr = 3456;
```

DB-Inhalt
(PNr, Gehalt)

2345	39.000
2345	40.000
3456	42.000
2345	39.000

Zeit

- Abhängigkeit von nicht freigegebener Änderung (eigentlich: rekursives Zurücksetzen)
- Write-read dependency
- Lösung:
  - Ermöglichen des isolierten Zurücksetzens
  - Lesen geänderter Daten immer erst dann, wenn sie freigegeben sind !

Beispiel für inkonsistente Analyse

Gehaltsänderungen TA_1

```
UPDATE Pers
SET Gehalt = Gehalt + 1000
WHERE PNr = 2345;
UPDATE Pers
SET Gehalt = Gehalt + 2000
WHERE PNr = 3456;
COMMIT;
```

Gehaltssumme TA_2

```
SELECT Gehalt INTO :g1
FROM Pers
WHERE PNr = 2345;

SELECT Gehalt INTO :g2
FROM Pers
WHERE PNr = 3456;

summe := g1 + g2;
```

DB-Inhalt
(PNr, Gehalt)

2345	39.000
3456	45.000
2345	40.000
3456	47.000

Zeit

- Während der Verarbeitung von TA_2 wird der Datenbestand durch TA_1 verändert.
- TA_2 sieht konsistente Zustände, jedoch unterschiedliche !
- Read-write dependency
- Lösung: Auch Leser müssen ihre Daten schützen – vor den Änderungen anderer

Abläufe (Schedules, Histories)

- **Kompakte Schreibweise für das, was passiert ist**
  - Leseoperation einer Transaktion *i* auf Datenobjekt *A*: $r_i[A]$
  - Schreiboperationen einer Transaktion *i* auf Objekt *A*: $w_i[A]$
  - Erfolgreicher Abschluss (commit) einer Transaktion: c_i
  - Abbruch (abort, rollback) einer Transaktion: a_i

- **Beispiele von oben in dieser Notation:**
  - Verlorengegangene Änderung:
    - r_1[PNr = 2345], r_2[PNr = 2345], w_1**[PNr = 2345]**, w_2**[PNr = 2345]**, c_1, c_2
  - "Dirty Read":
    - w_1**[PNr = 2345]**, r_2[PNr = 2345], w_2**[PNr = 3456]**, a_1, c_2
  - Inkonsistente Analyse:
    - r_2[PNr = 2345], w_1**[PNr = 2345]**, w_1**[PNr = 3456]**, r_2[PNr = 3456], c_1, c_2

12.2 Serialisierbarkeit

- **Ziel**
  - Garantie, dass beim verzahnten Ablauf mehrerer Transaktionen die Datenbank konsistent bleibt und jede Transaktion einen konsistenten Zustand sieht
- **Eigenschaft einer Transaktion (das C von ACID)**
  - Wenn eine Transaktion TA *allein* auf einer konsistenten DB ausgeführt wird, dann terminiert die TA (irgendwann) und hinterlässt die DB in einem konsistenten Zustand.
  - **... folglich:**

    Wenn alle Transaktionen *seriell* (hintereinander) ausgeführt werden, dann bleibt die Konsistenz der Datenbank erhalten.

- **Beispiel**

```
TA1: read(A);
     A := A - 50;
     write(A);
     read(B);
     B := B + 50;
     write(B);
```

```
TA2: read(A);
     temp := A * 0.1;
     A := A - temp;
     write(A);
     read(B);
     B := B + temp;
     write(B);
```

Beispiel zur Serialisierbarkeit

- **Zwei serielle Abläufe denkbar**
  - Anfangswerte A = 1000 und B = 2000
  - **Ablauf 1**: TA1 ; TA2 (Semikolon = "vor")
    - TA1: A = **950** B = **2050**
    - TA2: A = **855** B = **2145**
  - **Ablauf 2**: TA2 ; TA1
    - TA2: A = **900** B = **2100**
    - TA1: A = **850** B = **2150**
- **Merke:**
  - Beide Abläufe sind korrekt,
    obwohl sie unterschiedliche (aber jeweils konsistente) Ergebnisse liefern.
  - Bei n Transaktionen sind **n!** serielle Abläufe möglich.
- **Problem**
  - Nicht alle verzahnten Abläufe sind korrekt – manche aber schon.

Beispiel für nicht korrekten parallelen Ablauf

- **Beispiel für verzahnten, jedoch nicht korrekten Ablauf**
  - **Ablauf 3**:

```
TA1: read(A);
     A := A - 50;
                    TA2: read(A);
                         temp := A * 0.1;
                         A := A - temp;
                         write(A);
                         read(B);
     write(A);
     read(B);
     B := B + 50;
     write(B);
                         B := B + temp;
                         write(B);
```

  - liefert A = **950** B = **2100**
    - "Lost Update" (sogar zweimal) !!!
  - Kompakt: $r_1[A]$, $r_2[A]$, $\boldsymbol{w_2[A]}$, $r_2[B]$, $\boldsymbol{w_1[A]}$, $r_1[B]$, $\boldsymbol{w_1[B]}$, $\boldsymbol{w_2[B]}$, $\boldsymbol{c_1}$, $\boldsymbol{c_2}$

Serialisierbare Abläufe

- **Definition: serialisierbarer Ablauf**
  - Ein Ablauf von Transaktionen ist **serialisierbar**, wenn er zu irgendeinem seriellen Ablauf der in ihm enthaltenen Transaktionen äquivalent ist.
- **Definition: Äquivalenz von Abläufen**
  - Seien TA_i und TA_j beliebige erfolgreiche Transaktionen und H und G zwei dazugehörige Abläufe.
    H und G sind **äquivalent**, wenn für alle Operationen auf einem beliebigen Datenobjekt A folgende Bedingungen gelten:

$$r_i[A] <_H w_j[A] \Leftrightarrow r_i[A] <_G w_j[A]$$
$$w_i[A] <_H r_j[A] \Leftrightarrow w_i[A] <_G r_j[A]$$
$$w_i[A] <_H w_j[A] \Leftrightarrow w_i[A] <_G w_j[A]$$

  - D.h. bei solchen Operations-Paaren muss die Reihenfolge in beiden Abläufen gleich sein.
    - Man sagt auch: Sie stehen in **Konflikt** miteinander.
  - Durch Vertauschen von zwei benachbarten Operationen, die nicht in Konflikt zueinander stehen, kann man einen äquivalenten Ablauf erzeugen.
  - (Etwas ungenau: Endzustand der DB evtl. unterschiedlich ...)

Beispiel für korrekten parallelen Ablauf

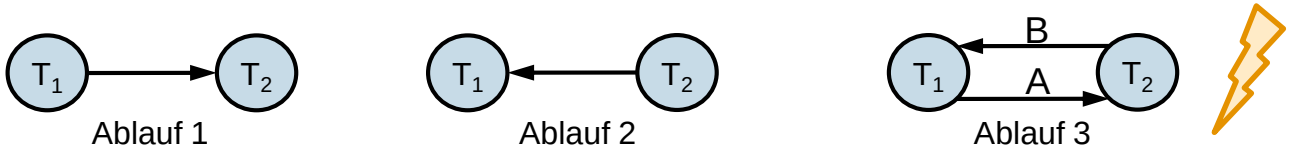
- **Beispiel für verzahnten, jedoch korrekten (serialisierbaren) Ablauf**
  - **Ablauf 4**:

```
TA1:read(A);
    A := A - 50;
    write(A);
                    TA2:read(A);
                        temp := A * 0.1;
                        A := A - temp;
                        write(A);
    read(B);
    B := B + 50;
    write(B);
                        read(B);
                        B := B + temp;
                        write(B);
```

  - liefert A = **855** B = **2145**
    - Also wie (serieller) Ablauf 1: TA1 ; TA2
  - Kompakt: $r_1[A]$, $\boldsymbol{w_1[A]}$, $r_2[A]$, $\boldsymbol{w_2[A]}$, $r_1[B]$, $\boldsymbol{w_1[B]}$, $r_2[B]$, $\boldsymbol{w_2[B]}$, $\boldsymbol{c_1}$, $\boldsymbol{c_2}$

Nachweis der Serialisierbarkeit

- **Abhängigkeitsgraph**
  - **Knoten**: einzelne Transaktionen
  - **Kanten**: Abhängigkeiten (Konflikt) zwischen zwei Transaktionen
    - Zwei TAen greifen auf dasselbe Objekt mit nicht reihenfolgeunabhängigen Operationen zu
      (z.B. Schreib-/Lese-, Lese-/Schreib-, Schreib-/Schreib-Konflikte).
  - Serialisierbarkeit liegt vor, wenn der Abhängigkeitsgraph *keine Zyklen* enthält.
  - Durch topologisches Sortieren ($O(N^2)$) erhält man die möglichen serialisierbaren Abläufe.
- **Beispiel:**



Lies: "T_1 vor T_2"

T_1 liest A, bevor T_2 A schreibt,
T_2 liest B, bevor T_1 B schreibt

  - Mehr dazu, wesentlich ausführlicher (und formeller!) in Vorl. Transaktionssysteme

Zusammenfassung Serialisierbarkeit

- **Zwei Abläufe sind äquivalent, wenn**
  - sie dieselben erfolgreich abgeschlossenen Transaktionen enthalten und
  - ihre Abhängigkeitsgraphen gleich sind.
- **Verhindert werden müssen (Berenson et al., 1995):**
  - Phänomen P0 (Dirty Write):
    - $w_1[x] \ldots w_2[x] \ldots ((c_1$ oder $a_1)$ und $(c_2$ oder $a_2)$ in beliebiger Reihenfolge)
  - Phänomen P1 (Dirty Read):
    - $w_1[x] \ldots r_2[x] \ldots ((c_1$ oder $a_1)$ und $(c_2$ oder $a_2)$ in beliebiger Reihenfolge)
  - Phänomen P2 (Non-repeatable Read):
    - $r_1[x] \ldots w_2[x] \ldots ((c_1$ oder $a_1)$ und $(c_2$ oder $a_2)$ in beliebiger Reihenfolge)
  - Phänomen P3 (Phantom):
    - $r_1[P] \ldots w_2[y$ in $P] \ldots ((c_1$ oder $a_1)$ und $(c_2$ oder $a_2)$ in bel. Reihenf.)
    - P steht für die Menge der Datenobjekte, die ein Prädikat erfüllen

Klammern bei c_i und a_i bedeuten nur Gruppierung, nicht "optional"

