

**Vorlesung**

# **Datenbankimplementierungstechniken / Datenbanken II**

***OvGU Magdeburg, SomSem 2010***

Gunter Saake

`saake@iti.cs.uni-magdeburg.de`

# Überblick

---

1. Aufgaben und Prinzipien von Datenbanksystemen
2. Architektur von Datenbanksystemen
3. Verwaltung des Hintergrundspeichers
4. Dateiorganisation und Zugriffsstrukturen
5. Zugriffsstrukturen für spezielle Anwendungen
6. Basisalgorithmen für Datenbankoperationen
7. Optimierung von Anfragen
8. Weitere Aspekte und Ausblick

# Nötiges Vorwissen

---

Datenbanken I:

- Grundprinzipien Datenbanksysteme
- Tabellen, Attribute, Schlüssel
- Relationale Algebra und SQL

*Wird am Anfang der Vorlesung kurz wiederholt!*

# Literatur

---

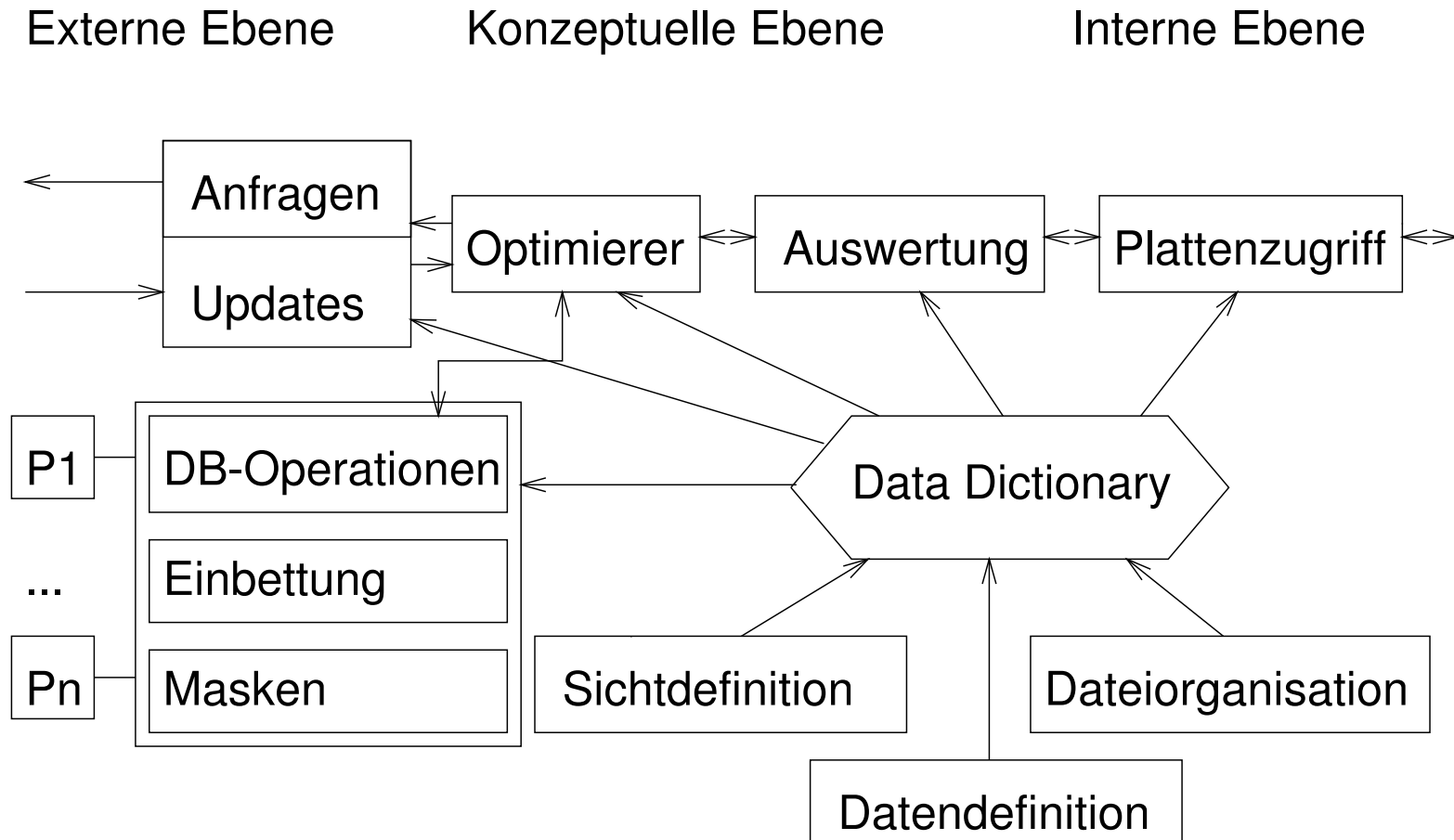
- Saake, G.; Heuer, A., Sattler, K.-U.: *Datenbanken — Implementierungskonzepte*. 2. Auflage, mitp-Verlag, Mai 2005
- Härder, T.; Rahm, E.: *Datenbanksysteme — Konzepte und Techniken der Implementierung*. Springer-Verlag, 2001
- Garcia-Molina, H.; Ullman, J.; Widom, J.: *Database System Implementation*. Addison-Wesley, 1999.
- Silberschatz, A.; Korth, H. F.; Sudarshan, S.: *Database System Concepts*. Wiley & Sons, 2001.