12.3 Implementierungsmethoden

- **Realisierung eines logischen Einbenutzerbetriebs**
  - Einführung von sog. **Sperren** (locks) für Zugriffe auf Datenobjekte
  - Für jedes benutzte Datenobjekt zentral in einer **Sperrtabelle** die Nutzung durch bestimmte Transaktionen registrieren
- **Arten von Sperren**
  - **X** (eXclusive)-Sperre (= Schreibsperre)
  - **S** (shared)-Sperre (= Lesesperre)
  - **Regeln für den Umgang mit Sperren**
    - Jedes Datenobjekt, auf das zugegriffen werden soll, muss *vorher* gesperrt werden.
    - Eine Transaktion fordert eine Sperre, die sie bereits besitzt, nicht noch einmal an.
    - Eine Transaktion muss die von anderen Transaktionen gesetzten Sperren beachten.
    - Am Ende einer Transaktion (und erst dann!) sind alle Sperren wieder freizugeben.

    (Eswaran, Gray, Lorie und Traiger, 1976)
  - **Serialisierbarkeit ist gewährleistet, wenn diese Regeln eingehalten werden!**

Synchronisation mit Sperren

- **Kompatibilitätsmatrix**
  - gibt Auskunft, ob eine Sperranforderung für ein (möglicherweise bereits gesperrtes) Objekt gewährt werden kann

Sperrmodus des Objekts

Sperranforderung der Transaktion

	keine Sperre	**S-Sperre**	**X-Sperre**
S-Sperre	Anforderung wird gewährt +	Anforderung wird gewährt +	Anforderung wird **nicht** gewährt –
X-Sperre	Anforderung wird gewährt +	Anforderung wird **nicht** gewährt –	Anforderung wird **nicht** gewährt –

Synchronisation mit Sperren (2)

- **Wann Sperren erwerben?**
  - **Statisches** Sperren
    - Zu Beginn der Transaktion alle Sperren anfordern ("preclaiming")
    - Nachteil: Man muss alles sperren, was man brauchen *könnte*.
  - **Dynamisches** Sperren
    - Während der Transaktion Sperren nach Bedarf anfordern
    - Nachteil: Verklemmungen (deadlocks) können auftreten.

- **Wann Sperren freigeben?**
  - Sperren müssen *bis zum Ende der Transaktion* gehalten werden, um Serialisierbarkeit zu garantieren
  - Abschwächung (früher freigeben) möglich
    - Siehe weiterführende Vorl. oder Literatur

Probleme mit Sperren

- **Sperrgranulat Tupel**
  - Nicht immer effizient
    - Aufwändig bei Transaktionen, die viele (alle) Tupel einer Relation benötigen
    - Große Sperrtabellen, hohe Verwaltungskosten
  - Nicht ausreichend, um alle Fehlerklassen auszuschließen
- **Phantom-Problem**
  - Sperren nur auf *existierende* Tupel
  - Nicht existierende Tupel können jederzeit eingefügt werden – sind dann aber nicht gesperrt (= Phantome).

Lesetransaktion
(Gehaltssumme bestimmen)

```
SELECT SUM(Gehalt)
INTO :summe1
FROM Pers
WHERE  ANr = 17;
        • • •
SELECT SUM(Gehalt)
INTO :summe2
FROM Pers
WHERE  ANr = 17;
if summe1 ≠ summe2
then <Fehler>;
```

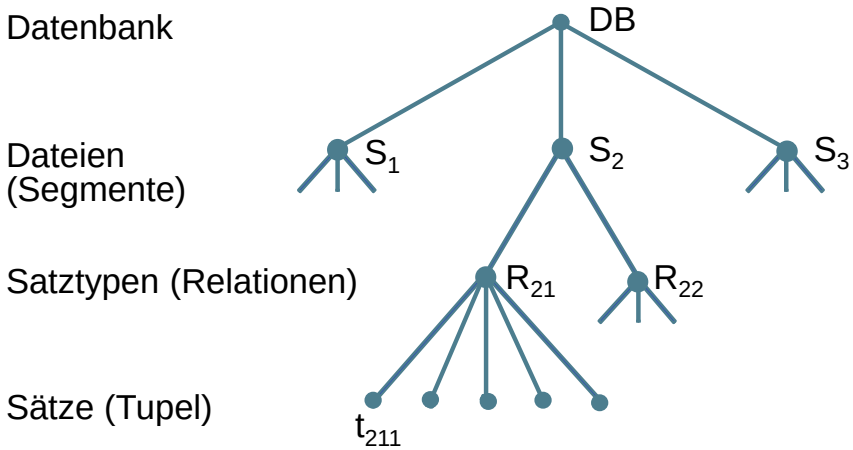
Änderungstransaktion
(Einfügen eines neuen Angestellten)

```
INSERT INTO Pers
  (PNr, ANr, Gehalt)
  VALUES (4567, 17, 55.000);
COMMIT;
```

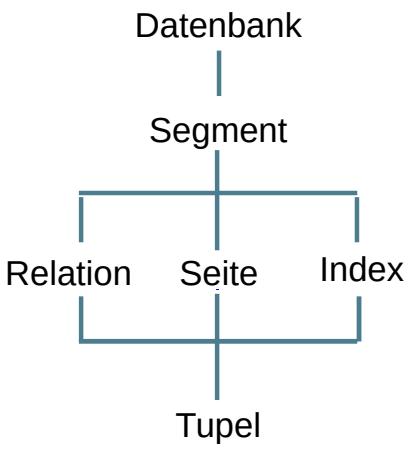
Zeit

Lösungsansatz

- **Hierarchische Schachtelung der Datenobjekte**
  - erlaubt Flexibilität bei der Wahl der Sperrgranulate
    - Synchronisation langer TAen auf Relationenebene
    - Synchronisation kurzer TAen auf Tupelebene



a) Beispiel einer Objekthierarchie



b) nicht-hierarchische Granularitäten

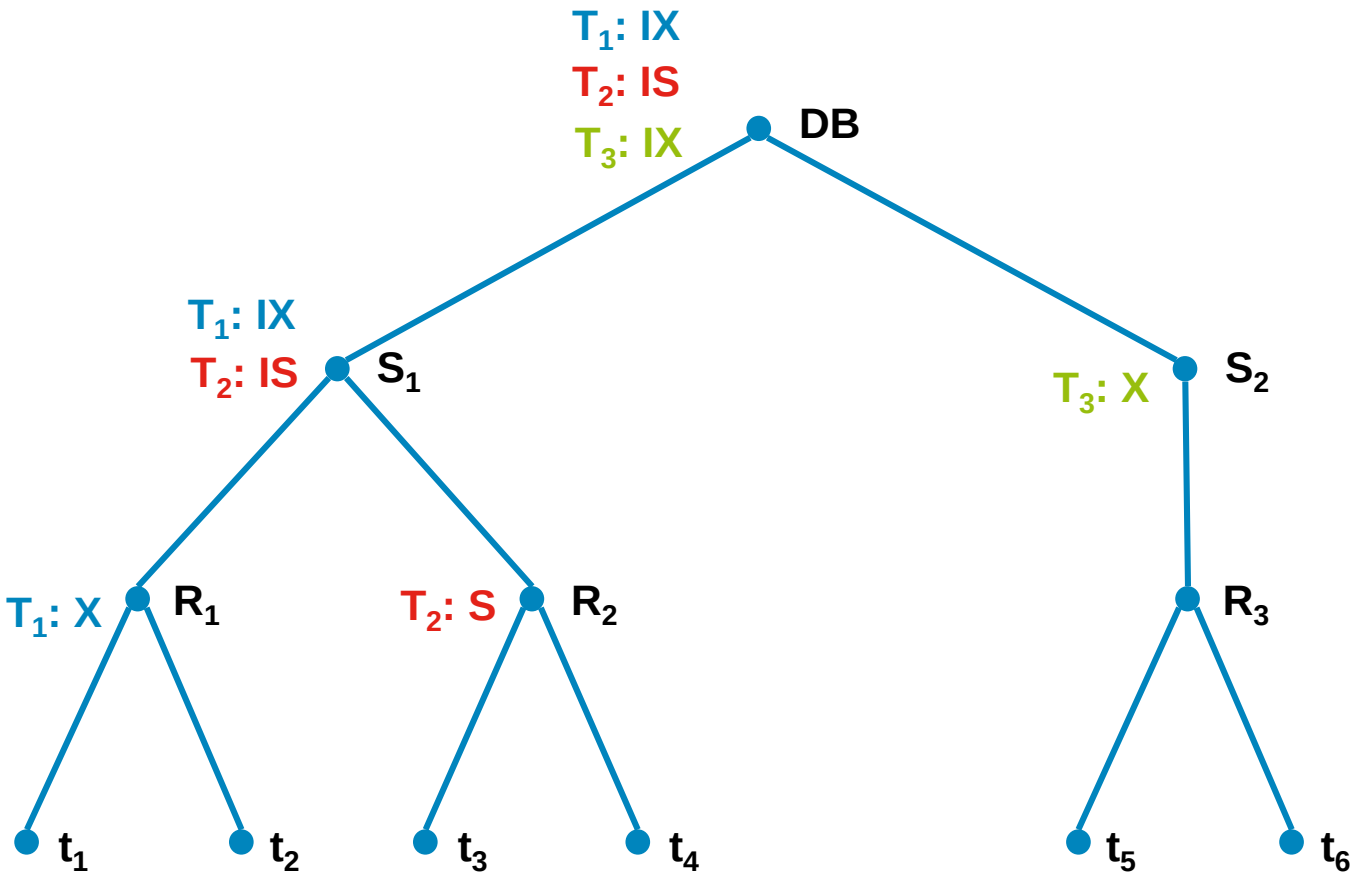
Anwartschaftssperren

- **Nachteil von reinen S- und X-Sperren**
  - Alle Nachfolgeknoten implizit mitgesperrt
  - Alle Vorgängerknoten ebenfalls sperren, um Unverträglichkeiten zu erkennen
    - X-Sperre auf DB: erzwingt Einbenutzerbetrieb
    - S-Sperre auf DB: nur Lese-Transaktionen können parallel laufen
- **Verwendung von Anwartschaftssperren (intention locks)**
  - I-Sperre oder Sperranzeige
    - **IS-Sperre (intention share)**,
      falls auf untergeordnete Objekte nur lesend zugegriffen wird
    - **IX-Sperre (intention exclusive)**,
      falls auf untergeordnete Objekte auch schreibend zugegriffen wird
  - Ersetzt die Sperren für Datenobjekte auf den höheren Hierarchie-Ebenen
  - Nutzung einer Untermenge wird angezeigt, aber in der Untermenge werden noch explizit Sperren gesetzt

Verwendung von Anwartschaftssperren

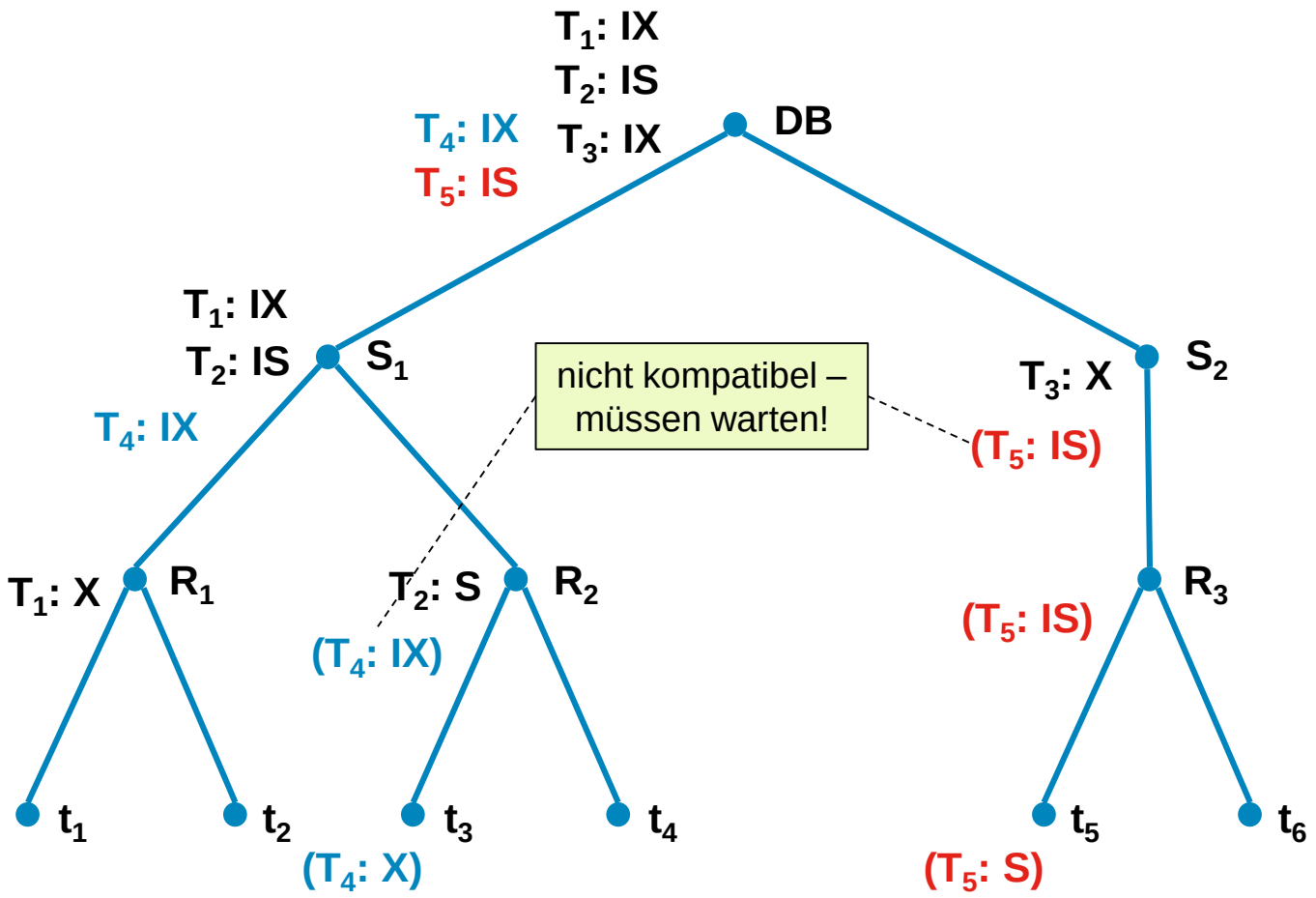
- **Top Down beim Erwerb von Sperren**
  - Bevor ein Knoten mit S oder IS gesperrt werden darf, müssen alle Vorgänger in der Hierarchie im IX- oder im IS-Modus gesperrt worden sein.
  - Bevor ein Knoten mit X oder IX gesperrt werden darf, müssen alle Vorgänger in der Hierarchie im IX-Modus gesperrt worden sein.
- **Bottom Up bei der Freigabe von Sperren**
  - Freigabe der Sperren von unten nach oben
  - Bei keinem Knoten darf die Sperre aufgehoben werden, wenn die betreffende Transaktion noch Nachfolger dieses Knotens gesperrt hat.
- **Beispiel**
  - T_1: benötigt exklusive Sperre von R_1 in S_1
  - T_2: benötigt Lesesperre von R_2 in S_1
  - T_3: benötigt exklusive Sperre von Segment S_2

Beispiel zu Anwartschaftssperren



Beispiel zu Anwartschaftssperren (2)

- T_4: benötigt exklusive Sperre auf Tupel t_3
- T_5: benötigt Lesesperre auf Tupel t_5



Noch eine Anwartschaftssperre

- **Kombination von Sperre und Sperranzeige**

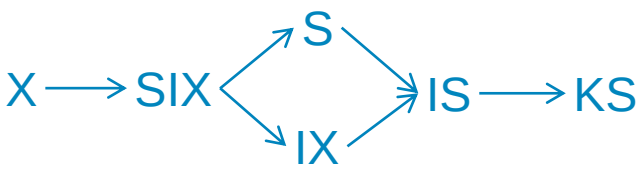
  SIX = S + IX (share and intention exclusive)

  - Sperrt das Objekt in S-Modus
  - Verlangt auf tieferen Hierarchieebenen nur noch IX- oder X-Sperren für zu ändernde Objekte
  - Sinnvoll für den Fall, dass alle Tupel einer Relation gelesen und nur einige davon geändert werden
    - X-Sperre auf Relation zu restriktiv
    - IX-Sperre auf Relation verlangt Sperren jedes Tupels zum Lesen
  - (Die analoge Lösung XIS ist natürlich Unsinn.)

Kompatibilitätsmatrix

	IS	IX	S	SIX	X
IS	+	+	+	+	–
IX	+	+	–	–	–
S	+	–	+	–	–
SIX	+	–	–	–	–
X	–	–	–	–	–

- Darstellung der Sperrmodi in einer Halbordnung (Dominanz-Relation):
  - "→" bedeutet: Wenn man die linke Sperre hat, kann man mehr machen als mit der rechten.



Weitere Probleme bei der Sperrverwaltung

- **Probleme bei der Implementierung von Sperren**
  - Sperranforderung und -freigabe sollten sehr schnell erfolgen, da sie sehr häufig benötigt werden.
  - Explizite, tupelweise Sperren führen u.U. zu umfangreichen Sperrtabellen und großem Zusatzaufwand.
  - Halten der Sperren bis Transaktionsende führt häufig zu langen Wartezeiten (starke Serialisierung).
  - Häufig berührte Zugriffspfade können zu Engpässen werden.
  - Eigenschaften des Schemas können "hot spots" erzeugen – Datenelemente, auf die fast alle Transaktionen zugreifen müssen.
- **Mögliche Optimierungen**
  - Änderungen auf privaten Objektkopien (verkürzte Dauer exklusiver Sperren)
  - Nutzung mehrerer Objektversionen
  - Spezialisierte Sperren (Nutzung der Semantik von Änderungsoperationen)

Alternative: "Optimistische" Verfahren

- **Erst einmal ohne Sperren einfach zugreifen**
  - Zugriffe aber protokollieren:
    - Menge der gelesenen Datenobjekte
    - Menge der geschriebenen Datenobjekte
- **Dann kurz vor Ende der Transaktion:**
  - Also in der Ausführung des Commit
  - Lese- und Schreibmengen der abschließenden Transaktion mit den Lese- und Schreibmengen der anderen gerade laufenden Transaktionen vergleichen
    - Verschiedene Möglichkeiten des Vergleichs – siehe Literatur
  - Bei Überschneidungen (= Zugriffskonflikten) muss die Transaktion, die gerade abschließen will, zurückgesetzt werden.
    - Deshalb "optimistisch": Ohne Überschneidungen klappt es, und sogar effizienter, weil ohne Sperren (und Wartezeiten).

Entstehung von Verklemmungen (Deadlocks)

- **Beispiel eines elementaren Deadlocks**

```
TA1 hält X-Sperre auf A        TA2 hält X-Sperre auf B
TA1 benötigt B zum Beenden     TA2 benötigt A zum Beenden
```

- **Notwendige Voraussetzungen für einen Deadlock**
  - Gleichzeitiger Zugriff
  - Exklusive Zugriffsanforderungen (X-Sperren)
  - Anfordernde TA besitzt bereits Sperren auf Datenobjekten
  - Keine vorzeitige Freigabe von Sperren auf Datenobjekten (non-preemption)
  - Zyklische Wartebeziehungen zwischen zwei oder mehr Transaktionen

Behandlung von Verklemmungen

- **Lösungsmöglichkeiten**
  - **Timeout**
    - Transaktion nach festgelegter Wartezeit auf eine Sperre zurücksetzen
    - Bestimmung des richtigen Timeout-Werts problematisch
  - **Verhütung** (Prevention)
    - Keine Laufzeitunterstützung zur Deadlock-Behandlung erforderlich
    - Bsp.: Preclaiming (siehe oben, in DBS i. Allg. nicht praktikabel)
  - **Vermeidung** (Avoidance)
    - Potenzielle Deadlocks im voraus (in dem Moment, wo eine TA auf eine andere warten muss) erkennen und durch entsprechende Maßnahmen vermeiden
    - $\Rightarrow$ Laufzeitunterstützung nötig
  - **Erkennung** (Detection)
    - Explizites Führen eines Wartegraphen (wait-for graph) und darin Zyklensuche
    - Auflösung durch Zurücksetzen einer oder mehrerer am Zyklus beteiligter TA (z. B. den Verursacher oder "billigste" TA zurücksetzen)

12.4 Zusammenfassung Synchronisation

- **Serialisierung paralleler Abläufe**
  - Bereitstellung eines logischen Einbenutzerbetriebs
  - Vermeidung von Anomalien
    - Lost updates, dirty reads, inkonsistente Analysen
- **Serialisierbarkeitstheorie**
  - Ein Ablauf ist serialisierbar, wenn er zu einem seriellen Ablauf äquivalent ist.
- **Implementierung: Sperrverfahren**
  - Einfache S- und X-Sperren
  - Zwei-Phasen-Sperrprotokoll
  - Hierarchisches Sperrkonzept
    - Anwartschaftssperren (I-Sperren)
  - Verklemmungen

Literatur

Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, Patrick E. O'Neil: A Critique of ANSI SQL Isolation Levels. In: Michael J. Carey (Hrsg.), Donovan A. Schneider (Hrsg.): *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, San Jose, California, May 22-25, 1995. ACM Press 1995, S. 1-10

Kapali P. Eswaran, Jim Gray, Raymond A. Lorie, Irving L. Traiger: The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM* 19(11): 624-633 (1976)

Vorlesung Implementierung von Datenbanksystemen

14. Beispiel Oracle Database

Prof. Dr. Klaus Meyer-Wegener

Wintersemester 2019/20

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

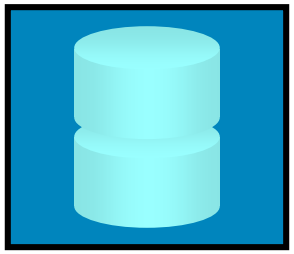
Überblick

- **Ausschnitte aus den Oracle-Schulungsunterlagen**
  - Oracle Database 11g: Administration Workshop I,
  - Oracle Database 11g: Administration Workshop II und
  - Oracle9i Database Administration Fundamentals I

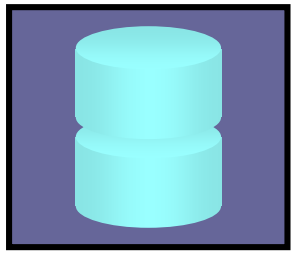
- **Storage Architecture**
- **Index Structures**
- **Server Architecture**
  - Memory Structure
  - Process Structure
- **Interacting with the Database**
- **Redo, Undo, Recovery**

(von mir umgeordnet und die PowerPoint-Notizen auf Deutsch in die Folien integriert)

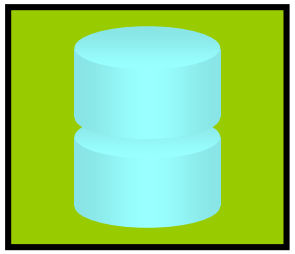
Database Storage Architecture



Control files

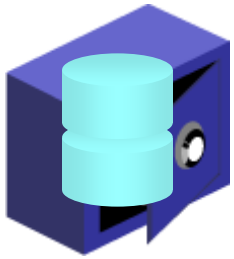


Data files

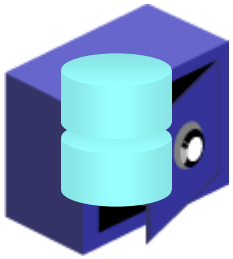


Online redo log files

Parameter file

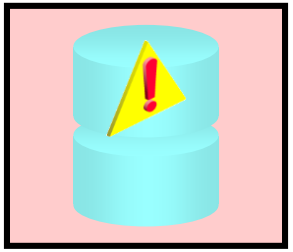


Backup files



Archived redo log files

Password file



Alert log and trace files

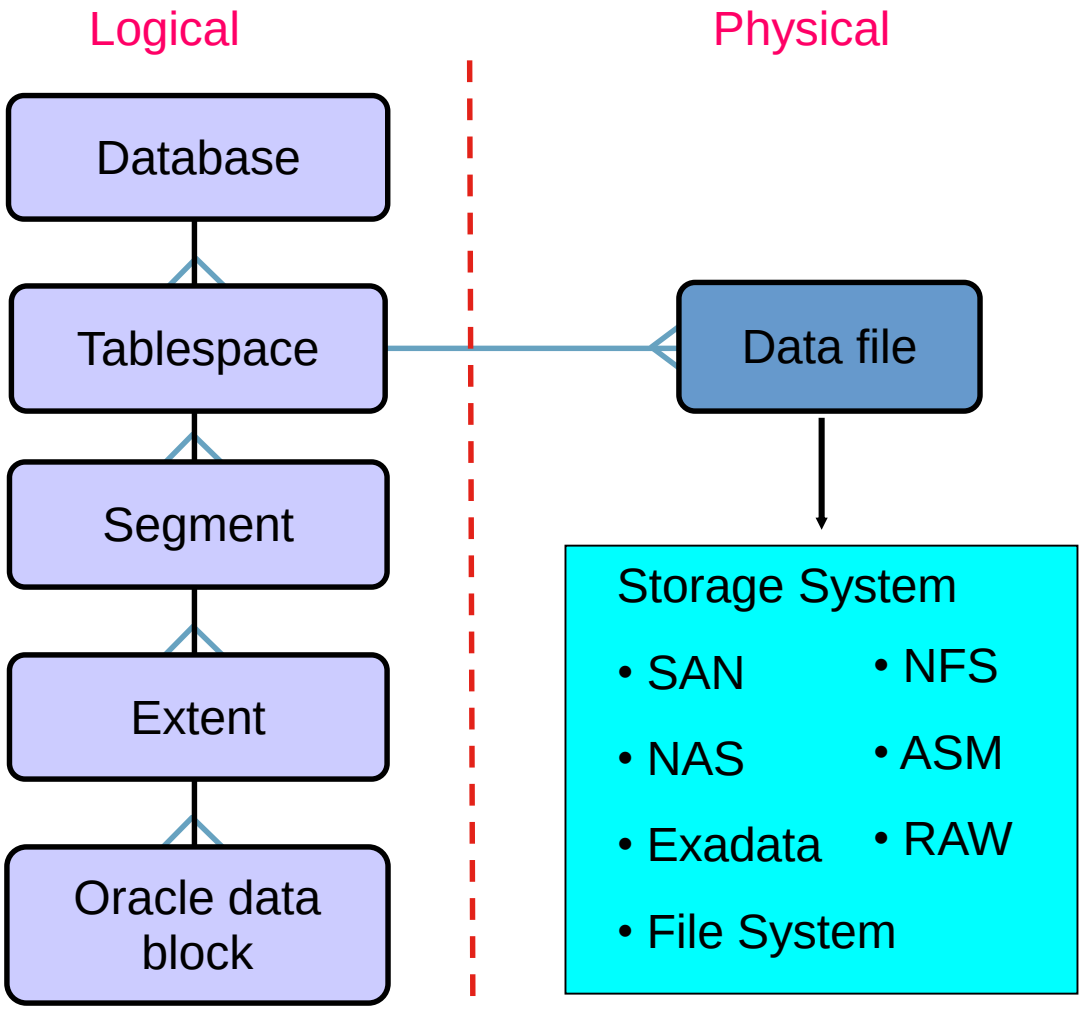
Dateien

- **Control files:**
  - Zentraler Einstiegspunkt – unbedingt erforderlich
  - Metadaten für Backup
- **Data files:**
  - Die eigentlichen Daten, Metadaten, Schema (Data Dictionary)
- **Online redo log files:**
  - Temporäre Protokolldatei (auch für Undo)
- **Parameter file:**
  - Konfiguration der DB-Instanz
- **Password file:**
  - Erlaubt Administratoren die Fernwartung der DB-Instanz
- **Backup files:**
  - Archivkopie der Datenbank
- **Archived redo log files:**
  - Inkrementelle Archivkopie-Dateien

Dateien (2)

- **Trace files:**
  - Wird von jedem Server- und Hintergrund-Prozess geschrieben
  - Aufzeichnung interner Fehler mit Zusatzinfo
  - Teils für den Administrator, teils für Oracle Support Service
- **Alert log file:**
  - Spezielle Trace-Einträge
  - Chronologisches Protokoll von Nachrichten und Fehlern
  - Sollte regelmäßig überprüft werden

Logical and Physical Database Structures



Logische und physische DB-Strukturen

- **Databases, Tablespaces und Data Files (Dateien)**
  - Datenbank aufgeteilt in zwei oder mehr Tablespaces
  - Eine Datei (oder mehrere) pro Tablespace
    für alle Daten aller Segmente im Tablespace
- **Tablespaces**
  - gruppieren zusammenhängende Strukturen
  - Z.B. alle Segmente einer bestimmten Anwendung,
    um die Administration zu vereinfachen
- **Data Blocks**
  - Seite
  - Kleinstes Granulat
  - Größe wird für jeden Tablespace bei seiner Erzeugung festgelegt
  - Einheit der Speicher-Allokation und -Freigabe

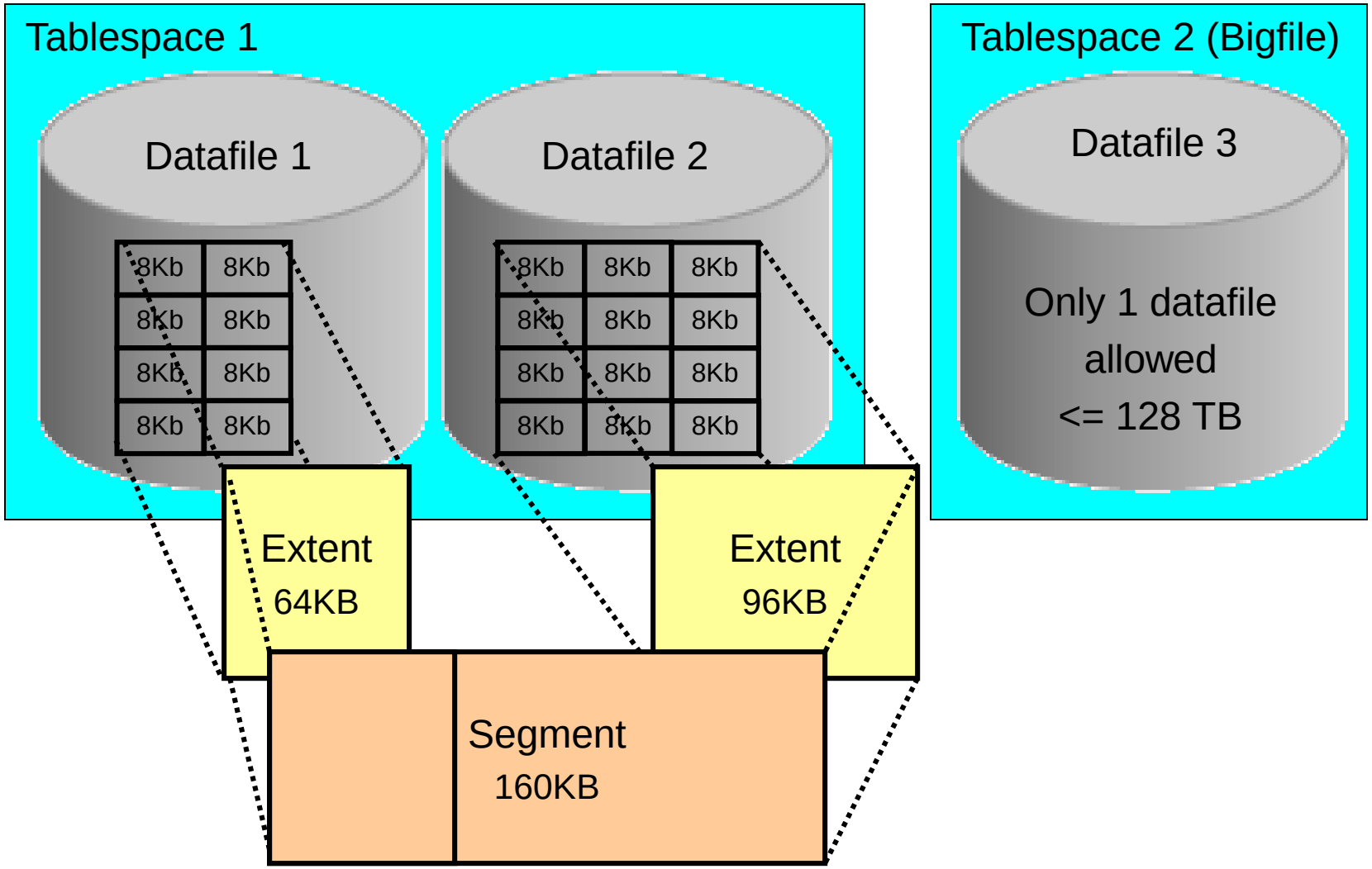
Logische und physische DB-Strukturen (2)

- **Extents**
  - Anzahl von Blöcken, die in einer Allokation zusammen angefordert wurden
  - Speichern eine bestimmte Art von Information
  - Logisch zusammenhängend, aber evtl. physisch verstreut (RAID striping, Dateisystem-Implementierung)
- **Segments**
  - Menge von Extents für eine bestimmte logische Struktur
- **Data segments**
  - Jede nicht geclusterte, nicht indexierte Tabelle hat ein eigenes Segment.
  - Ausnahmen: externe Tabellen, globale temporäre Tabellen, partitionierte Tabellen (dort ein Segment pro Partition)
  - Ein Cluster hat nur ein Segment für alle seine Tabellen.

Logische und physische DB-Strukturen (3)

- **Index segments**
  - Für jeden Index eins, mit allen seinen Daten
  - Bei partitionierten Indexen hat jede Partition eins.
- **Undo segments**
  - Ein Undo-Tablespace für jede DB-Instanz,
    enthält etliche Undo-Segmente
  - Verwendet für die Bereitstellung eines konsistenten DB-Zustands
    für lesende Transaktionen (während schon Änderungen laufen)
  - Und natürlich für Undo
- **Temporary segments**
  - Für die Dauer einer SQL-Anweisung
  - Entweder für jeden Benutzer eins oder für alle zusammen

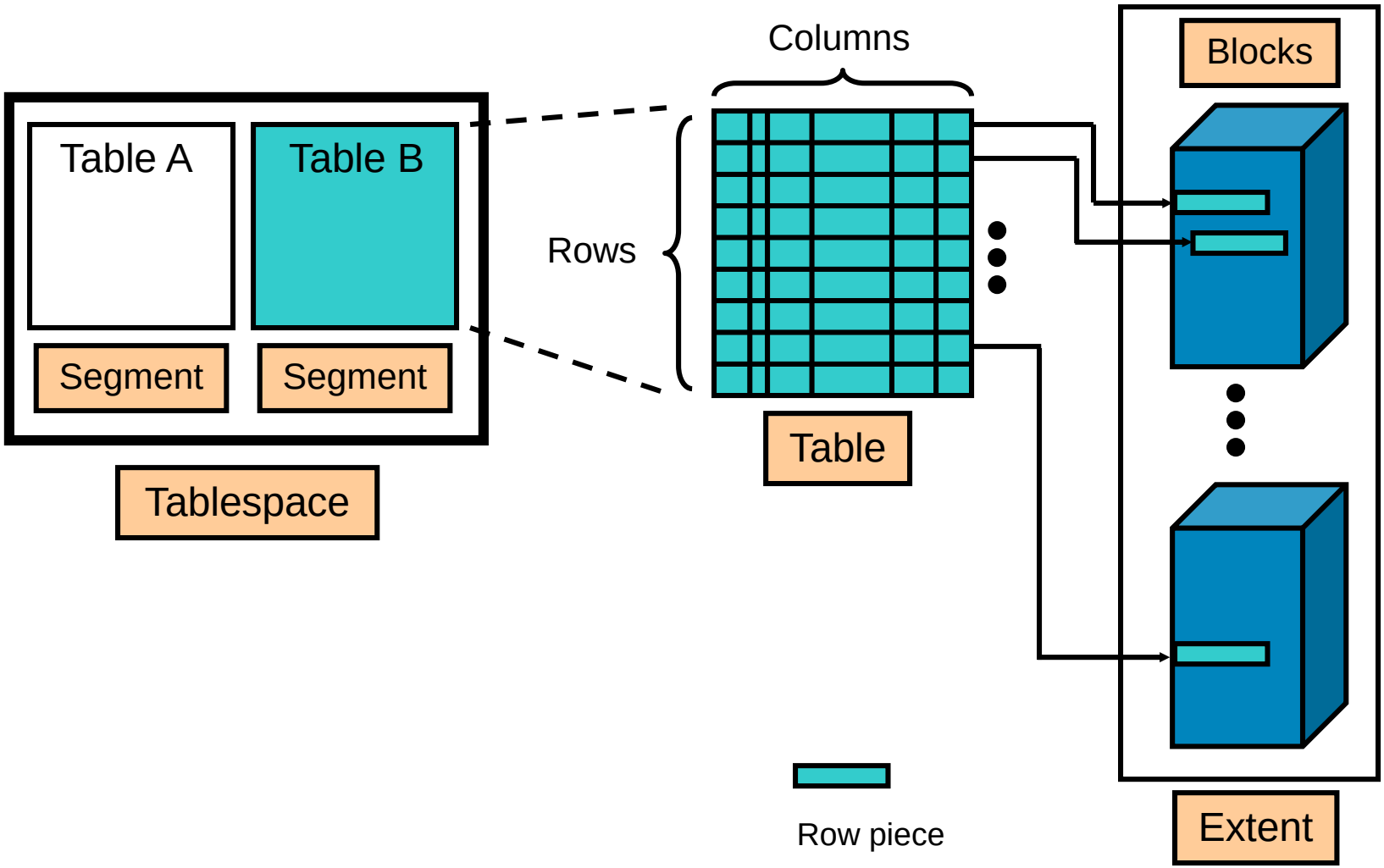
Tablespaces and Data Files



Tablespaces und Dateien

- **Tablespaces (Wdh.)**
  - Fassen logische Einheiten zusammen
  - Zwei oder mehrere: `SYSTEM` und `SYSAUX`
  - Verwenden eine oder mehrere Dateien für die Speicherung
- **Beispiel auf der letzten Folie:**
  - Tablespace 1 besteht aus zwei Dateien.
  - Segment von 160 KB Größe geht über beide Dateien, besteht deshalb aus zwei Extents.
  - Erstes Extent hat 64 KB und liegt in Datei 1, zweites hat 96 KB und liegt in Datei 2.
  - Beide Extents bestehen aus zusammenhängenden Blöcken von je 8 KB.
- **Bigfile Tablespaces:**
  - Nur eine Datei (die oft sehr groß wird)
  - Maximale Größe durch Art der Row-ID (siehe unten) vorgegeben: Blockgröße des Tablespace multipliziert mit 2^{36}, also z.B. 128 TB für eine Blockgröße von 32 KB.

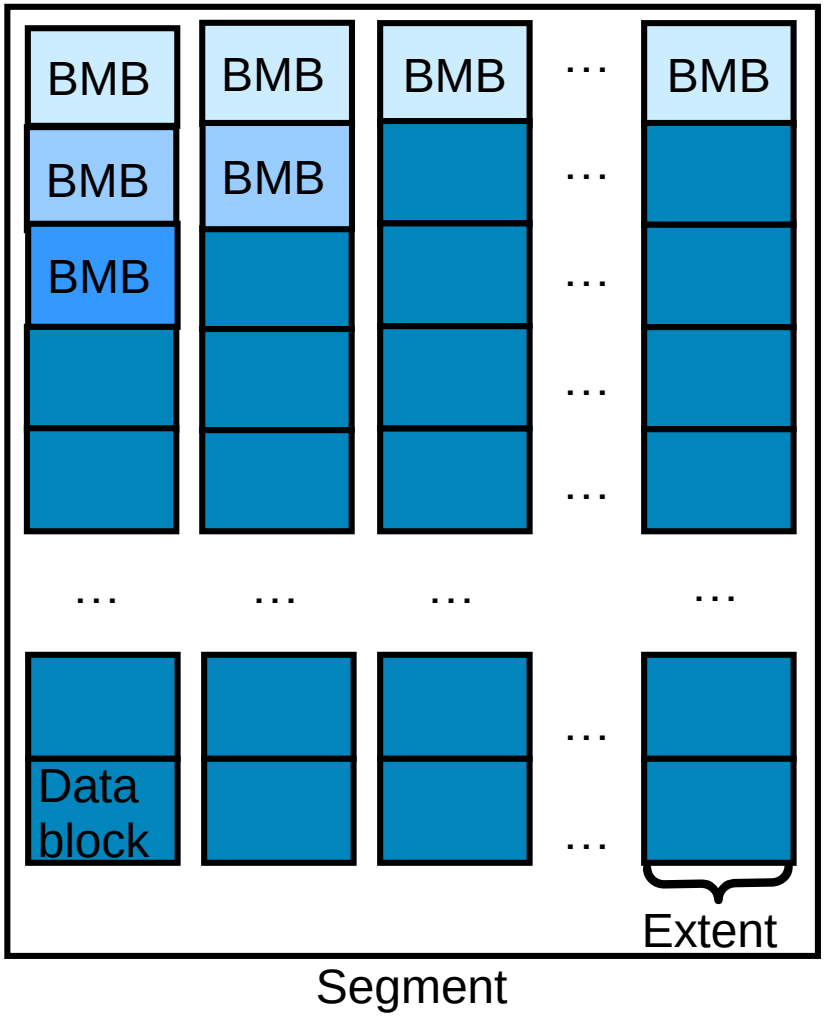
How Table Data Is Stored



Speicherung von Tabellen

- **Erzeugung**
  - Bei Erzeugung einer Tabelle
    wird ein Segment angelegt für ihre Daten.
- **Zeilen (Tupel, rows)**
  - In Blöcken abgelegt als Zeilen-Stücke (row piece, Sätze)
  - Zeile in mehreren Stücken,
    - wenn zu lang für einen Block (chained row)
    - oder durch Änderung zu groß geworden (migrated row)
  - Auch bei Tabellen mit mehr als 255 Spalten
    - Dann evtl. sogar mehrere Stücke im selben Block (intra-block chaining)

Free Space Management Within Segments

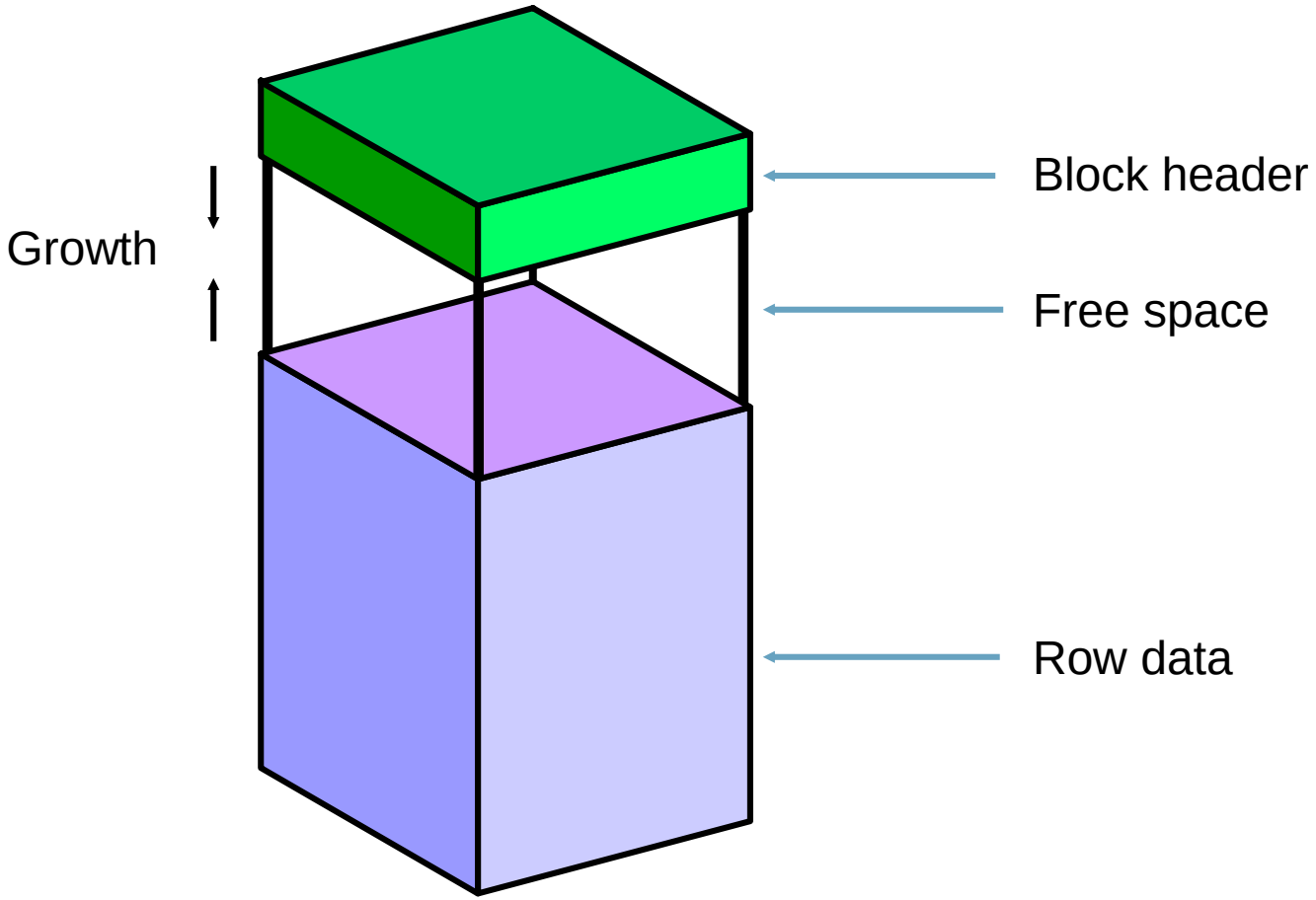


- Tracked by bitmap blocks (BMBs) in segments
- Benefits:
- More flexible space utilization
- Run-time adjustment
- Multiple process search of BMBs

Freispeicherverwaltung in Segmenten

- **Bitmaps**
  - Einschalten von "Automatic Segment Space Management" bei Erzeugung eines Tablespace
  - Gilt für alle Segmente, die ab dann darin angelegt werden
- **Menge von Bitmap Blocks (BMBs)**
  - Organisiert als Baumstruktur
  - Wurzel im Segment Header, enthält Verweise auf alle inneren Knoten
  - Blattknoten mit Belegungsinformation für zusammenhängende Folge von Datenblöcken
  - Maximal drei Ebenen im Baum

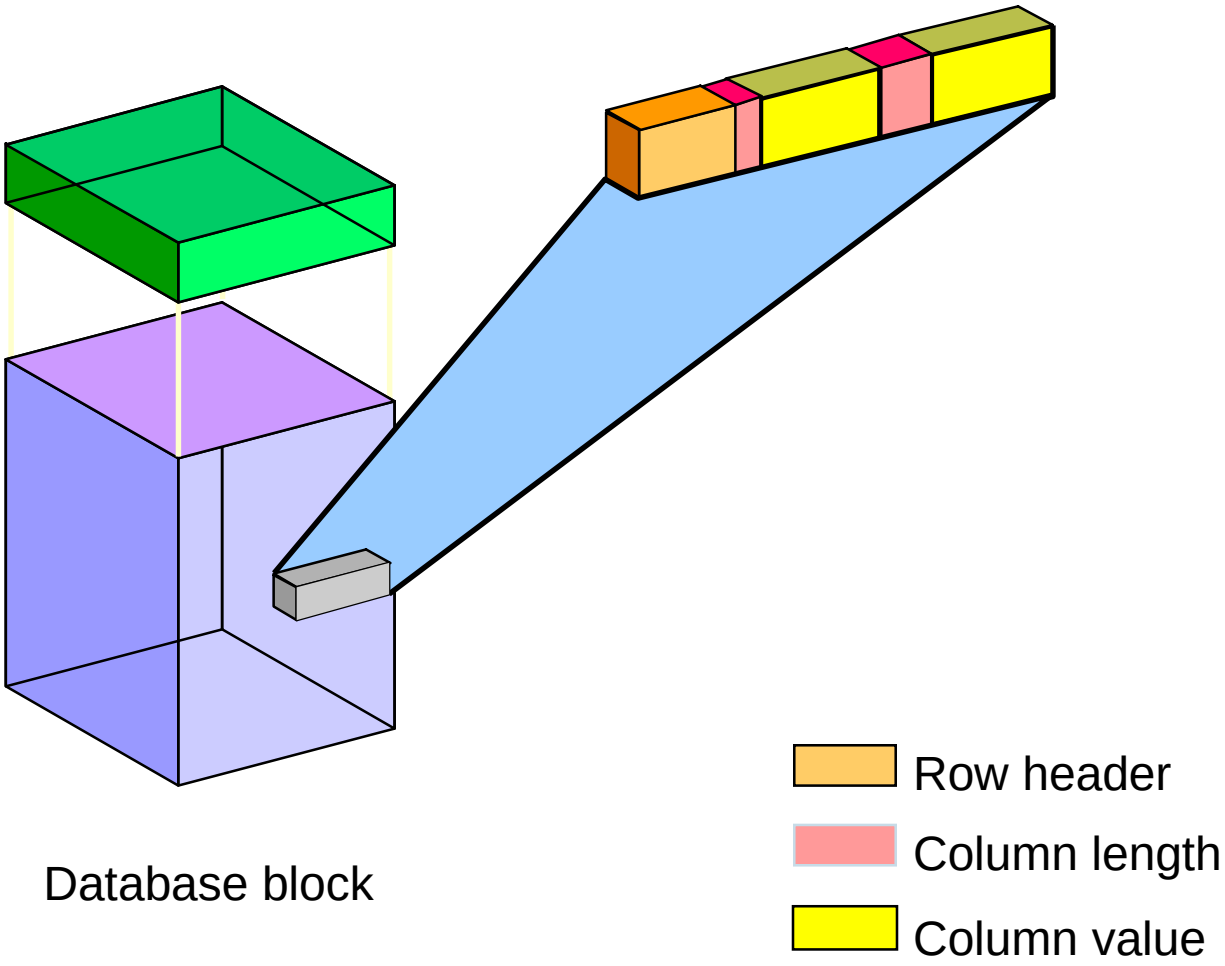
Database Block: Contents



Aufbau der Blöcke (Seiten)

- **Block header:**
  - Enthält Segment-Typ (z.B. Tabelle, Index), Datenblockadresse, Tabellenverzeichnis, Zeilenverzeichnis (Positionsindex) und Transaktionseinträge von jeweils ca. 23 Bytes, die bei Änderungen von Zeilen im Block benutzt werden
  - Wächst vom Anfang des Blocks her
    (durch neue Einträge im Zeilenverzeichnis und neue Transaktionseinträge)
- **Row data:**
  - Die Zeilen-Stücke: Sätze
  - Wächst vom Ende des Blocks her
- **Freier Speicherplatz:**
  - In der Mitte des Blocks
  - Am Anfang noch zusammenhängend, kann aber fragmentiert werden
  - Wird ggf. durch den Oracle-Server defragmentiert

Structure of a Row

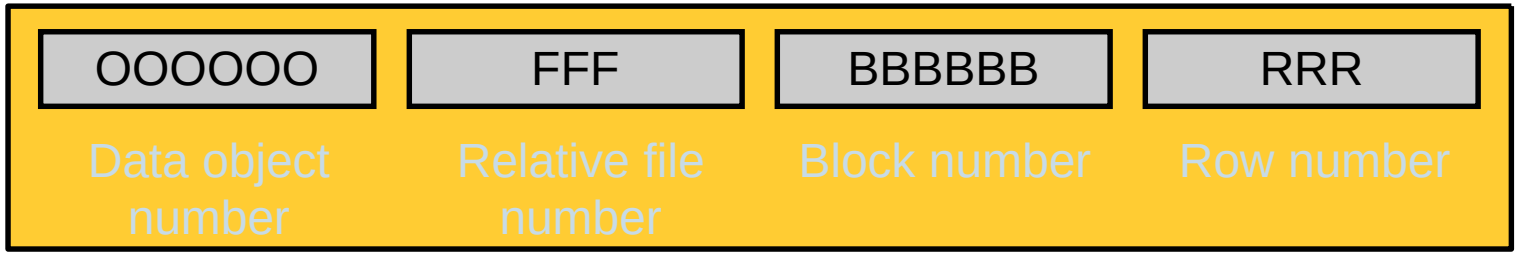


Struktur eines Satzes

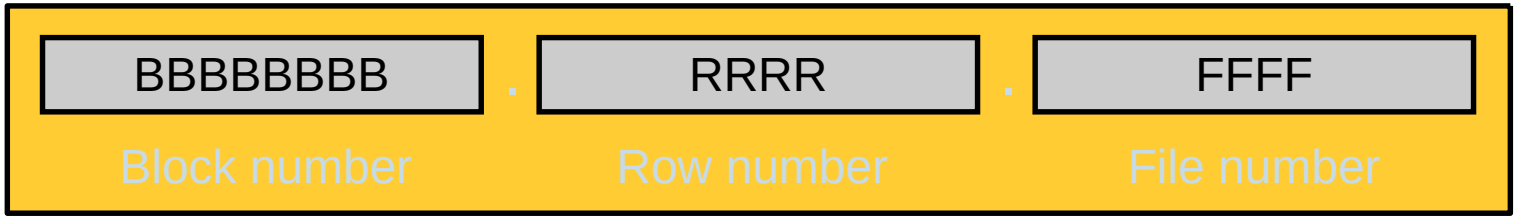
- **Variable Länge**
- **Spalten (Attribute) in der Reihenfolge ihrer Definition**
  - NULL-Werte
    - werden nur am Ende weggelassen
    - Ansonsten durch ein Byte (Längenfeld) dargestellt
- **Header:**
  - Anzahl der Spalten, Verkettungsinformation, Sperren
- **Daten:**
  - Für jede Spalte ein Längenfeld von einem Byte;
    bei sehr großen Werten (mehr als 250 Bytes) drei Bytes
  - Wert unmittelbar dahinter abgelegt
  - Kein Abstand erforderlich zwischen Sätzen
  - Jeder Satz ist im Zeilenverzeichnis (Positionsindex) eingetragen
    mit Zeiger auf den Anfang des Satzes.

ROWID Format

- Extended `ROWID` Format



- Restricted `ROWID` Format



ROWID

- **Analog zum TID**
- **Extended ROWID**
  - Braucht 10 Bytes Speicher und wird mit 18 Zeichen lesbar angezeigt
  - **Data object number:**
    - Wird jedem Datenobjekt (Tabelle, Index, …) bei seiner Erzeugung zugeordnet; eindeutig in der Datenbank (32 bits, 6 Zeichen)
  - **Relative file number:**
    - Eindeutig für jede Datei in einem Tablespace (10 bits, 3 Zeichen)
  - **Block number:**
    - Laufende Nummer des Blocks in der Datei (22 bits, 6 Zeichen)
  - **Row number:**
    - Laufende Nummer des Eintrags im Zeilenverzeichnis im Block-Header (16 bits, 3 Zeichen)
  - Lesbare Anzeige mit den 64 möglichen Zeichen A-Z, a-z und 0-9

ROWID (2)

- **Restricted ROWID**
  - Oracle7 und früher
  - Nur 6 Bytes Speicher
  - Dateinummern waren noch eindeutig in einer Datenbank.
    - Daher nicht mehr als 1.022 Dateien in einer Datenbank
    - Jetzt: in einem Tablespace
  - Immer noch verwendet in nicht-partitionierten Indexen und nicht-partitionierten Tabellen, weil alle Index-Einträge auf Zeilen (Sätze) im selben Segment verweisen.
- **Lokalisierung einer Zeile mit der ROWID:**
  - Segment liegt immer in genau einem Tablespace.
  - Also kann mit der Datenobjekt-Nummer der Tablespace identifiziert werden, in dem die Zeile (der Satz) gespeichert ist.

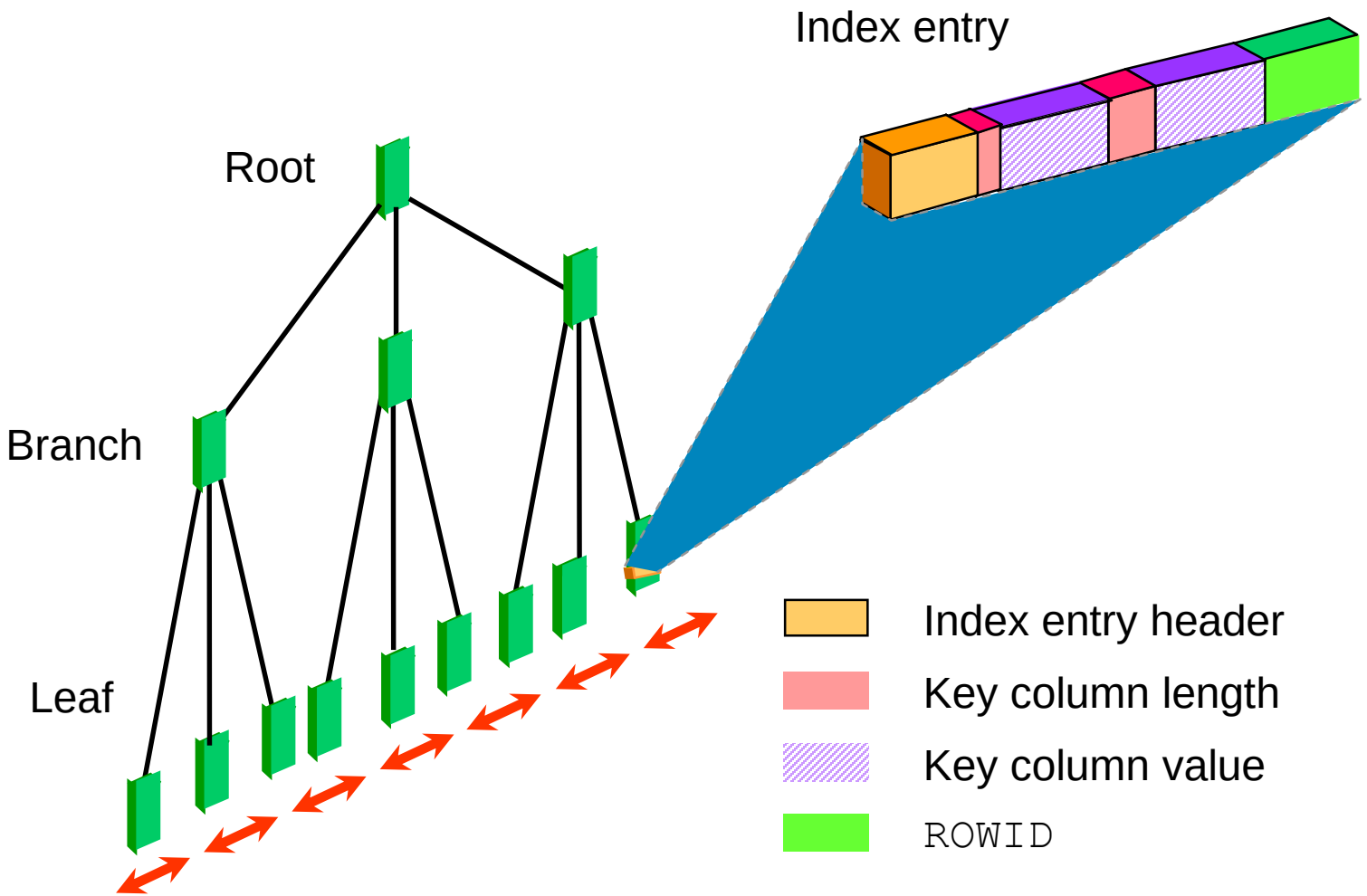
Indexstrukturen

- **Baumstruktur**
  - Kein Hashing, aber B-Baum und Bitmap
- **Single-column Index:**
  - Nur eine Spalte (ein Attribut)
- **Concatenated Index:**
  - Auch: composite index
  - Mehrere Spalten zusammen
    - Reihenfolge wählbar
    - Müssen auch nicht direkt hintereinander definiert worden sein
  - Maximal 32 Spalten
  - Gesamtgröße darf Hälfte der Blockgröße nicht überschreiten.

Indexstrukturen (2)

- **Unique and Nonunique Indexes:**
  - Klar
- **Function-based Indexes:**
  - Index-Einträge sind Werte einer Funktion (eines Ausdrucks), die auf eine Spalte oder mehrere Spalten einer Zeile angewendet wird.
  - Siehe Übung!
- **Domain Indexes:**
  - Für spezielle Datentypen (Text, Vektorgraphik)
  - Erzeugt, verwaltet und benutzt von einem eigenen Index-Typ
  - Nur single-column erlaubt
  - Spalten sind atomar, Objekte oder Large Objects (LOBs)
- **Partitioned and Nonpartitioned Indexes:**
  - Partitionierung über mehrere Segmente oder sogar Tablespaces
  - Erhöht Nebenläufigkeit (Sperrgranulat) und Wartbarkeit
  - Oft verwendet zusammen mit partitionierten Tabellen;
    dann je eine Index-Partition pro Tabelle-Partition.

B-Tree Index


Index entry
Root
Branch
Leaf
Index entry header
Key column length
Key column value
ROWID

B-Baum

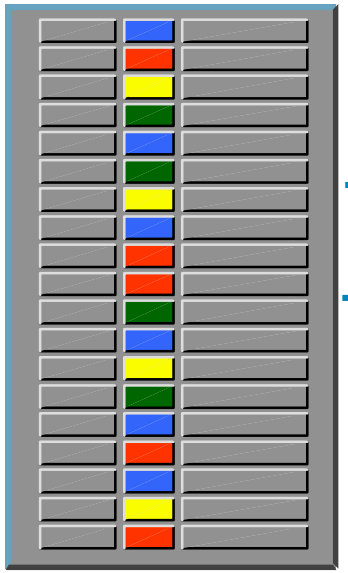
- **Liste von ROWIDs für jeden Schlüsselwert**
  - Also nur als **Sekundärorganisation**
- **Struktur:**
  - Index-Einträge nur in den Blattknoten, also **B*-Baum**
  - Doppelte Verkettung der Blattknoten wie gehabt
- **Index-Einträge:**
  - Header: Zahl der Spalten und Sperrinformation
  - Für jede Spalte das Paar von Länge und Wert
  - ROWID

B-Baum (2)

- **Blattknoten:**
  - Schlüsselwerte werden wiederholt, wenn sie in mehreren Zeilen vorkommen, falls der Index nicht komprimiert wird.
  - Kein Index-Eintrag, wenn alle Schlüsselwerte NULL sind
    - Suche mit IS NULL in der WHERE-Klausel benutzt den Index also nicht.
  - Verwendung der Restricted ROWID
- **Wirkung von Änderungsoperationen:**
  - Einfügen einer Zeile erzeugt einen neuen Index-Eintrag.
  - Löschen einer Zeile markiert nur den Index-Eintrag als gelöscht.
  - Änderungen werden als Löschen und Einfügen realisiert.

Bitmap Indexes

Table



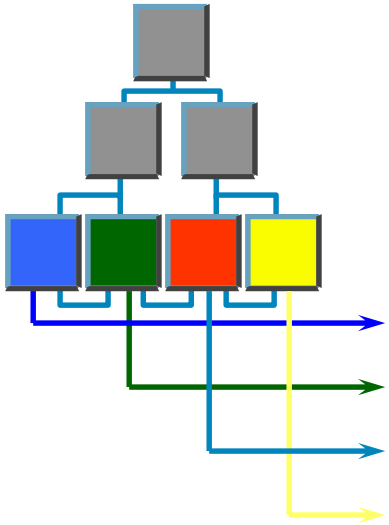
File 3

Block 10

Block 11

Block 12

Index



key	start ROWID	end ROWID	bitmap
<Blue,	10.0.3,	12.8.3,	1000100100010010100>
<Green,	10.0.3,	12.8.3,	0001010000100100000>
<Red,	10.0.3,	12.8.3,	0100000011000001001>
<Yellow,	10.0.3,	12.8.3,	0010001000001000010>

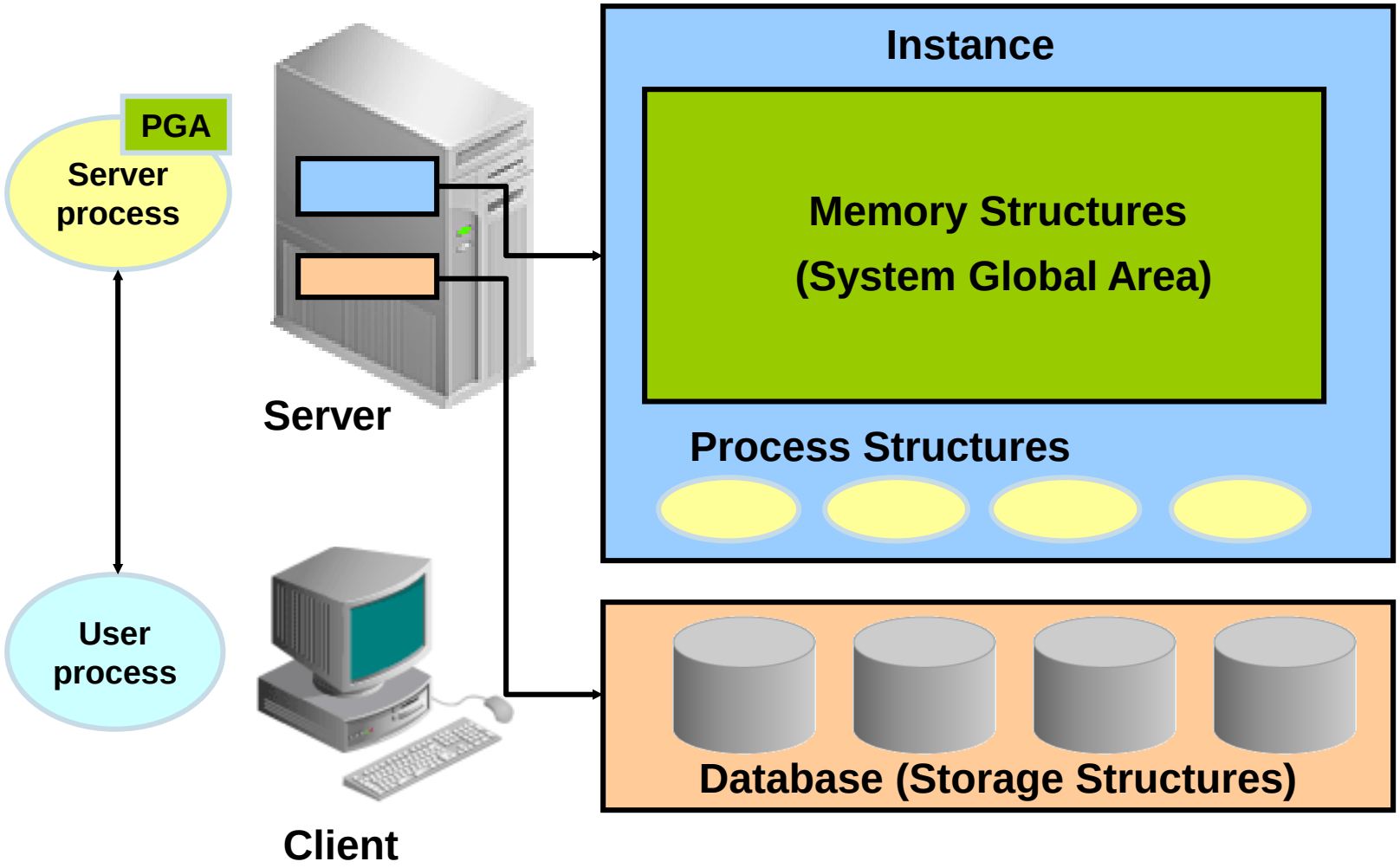
Bitmap-Index

- **Vorteile (wie bekannt):**
  - Bei vielen Zeilen und wenigen Werten in den Schlüsselspalten (Geschlecht, Familienstand)
  - Bei Anfragen mit mehreren Bedingungen in den WHERE-Klausel, die mit OR verknüpft sind
  - Bei vielen lesenden und wenigen schreibenden Zugriffen auf die Schlüsselspalten
- **Struktur:**
  - Ebenfalls als B*-Baum organisiert,
    aber mit einer Bitmap in den Index-Einträgen an Stelle der ROWIDs
  - Jedes Bit gehört zu einer bestimmten ROWID.

Bitmap-Index (2)

- **Blattknoten (= ein Index-Eintrag):**
  - Header: Zahl der Spalten und Sperrinformation
  - Für jede Spalte das Paar von Länge und Wert
  - Start ROWID: erste Zeile der Tabelle, zu der das erste Bit im Bitmap gehört
  - End ROWID: letzte Zeile der Tabelle, zu der das letzte Bit im Bitmap gehört
  - Bitmap komprimiert gespeichert
  - Verwendung der Restricted ROWID

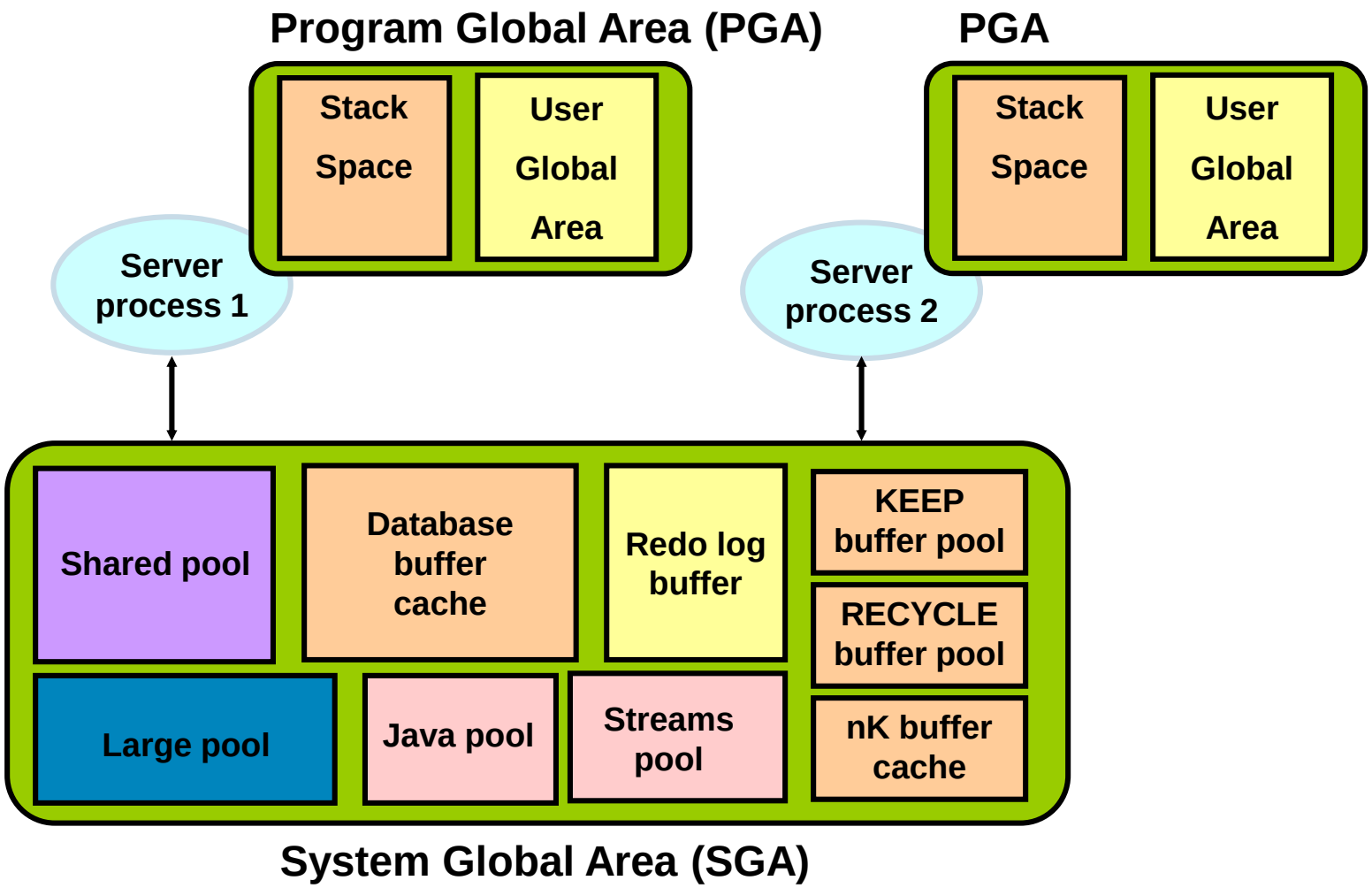
Oracle Database Server Architecture: Overview



Oracle-Datenbank-Instanzen

- **Oracle DBS besteht aus:**
  - Oracle-Datenbank und
  - DB-Instanz
- **DB-Instanz**
  - Datenstrukturen und Prozesse, die damit assoziiert sind
  - System Global Area (SGA)
  - Hintergrundprozesse
- **Datenbank assoziieren: Mounting**
- **Anmerkung:**
  - Oracle Automatic Storage Management (ASM)
    verwendet ebenfalls das Konzept einer Instanz mit Datenstrukturen und Prozessen, ist aber nicht mit einer bestimmten Datenbank assoziiert.

Oracle Database Memory Structures



Datenstrukturen

- **Im Hauptspeicher ("Memory")**
  - Code, gemeinsame Daten, Benutzer-spezifische Daten
- **System Global Area (SGA):**
  - Gemeinsam für eine DB-Instanz
  - Zugriff durch alle Server- und Hintergrund-Prozesse
  - Beispiele: Blöcke im Puffer, gemeinsame SQL-Bereiche
- **Program Global Areas (PGA):**
  - Getrennt für jeden einzelnen Server- oder Hintergrund-Prozess

System Global Area (SGA)

- **Shared pool:**
  - Diverse Dinge für alle Benutzer
- **Database buffer cache:**
  - Datenbank-Puffer
- **KEEP buffer pool:**
  - Spezieller Puffer für Blöcke, die länger im Hauptspeicher bleiben sollen
- **RECYCLE buffer pool:**
  - Spezieller Puffer, um schnell Blöcke wiederverwenden oder aus dem Puffer entfernen zu können
- **nK buffer cache:**
  - Spezieller Puffer für Blöcke mit abweichender Größe
- **Redo log buffer:**
  - Zwischenspeicher für Redo-Information, bis sie auf die Log-Datei auf Platte geschrieben werden kann

SGA (2)

- **Large pool:**
  - Optionaler Bereich, der größeren Speicher für bestimmte Prozesse bereitstellt, etwa für Backup und Recovery sowie für I/O-Server-Prozesse
- **Java pool:**
  - Für sämtlichen Session-spezifischen Java-Code und die Daten der JVM
- **Streams pool:**
  - Für Oracle Streams zum Speichern von Information, die von "capture" und "apply" benötigt wird

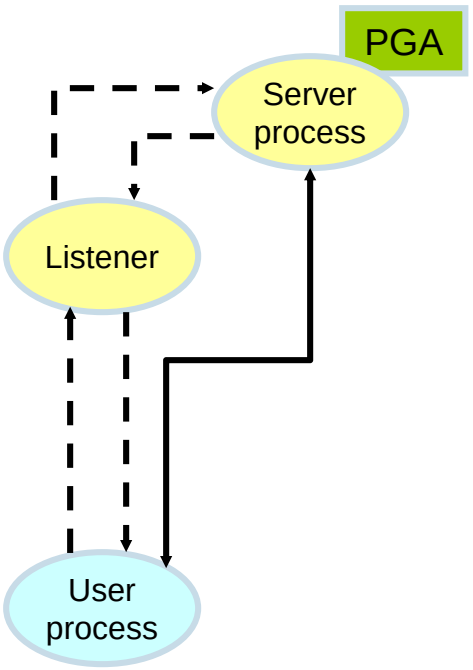
- **Dynamic SGA Infrastructure**
  - Die Größe von Database buffer cache, Shared pool, Large pool, Java pool und the Streams pool kann sich ändern, ohne dass die Instanz neu gestartet warden muss.

Program Global Area (PGA)

- **Für jeden Server-Prozess getrennt**
  - Bedient einen Client
  - Zwei Teile: Stack Space und User Global Area (UGA)

- **Initialisierungsparameter**
  - Für die Erzeugung und Verwaltung der Datenstrukturen
  - Oracle kann sie auch selbst setzen.
    - Dann nur noch Zielgröße (MEMORY_TARGET) und Maximalgröße (MEMORY_MAX_TARGET) für den Speicher setzen.

Process Structures



Instances (ASM and Database separate)

System Global Area (SGA)

Background processes

Required: DBW*n* CKPT LGWR SMON PMON RECO

Optional: ARC*n* ASMB RBAL Others

Grid Infrastructure Processes
(ASM and Oracle Restart)

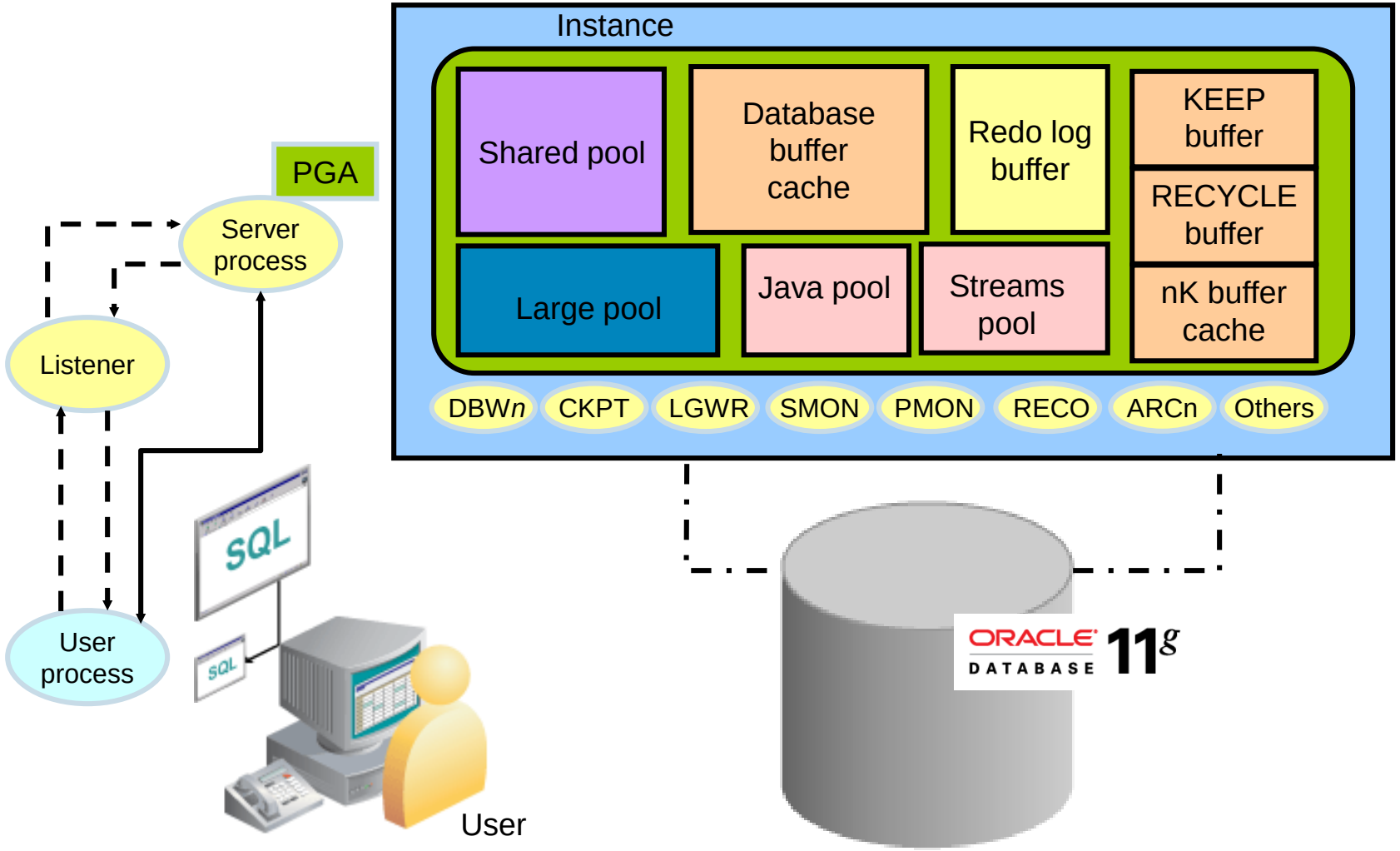
ohas ocssd diskmon

orarootagent oraagent cssdagent

Prozess-Strukturen

- **Server-Prozesse**
  - Bearbeiten Anfragen der Benutzerprozesse
  - Diese kommunizieren erst mit dem **Listener**,
    der einen Server-Prozess in einer definierten Umgebung erzeugt.
  - Parsen SQL-Anweisungen und führen sie aus
  - Holen benötigte Datenblöcke aus den Dateien in den Datenbank-Puffer in der SGA (falls sie nicht schon in der SGA vorhanden waren)
  - Liefern die Ergebnisse in passender Form an die Benutzerprozesse zurück
- **Hintergrund-Prozesse**
  - Performance verbessern, viele Benutzer bedienen

Oracle Database Server Architecture: Overview



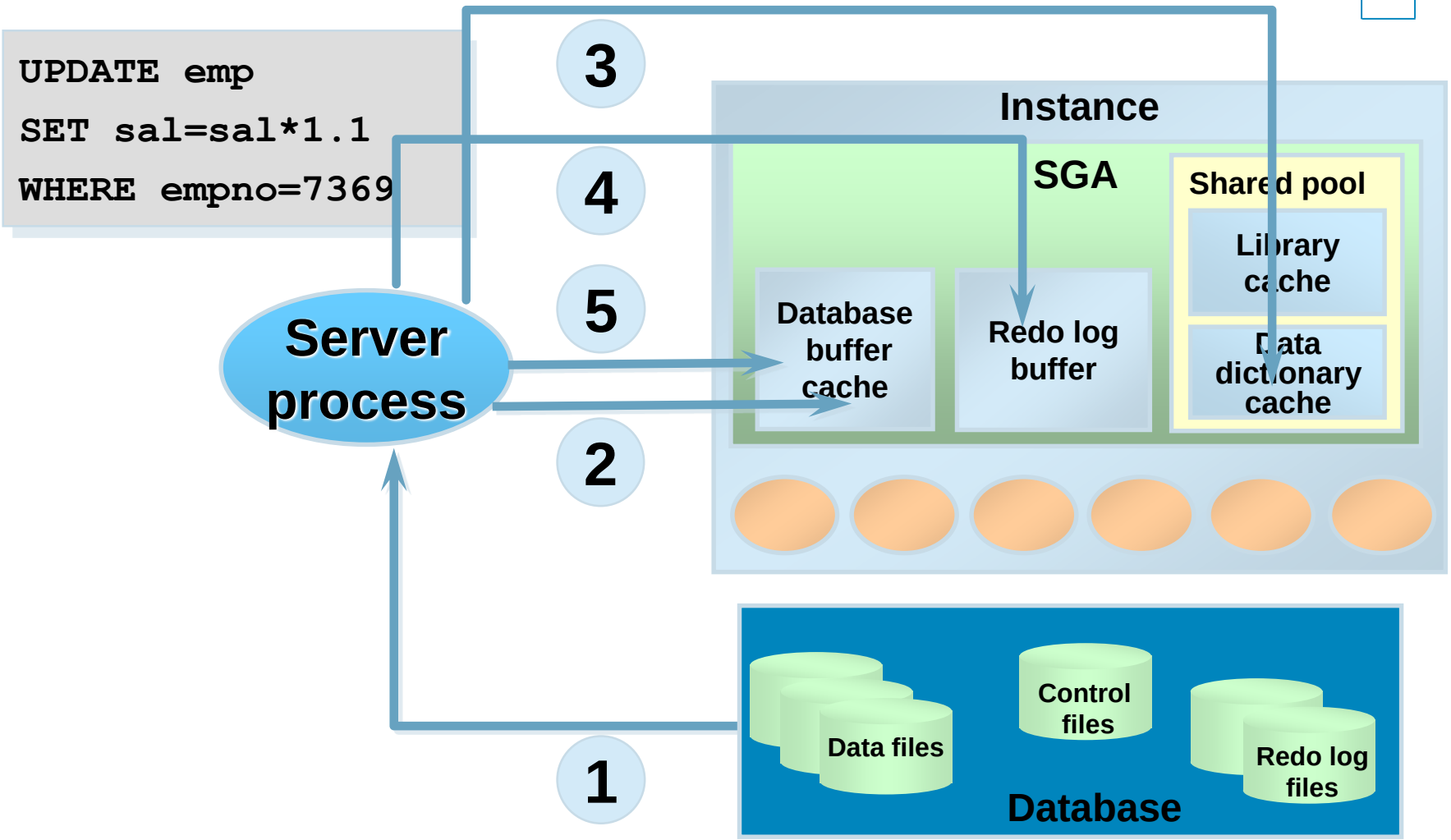
Interaktion mit einer Oracle-Datenbank

1. Eine DB-Instanz läuft auf einem Knoten;
   oft als *Host* oder *Datenbank-Server* bezeichnet.
2. Ein Benutzer startet eine Anwendung in einem Benutzer-Prozess.
   Diese Anwendung nimmt Verbindung zum Server auf.
   Die Verbindung kann lokal, Client-Server oder auch Middleware sein.
3. Server hat einen **Listener**, der über den erforderlichen Oracle-Net-Services-Handler verfügt. Listener erkennt die Verbindungsaufforderung der Anwendung und erzeugt einen zugeordneten Server-Prozess.
4. Benutzer führt SQL-Anweisungen aus und beendet die Transaktion (commit).
5. Server-Prozess nimmt die Anweisungen entgegen und prüft im Shared Pool, ob die **Shared SQL Area** eine identische Anweisung enthält. Falls ja, prüft der Server-Prozess die Zugriffsberechtigungen und führt dann die Anweisung mit den Inhalten der Shared SQL Area aus. Andernfalls wird eine neue Shared SQL Area angelegt, mit der die Anweisung dann geparst und ausgeführt wird.

Interaktion mit einer Oracle-Datenbank (2)

6. Server-Prozess holt die erforderlichen Daten, entweder aus Dateien oder aus dem Datenbank-Puffer.
7. Server-Prozess modifiziert die Daten in der SGA. Wenn die Transaktion erfolgreich abgeschlossen wird, schreibt der **Log-Writer-Prozess** (LGWR) die Daten unmittelbar in die Redo-Log-Datei. Der **Database-Writer-Prozess** (DBW*n*) schreibt die geänderten Blöcke erst dann auf die Platte, wenn es effizient möglich ist (NoFORCE).
8. Bei erfolgreichen Transaktionen sendet der Server-Prozess eine Nachricht über das Netz an die Anwendung. Bei Misserfolg wird eine Fehlermeldung gesendet.
9. Während dieser ganzen Prozedur laufen auch die anderen Hintergrund-Prozesse weiter und achten auf Bedingungen, die ein Eingreifen erfordern könnten. Außerdem führt der Datenbank-Server auch noch andere Transaktionen aus und verhindert Konflikte, wenn Transaktionen auf dieselben Daten zugreifen.

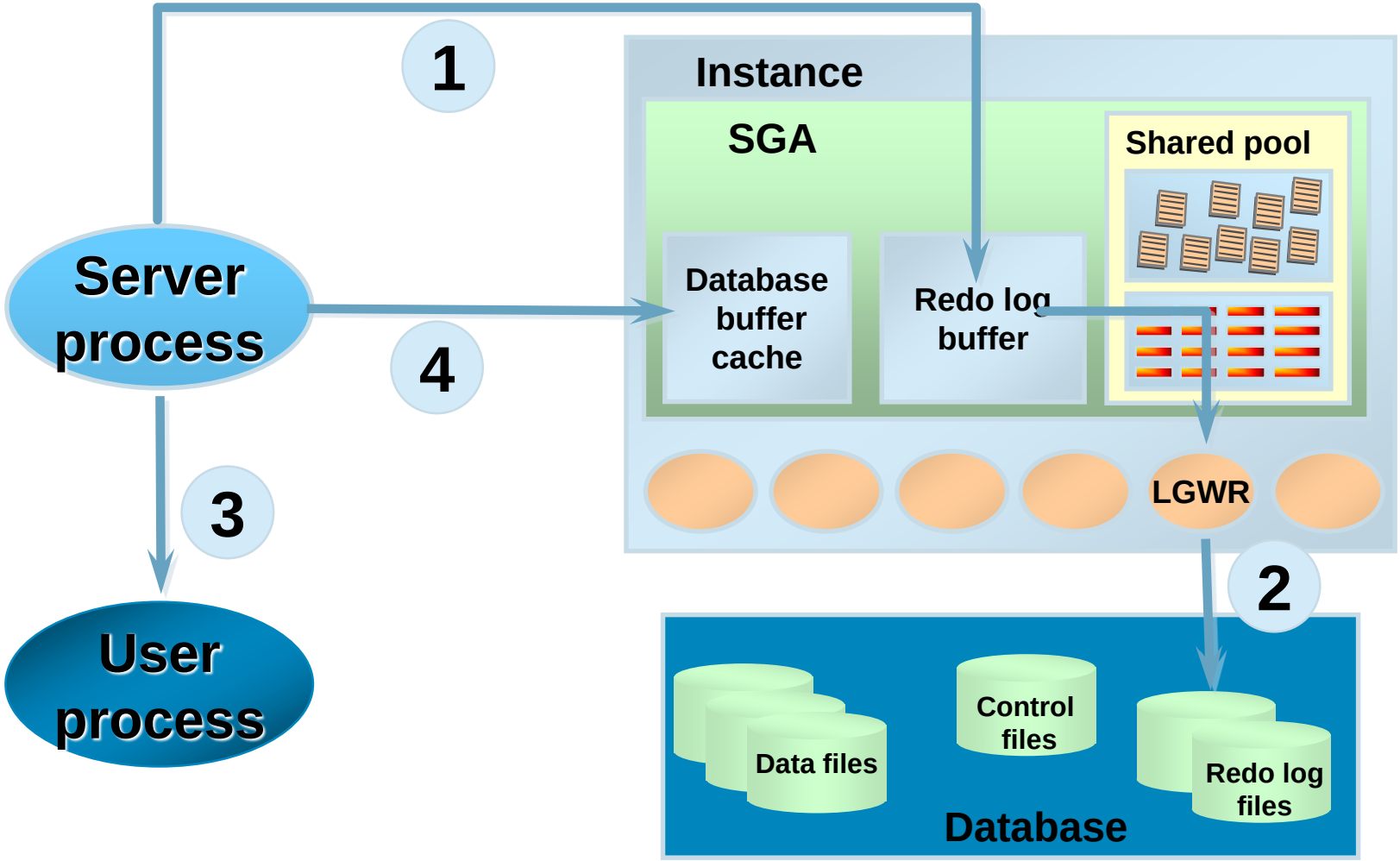
Processing a DML Statement



Verarbeitung einer Änderungsanweisung

- **Schritte:**
  1. Der Server-Prozess liest die Daten- und Rollback-Blöcke aus der Datei, falls sie nicht bereits im Puffer sind.
  2. Kopien der gelesenen Blöcke werden im Puffer abgelegt.
  3. Der Server-Prozess setzt Sperren auf den Daten.
  4. Der Server-Prozess protokolliert die durchzuführenden Änderungen für Rollback (Before-Image) und für Wiederherstellung (After-Image) in den Redo-Log-Puffer.
  5. Der Server-Prozess schreibt das Before-Image in den Rollback-Block und ändert den Daten-Block, beides im Datenbank-Puffer. Die beiden geänderten Blöcke werden als "dirty" markiert, d.h. sie haben nicht mehr den gleichen Inhalt wie die entsprechenden Blöcke auf der Platte.
- **Anmerkung:**
  - Die Verarbeitung von DELETE oder INSERT erfolgt in ähnlichen Schritten. Das Before-Image für ein DELETE enthält die Attributwerte der gelöschten Zeile. INSERTs benötigen nur die Identifikation der Zeile als Information im Rollback.

COMMIT Processing



COMMIT-Verarbeitung

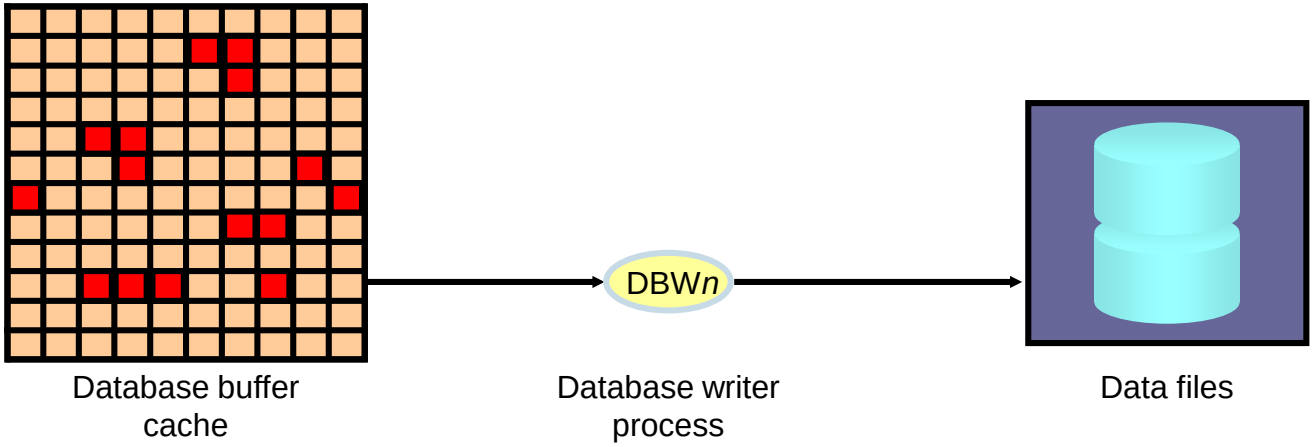
- **System Change Number (SCN)**
  - Immer wenn eine Transaktion erfolgreich endet, weist Oracle eine SCN zu. Diese sind monoton aufsteigend und eindeutig innerhalb der Datenbank. Die SCN wird von Oracle als interner Zeitstempel benutzt, um Datenzugriffe zu synchronisieren und um Lesekonsistenz zu gewährleisten. Oracle macht damit Konsistenzprüfungen, ohne sich auf Datum und Uhrzeit des Betriebssystems verlassen zu müssen.
- **Schritte der COMMIT-Verarbeitung:**
  - Der Server-Prozess schreibt einen COMMIT-Satz zusammen mit der SCN in den Redo-Log-Puffer.
  - LGWR schreibt zusammenhängend alle Einträge des Redo-Log-Puffers bis zum diesem COMMIT-Satz (einschließlich) in die Redo-Log-Datei. Danach kann Oracle garantieren, das die Änderungen auch im Fehlerfall nicht mehr verlorengehen.
  - Der Benutzer wird informiert, dass das COMMIT vollständig abgeschlossen ist.
  - Der Server-Prozess schreibt Informationen, die anzeigen, dass die Transaktion vollständig ist und dass die Sperren freigegeben werden können. Das Schreiben der geänderten Pufferinhalte in die Dateien kann davon unabhängig vom DBWR vorgenommen werden (NoFORCE). Es kann sowohl vor als auch nach dem COMMIT erfolgen.

COMMIT-Verarbeitung (2)

- **Anmerkung:**
  - Das Zurücksetzen einer Transaktion veranlasst den LGWR nicht, auf Platte zu schreiben. Oracle setzt unvollständige Transaktionen immer zurück, wenn es Fehler behandelt. Falls ein Fehler nach einem Rollback auftritt und noch bevor die Rollback-Einträge (als durchgeführt) auf die Platte geschrieben werden können, genügt das Fehlen des COMMIT-Satzes, um sicherzustellen, dass die Änderungen der Transaktion zurückgesetzt werden.
- **Vorteile dieses Fast COMMIT (= NoFORCE):**
  - Sequenzielles Schreiben in die Protokoll-Dateien ist schneller als das verstreute Schreiben von Daten-Blöcken.
  - Nur minimale Information, die unbedingt erforderlich ist, um Änderungen zu protokollieren, wird in die Protokoll-Dateien geschrieben, während beim Schreiben in die Datenbank immer ganze Blöcke geschrieben werden müssen.
  - Redo-Log-Sätze mehrerer Transaktionen, die zur gleichen Zeit abgeschlossen werden, können in einer einzigen Schreiboperation gesichert werden (sog. "Group-Commit").
  - Falls der Redo-Log-Puffer nicht zu voll ist, wird nur eine einzige synchrone Schreiboperation pro Transaktion benötigt (oder sogar weniger als eine).
  - Der Umfang einer Transaktion beeinflusst die Zeit, die für ein COMMIT benötigt wird, nicht.

Database Writer Process (DBWn)

- **Writes modified (dirty) buffers in the database buffer cache to disk:**
  - Asynchronously while performing other processing
  - To advance the checkpoint



Database-Writer-Prozess (DBW*n*)

- **Schreibt den Inhalt von Pufferseiten in die Dateien**
  - Modifizierte ("dirty") Seiten
  - Einer (DBW*0*) genügt für die meisten Systeme,
    aber man kann weitere einrichten (DBW*1* bis DBW*9* und DBW*a* bis DBW*z*), um die Schreibgeschwindigkeit zu erhöhen, wenn die Daten oft geändert werden. Allerdings zahlt sich das nur bei Mehrprozessorsystemen aus.
  - Wird eine Seite im Puffer geändert, wird sie als "dirty" markiert und in die Sicherungspunkt-Warteschlange eingetragen. Diese Warteschlange wird in SCN-Reihenfolge organisiert. Das entspricht der Reihenfolge der Redos, die in den Redo-Log geschrieben werden.
  - Fällt die Zahl der freien Pufferseiten unter einen internen Schwellenwert, schreibt DBW*n* nicht so häufig genutzte Seiten in die Dateien zurück, beginnend mit dem Ende der LRU-Kette.
  - DBW*n* schreibt außerdem vom Ende der Sicherungspunkt-Warteschlange her, damit die Sicherungspunkte neu gesetzt werden können.

DBW*n* (2)

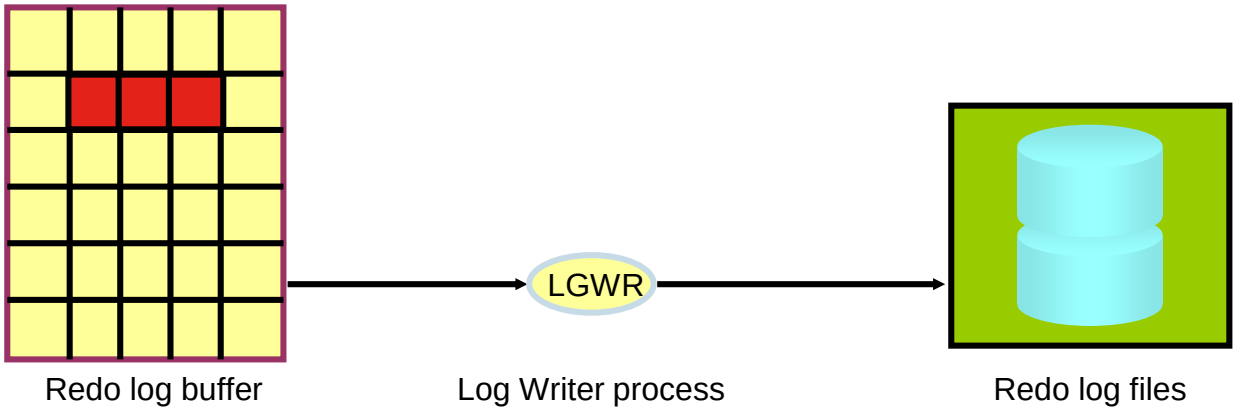
- **Redo Byte Address (RBA)**
  - Steht in der SGA
  - Zeiger auf Position im Redo-Strom, an der die Recovery beginnen soll.
  - Wird alle drei Sekunden vom CKPT-Prozess in die Steuerdatei geschrieben.
  - Da DBW*n* geänderte Seiten in SCN-Reihenfolge schreibt und der Redo-Log ebenfalls in SCN-Reihenfolge abgelegt wird, kann DBW*n* den Zeiger immer dann weiterrücken, wenn er geänderte Seiten aus der LRU-Liste zurückschreibt. Deshalb beginnt die Recovery das Redo an der annähernd korrekten Position und vermeidet unnötige E/A. Man nennt das auch Inkrementelles Checkpointing.
- **Anmerkung:**
  - Es gibt weitere Anlässe für den DBW*n* zu schreiben, z.B. wenn ein Tablespace auf "read-only" oder "offline" gesetzt wird. Dann wird kein inkrementeller Sicherungspunkt gesetzt, weil alle geänderten Seiten der zugehörigen Dateien einfach in die Datenbank geschrieben werden, ohne auf die SCN-Reihenfolge zu achten.

DBW*n* (3)

- **LRU-Algorithmus**
  - Hält die häufig verwendeten Blöcke im Puffer (wie gehabt).
  - CACHE-Option kann für bestimmte Tabellen gesetzt werden, damit ihre Seiten länger im Puffer bleiben.
- **Initialisierungsparameter DB_WRITER_PROCESSES**
  - Legt die Zahl der DBW*n*-Prozesse fest.
  - Maximum ist 36.
  - Wird er beim Hochfahren nicht angegeben, bestimmt Oracle den Wert auf der Basis der Prozessorzahl und -gruppen.
- **Bedingungen für das Schreiben geänderter Pufferseiten durch DBW*n*:**
  - Findet ein Server-Prozess nach dem Scannen einer gegebenen Zahl von Pufferseiten keine freie, stößt er DBW*n* zum Schreiben an. DBW*n* schreibt asynchron.
  - DBW*n* schreibt, um den Sicherungspunkt weiterzurücken. Er entspricht der ältesten geänderten Seite im Puffer.
- **DBW*n* schreibt immer mehrere Blöcke auf einmal (batch, multiblock).**
  - Die Zahl variiert mit dem Betriebssystem.

Log Writer Process (LGWR)

- Writes the redo log buffer to a redo log file on disk
- Writes:
  - When a user process commits a transaction
  - When the redo log buffer is one-third full
  - Before a DBWn process writes modified buffers to disk
  - Every 3 seconds

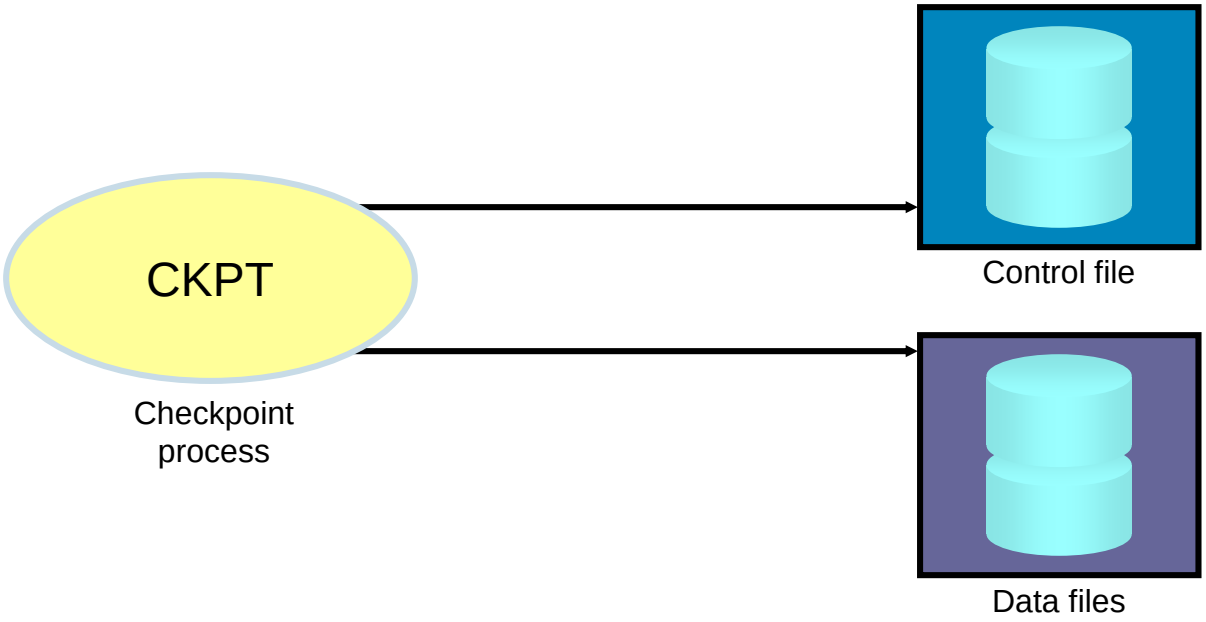


Log-Writer-Prozess (LGWR)

- **Verwaltet den Redo-Log-Puffer**
  - Schreibt dessen Einträge in die Redo-Log-Dateien auf der Platte.
  - Immer alle, die seit dem letzten Schreiben eingetragen wurden.
- **Zyklischer Puffer**
  - Während LGWR in die Protokolldatei schreibt, können Server-Prozesse neue Einträge schon dort ablegen, wo bereits ausgeschrieben wurde.
  - LGWR schreibt normalerweise schnell genug, um sicherzustellen, dass immer genug Platz ist für neue Einträge, selbst wenn viele Zugriffe auf den Redo-Log erfolgen.
  - LGWR schreibt einen zusammenhängenden Abschnitt des Puffers auf die Platte.
- **Anlässe:**
  - Benutzerprozess beendet eine Transaktion mit COMMIT (WAL)
  - Redo-Log-Puffer ist zu einem Drittel voll
  - Bevor ein DBW*n*-Prozess geänderte Seiten auf die Platte schreibt (falls nötig)
  - Alle drei Sekunden

Checkpoint Process (CKPT)

- Records checkpoint information in
  - Control file
  - Each data file header

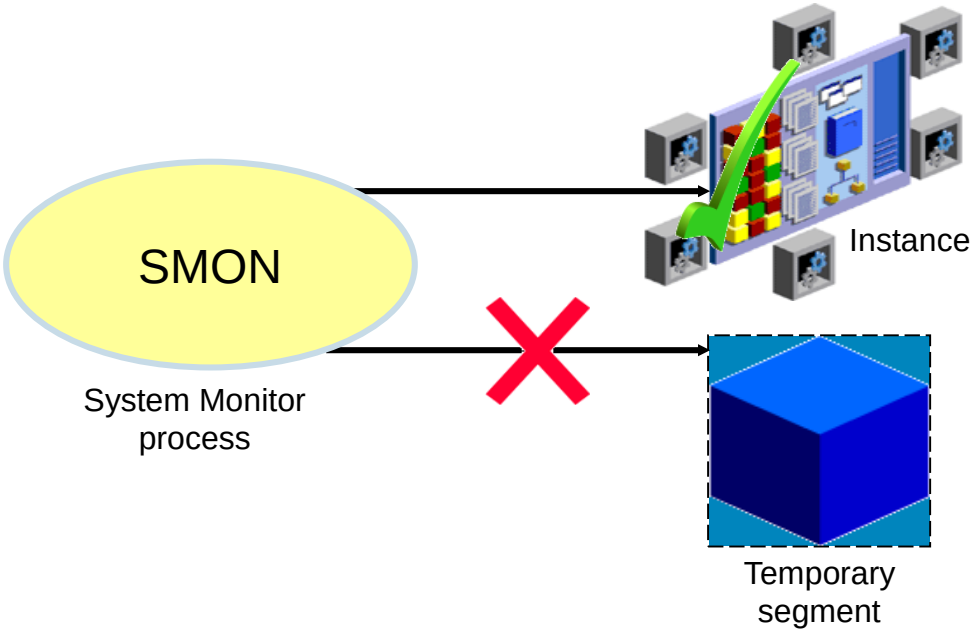


Checkpoint-Prozess (CKPT)

- **Sicherungspunkt**
  - Datenstruktur, die eine System Change Number (SCN) in der Redo-Abfolge einer Datenbank definiert.
  - Werden in der Steuerungsdatei (Control File) und im Header jeder Datei vermerkt.
  - Das macht der CKPT-Prozess.
  - Der schreibt selbst keine Seiten auf die Platte; das macht DBW*n*.
  - Die SCN im Datei-Header stellt sicher, dass alle Änderungen an der Datenbank, die vor dieser SCN durchgeführt wurden, bereits in der Datei auf Platte enthalten sind.

System Monitor Process (SMON)

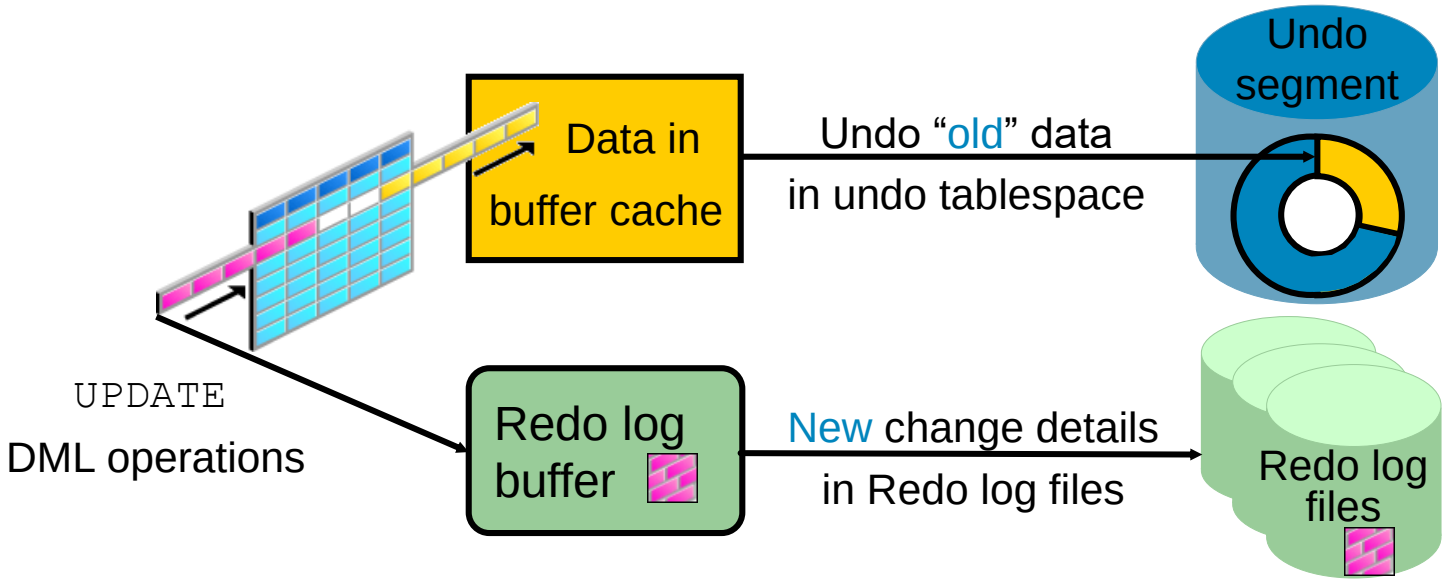
- Performs recovery at instance startup
- Cleans up unused temporary segments



System-Monitor-Prozess (SMON)

- **Recovery beim Hochfahren**
  - Falls erforderlich
  - Außerdem temporäre Segmente entfernen, die nicht mehr benötigt werden.
  - Falls abgeschlossene Transaktionen in der Recovery übergangen werden mussten, weil es Lesefehler bei Dateien gab oder Dateien nicht verfügbar waren, werden sie wiederhergestellt, wenn der Tablespace oder die Datei wieder geöffnet werden.
  - SMON prüft regelmäßig, ob diese Verarbeitung benötigt wird. Andere Prozesse dürfen SMON aufrufen, wenn sie es für nötig halten.

Transactions and Undo Data



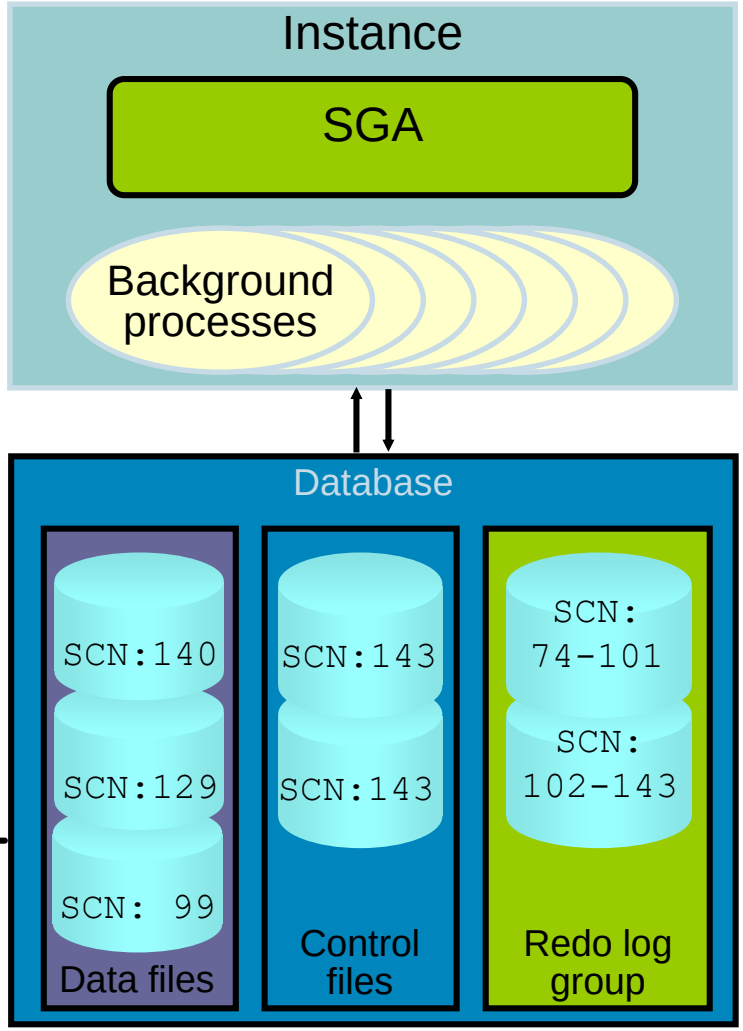
- Each transaction is assigned to only one undo segment.
- An undo segment can service more than one transaction at a time.

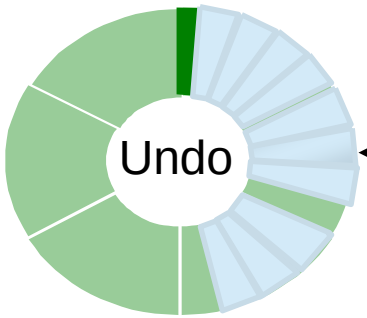
Transaktionen und Undo-Information

- **Undo-Segment**
  - Wird einer Transaktion zugeordnet, wenn sie beginnt.
    - Die Zuordnung kann im Dynamic Performance View `V$TRANSACTION` nachgesehen werden.
  - Besteht wie alle Segmente aus Extents, die ihrerseits aus Blöcken bestehen
  - Kann wachsen und schrumpfen je nach Bedarf
  - Verhält sich wie ein zyklischer Puffer
  - Während die Transaktion läuft, wird vor einer Änderung von Daten der Originalzustand in das Undo-Segment kopiert.
- **Trick:**
  - Undo-Segmente sind Segmente!
  - Änderungen an ihnen erzeugen ebenfalls Redo-Einträge.
  - D.h. Redo nach einem Ausfall stellt auch die Undo-Segmente wieder her.
  - Können danach für Rollback genutzt werden.

Phases of Instance Recovery

1. **Startup instance (data files are out of sync)**
2. **Roll forward (redo)**
3. **Committed and uncommitted data in files**
4. **Database opened**
5. **Roll back (undo)**
6. **Committed data in files**





Recovery-Phasen

- **Beim Öffnen von Dateien:**
  - SCN im Header mit SCN in der Steuerungsdatei vergleichen.
  - Bei Abweichung Redo-Daten anwenden.
  - Nachdem alle Dateien aktualisiert wurden, wird die Datenbank geöffnet und die Benutzer können sich wieder anmelden.
- **Anmerkung:**
  - Anwendung des Redo-Logs versetzt *alle* Transaktionen wieder in ihren Zustand zum Zeitpunkt des Fehlers – auch die noch nicht abgeschlossenen.
  - Nach dem Öffnen der Datenbank werden diese noch nicht abgeschlossenen Transaktionen zurückgesetzt (Undo).
  - Am Ende dieser Rollback-Phase enthalten die Dateien nur noch Daten von abgeschlossenen Transaktionen ("committed data").

Vorlesung Implementierung von Datenbanksystemen

15. Epilog – Zusammenfassung und Ausblick

Prof. Dr. Klaus Meyer-Wegener

Wintersemester 2019/20

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

15.1 Zusammenfassung der Vorlesung

- **Datenbanksystem**
  - Zentrale IT-System-Komponente in fast jedem Unternehmen
  - Systemtechnische Grundlage des eCommerce / eBusiness / e???
- **Architektur von Datenbanksystemen**
  - Strukturierung (Schichtenmodell, Komponentensichtweise)
  - Schwerpunkt auf generischer Softwareentwicklung (Anwendungsneutralität)
  - Definition / Erfüllung von Schnittstellen
  - Wechselwirkungen beteiligter Komponenten
  - Speicher-, Zugriffs- und Verarbeitungsschicht
- **Transaktionsverarbeitung**
  - Synchronisation gleichzeitiger Zugriffe auf gemeinsame Daten
  - Protokollierung und Wiederherstellung

Schichtenmodell – unterer Teil

interne Satzschnittstelle

r := READ <Satzadr>;
<Satzadr> := INSERT r;

Speicherungsstrukturen

Adressierungseinheiten:	Sätze, Indexe
Hilfsstrukturen:	Positionsindex, Freisp., Hash-Buckets usw.
Adressierungseinheiten:	Seiten, Segmente

Satzverwaltung, Zugriffspfadverwaltung

seitenorientierte Pufferschnittstelle

BEREITSTELLEN Segment, Seite;
FREIGEBEN Seite;

Seitenzuordnungsstrukturen

Adressierungseinheiten:	Seiten, Segmente
Hilfsstrukturen:	Seitentabelle, Blocktabellen
Adressierungseinheiten:	Blöcke, Dateien

Pufferverwaltung, Einbringstrategien

blockorientierte Dateischnittstelle

LIES Datei, Block, k;
SCHREIB k, Datei, Block;

Speicherzuordnungsstrukturen

Adressierungseinheiten:	Blöcke, Dateien
Hilfsstrukturen:	Dateikatalog, Freisp., Extent-Tabellen
Adressierungseinheiten:	Slots, Spuren, Zylinder, Geräte usw.

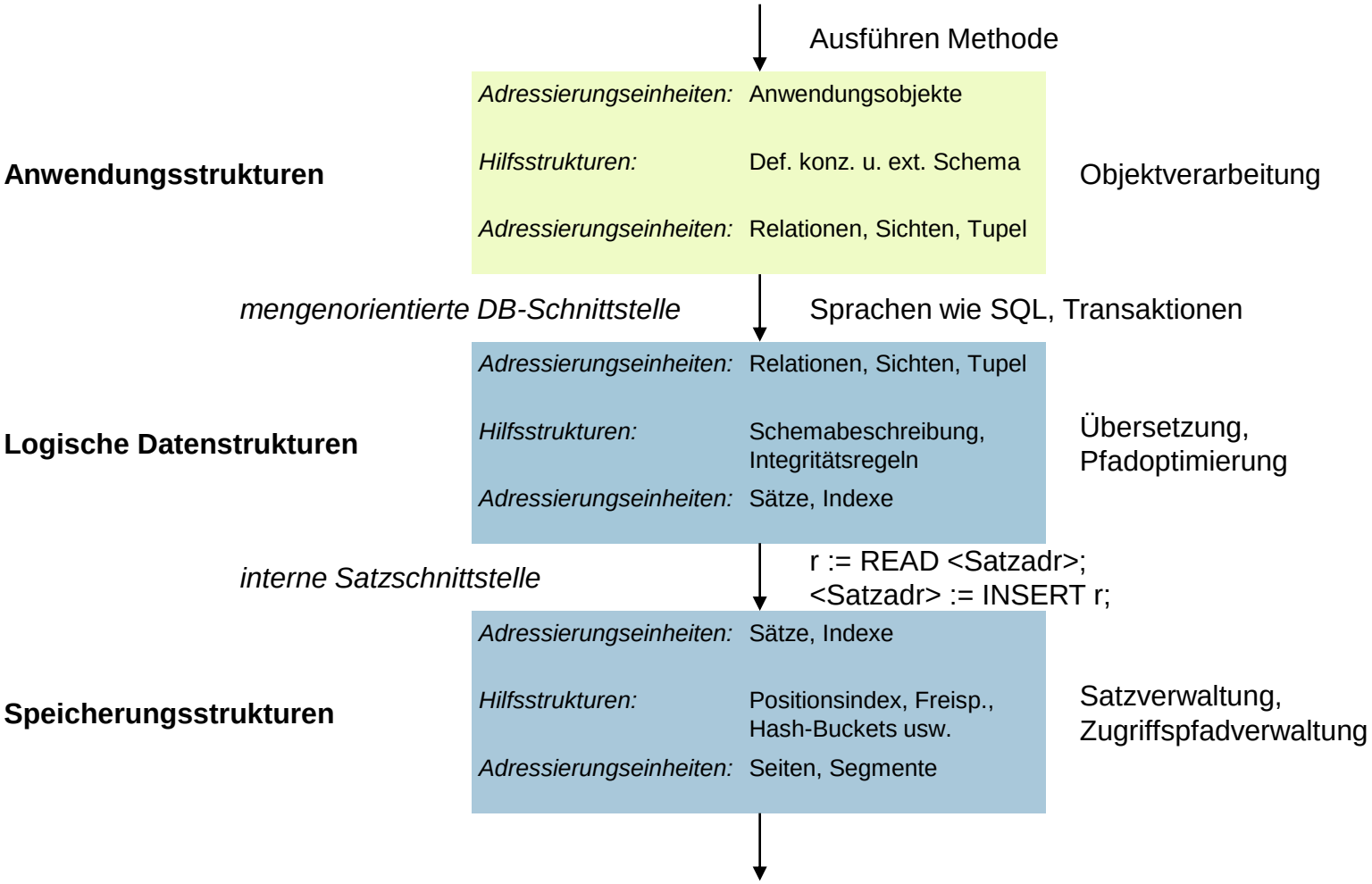
Externspeicherverwaltung

Geräteschnittstelle

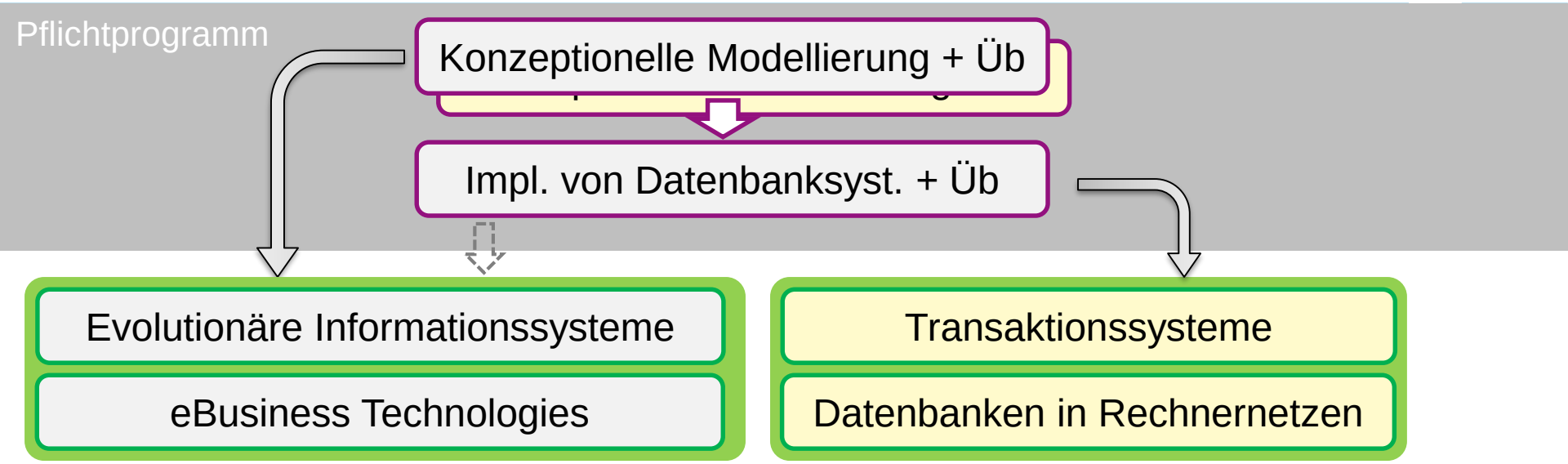
Kanalkommandos

Phys. DB

Schichtenmodell – oberer Teil



15.2 Ausblick: Lehrangebot INF6



Module im Bachelor oder Master wählbar

- jeweils 5 ECTS

SS

WS

Vorlesungen und Module

Pflichtprogramm

Konzeptionelle Modellierung + Üb

Impl. von Datenbanksyst. + Üb

Evolutionäre Informationssysteme

eBusiness Technologies

(Wintersemester)

Transaktionssysteme

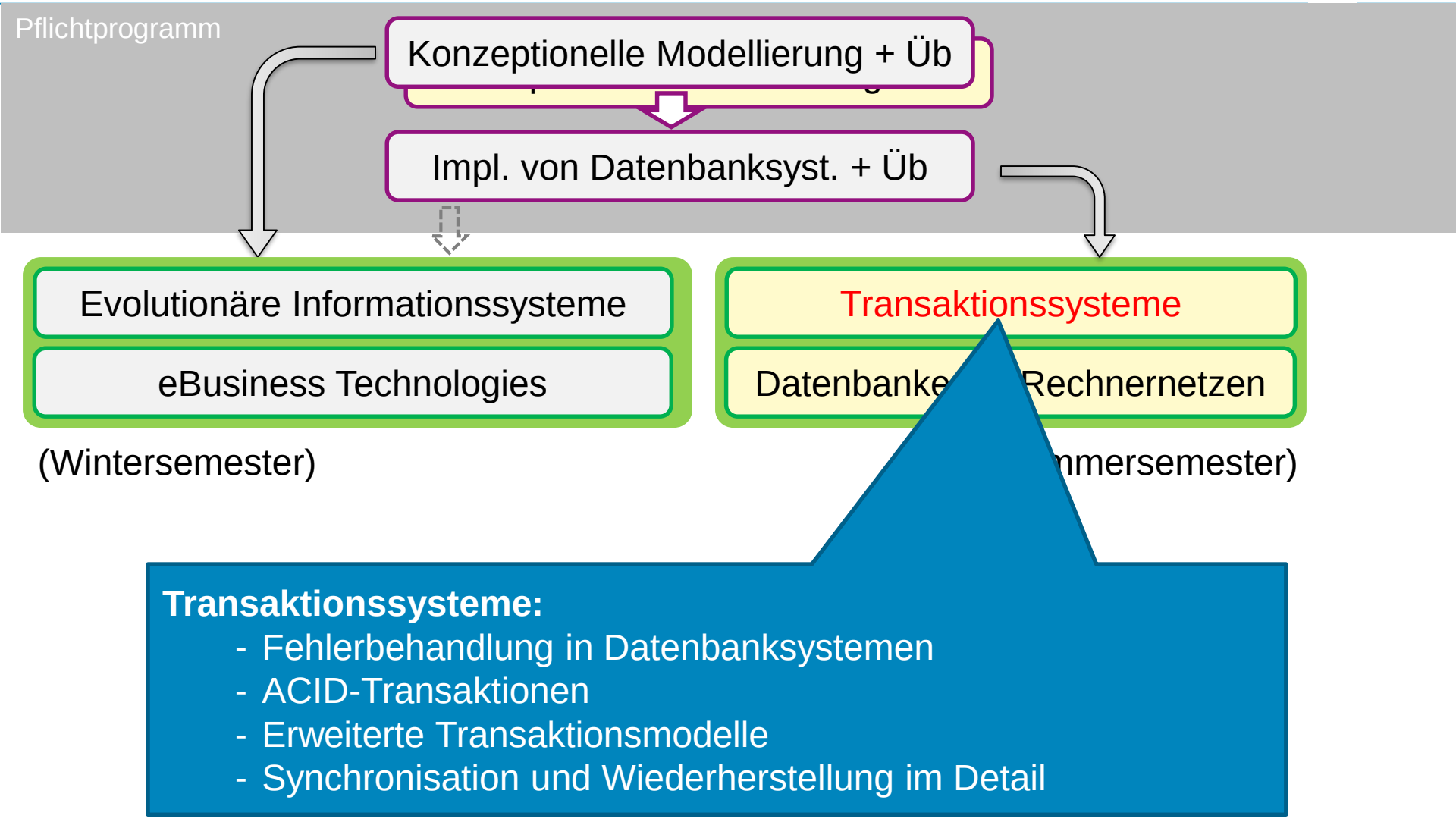
Datenbanken in Rechnernetzen

(Sommersemester)

EIS und eBT:

- Einsatz von Datenbanksystemen in Unternehmen
- Datenbank-zentrierte Web-Anwendungen
- Transparenter Zugriff auf Datenbanken
- Umgang mit sich änderndem Bedarf

Vorlesungen und Module (2)



Transaktionssysteme:

- Fehlerbehandlung in Datenbanksystemen
- ACID-Transaktionen
- Erweiterte Transaktionsmodelle
- Synchronisation und Wiederherstellung im Detail

Vorlesungen und Module (3)

Pflichtprogramm

Konzeptionelle Modellierung + Üb

Impl. von Datenbanksys. + Üb

Evolutionäre Informationssysteme

eBusiness Technologies

(Wintersemester)

Transaktionssysteme

Datenbanken in Rechnernetzen

mmersemester)

Datenbanken in Rechnernetzen:

- Verteilte Datenbanken
- Verteilte Anfrageverarbeitung
- Datenreplikation
- Heterogene Datenbanken → Föderative Datenbanken
- Parallele Anfrageverarbeitung: Umgang mit "Big Data"

Übergang Bachelor – Master

Pflichtprogramm

Konzeptionelle Modellierung + Üb

Impl. von Datenbanksyst. + Üb

★ Evolutionäre Informationssysteme (EIS)

★ eBusiness Technologies (EBT)

★ Transaktionssysteme (TAS) E

★ Datenbanken in Rechnernetzen (DBRN)

Prozessorientierte IS (POIS) + Übung E

★ Data Warehousing

Elektronische Signaturen

Aspekte des Datenschutzes

★ Je 2 Veranstaltungen im selben Semester kombinierbar zu einem 5-ECTS-Modul

E = Englisch

Ergänzende Angebote

- **Bachelor-Seminar**
  - Z.B. im Wintersemester 2019/20: Big Data
- **Praktikum**
  - Z.B. im Sommersemester 2020: SWAT ("SWAT is a Web Application Tutorial" ☺ ; siehe nächste Folie
- **Bachelorarbeit**
  - Angebote stets auch im UnivIS

- **Master-Seminar**
  - Z.B. im Sommersemester 2020: Big Data und Topologische Datenanalyse
- **Projekt**
  - Z.B. im Sommersemester 2020: individuell und projektbezogen (UnivIS)
- **Masterarbeit**
  - Angebote stets auch im UnivIS

SWAT is a Web Application Tutorial SS2020

- **Task:**
  - Entwurf und Implementierung einer typischen Web-Applikation
  - Kreatives Arbeiten im Team
  - Agile Softwareentwicklung
  - Verwendung von aktuellen Technologien
  - Moderne Programmiertechniken
- **Tools & Techniques:**
  - Apache Jena; Spring; Vaadin
  - Java; SPARQL; RDF; SCRUM

Topic SS2020: Graphbasierte Datenspeicherung und -auswertung

vaadin

Jena

spring®

Anmeldung auf StudOn

15.3 Forschungsthemen am Lehrstuhl INF6

Datenbanksysteme

Effiziente Speicherung und Verarbeitung von Daten

Evolutionäre Informationssysteme

Umgang mit Veränderung
Architektur evolutionärer Informationssysteme
Prozessunterstützung

Datenqualität

Methoden und Werkzeuge zur
Verbesserung der Datenqualität
in Unternehmen

Datenstromsysteme und Ereignisverarbeitung

Umgang mit zeitbehafteten Datensätzen

Forschung am Lehrstuhl INF6 (2)

	BIG-DATA-ASPEKTE
Datenbanksysteme Effiziente Verarbeitung von Daten	Neue Strategien für neue Anforderungen
Evolutionäre Informationssysteme Umgang mit Veränderung Architektur evolutionärer Informationssysteme Prozessunterstützung	Immer schnellere Reaktion auf neue Erkenntnisse
Datenqualität Methoden und Werkzeuge zur Verbesserung der Datenqualität in Unternehmen	Traditionelle Methoden der Qualitätssicherung versagen bei externen Datenquellen.
Datenstromsysteme und Ereignisverarbeitung Umgang mit zeitbehafteten Datensätzen	Aktuelle zeitbehaftete Daten müssen mit Unternehmensdaten verknüpft werden.

- **ReProVide:**
  Query Optimisation and Near-Data Processing on Reconfigurable SoCs for Big Data Analysis (DFG-Projekt)

  ➔ **Lekshmi B.G., M.Sc., M.Phil.**

  - Ausnutzung neuer Hardware für die Anfrageoptimierung in Datenbanksystemen

  - Kooperation mit Lehrstuhl für Informatik 12 (Hardware-Software-Co-Design, Prof. Teich)

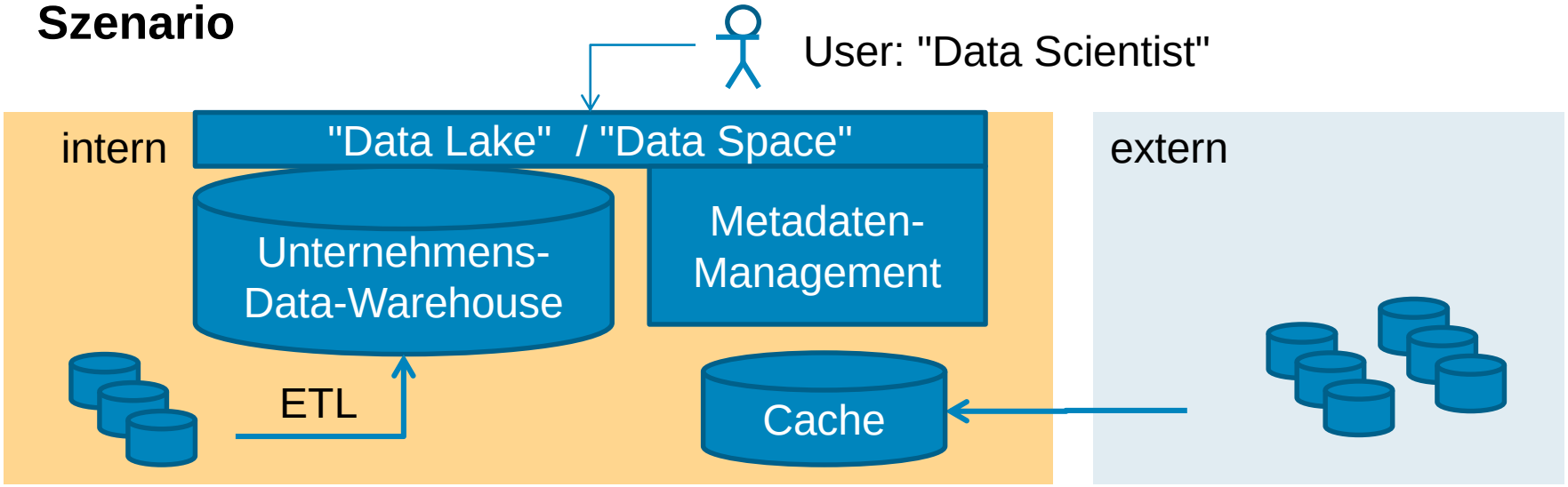
Von Altera Corporation - Altera Corporation, CC BY 3.0,
https://commons.wikimedia.org/w/index.php?curid=6642224

Evolutionäre Informationssysteme

Unternehmen unterliegen einem ständigen Wandel
IT ist Teil der Unternehmen und sollte den Wandel nicht behindern

IT-Unterstützung für die
"**Lernende Organisation**"

- **Szenario**

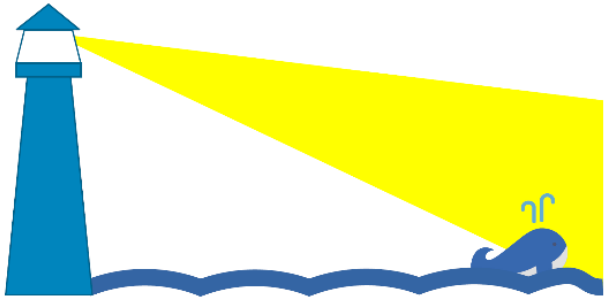


Evolutionäre Informationssysteme (2)

- **Anfrage-getriebenes offenes Datenmanagement für dynamisch wachsende Unternehmensdatenbestände**

  - Anfrage-getriebene bedarfsorientierte Schemaentwicklung
    ➔ **Peter Schwab**

  - **PHAROS:** Learning schemata from SQL Queries and describing them in a knowledge graph
    ➔ **David Haller**



Datenqualität

- **Warum Datenqualität ?**

Datenqualitätsprobleme verursachen in USA jährlich Kosten in Höhe von $ 600 000 000 000.

Quelle: Wayne Eckerson:
Data Quality and the Bottom Line: Achieving Business Success through a Commitment to High Quality Data.
The Data Warehouse Institute. Repost Series. 2002

Das "Jahr-2000"-Problem war ein Datenqualitätsproblem, das geschätzte Kosten von $ 1 500 000 000 000 verursacht hat.

Quelle: L.P. English:
Improving Data Warehouse and Business Information Quality.
Wiley & Sons. 1999

Datenqualität (2)

- **Projekte**
  - **SIML – Schema Inference and Machine Learning**
    Knowledge Discovery; Functional Dependency; topologische Datenanalyse; Clustering; unsupervised Learning; Neural-Network Theory
    ➔ **Luciano Melodia**

Datenstromsysteme

Datenbanksysteme:

- Es kommt genau so viel, wie ich möchte – aber nicht von allein.

Datenstromsysteme:

- Ich will das verarbeiten, was sowieso kommt.

Datenstromsysteme (2)

- **Data-Stream Management System (DSMS)**
  - Keine Speicherung von Daten, Auswertung "on the fly", geringe Latenz
  - **Sensornetzwerke** als Datenquelle
  - Identifikation von Ereignissen (Complex-Event Processing, CEP)
  - Data-Stream Application Manager (DSAM)

- **Projekte:**
  - EFRE-**E|ASY-Opt** Teilprojekt
    Digitalisierung: **Industrie 4.0 und Predictive Maintenance**
    ➔ Vorausschauende Wartung der anfallenden Sensordaten bei Fertigungsanlagen mittels Data Mining bzw. Machine Learning
    ➔ **Melanie Sigl**

Digital Humanities

- **Informatik in den Geistes- und Sozialwissenschaften**
  - Analyse von Texten und Bildern, umfassende Annotationen
  - Wissensbanken
  - Multimedia

- **Projekt:**
  - **WissKI**:
    Wissenschaftliche Kommunikationsinfrastruktur;
    in der AG Digital Humanities (Prof. Dr. Günther Görz);
    Content Management (Drupal), RDF, OWL, Multimedia
    ➔ **Corina Lehmann, Peggy Große** (AG Digital Humanities, GNM)

15.4 Prüfung (schriftlich)

- **Vorauss. am Freitag, dem 14. Februar 2020**
  - Uhrzeit und Räume stehen noch nicht fest.
- **Dauer: 90 Minuten**
- **Viel Multiple Choice**
- **Keine Hilfsmittel!**

- **Fragestunde heute, hier**
- **Fragen in der Vorbereitung werden auch im StudOn-Forum beantwortet.**

Viel Erfolg und alles Gute!!!