# 1. Aufgaben und Prinzipien von DBS

---

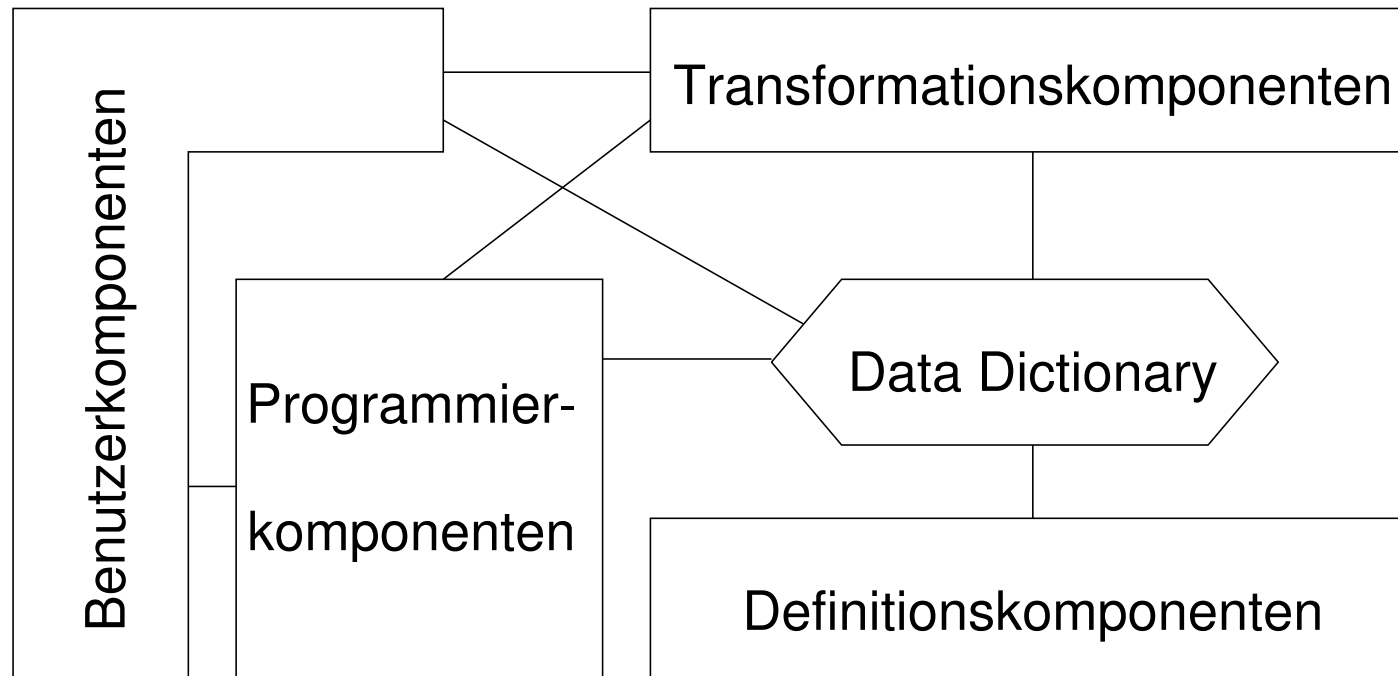
- Wiederholung Datenbankgrundbegriffe
- Überblick über behandelte Komponenten

# Datenbankgrundbegriffe: Komponenten



# Klassifikation von Komponenten

---



# Neun Funktionen nach Codd

---

1. Integration
2. Operationen
3. *Katalog*
4. *Benutzersichten*
5. Konsistenzüberwachung
6. Datenschutz
7. *Transaktionen*
8. *Synchronisation*
9. *Datensicherung*



# Datenmodelle und Datendefinition

---

Wichtigste Modelle in kommerziellen Systemen

- das *hierarchische Datenmodell*: Daten in Baumform als hierarchisch strukturierte Datensätze,
- das *Netzwerkmodell*: Unterstützung von Netzwerken von verzeigten Datensätzen,
- das *relationale Datenbankmodell*: Daten in Tabellenform,
- das *objektorientierte Datenmodell*: modelliert Daten objektorientiert durch in Klassen organisierte, verzeigte Objekte,
- das *semistrukturierte Datenmodell*: Verwaltung „schemaloser“, selbstbeschreibender Daten in Graphstrukturen (XML).

# Relationale Datenbanken

---

Ausleih	InventarNr	Name
	4711	Meyer
	1201	Schulz
	0007	Müller
	4712	Meyer

Buch	InventarNr	Titel	ISBN	AUTOR
	0007	Dr. No	3-125	James Bond
	1201	Objektbanken	3-111	Heuer
	4711	Datenbanken	3-765	Vossen
	4712	Datenbanken	3-891	Ullman
	4717	Pascal	3-999	Wirth

# SQL-DDL

---

```
create table Buch (  
    ISBN    char(10),  
    Titel   varchar(200),  
    Verlagsname varchar(30),  
    primary key (ISBN),  
    foreign key (Verlagsname)  
        references Verlage (Verlagsname) )
```

# Anfragen

---

## Grundlagen

- Relationenalgebra sowie
- Tupel- oder Bereichskalkül.

# Relationenalgebra

---

$$\sigma_{\text{Name}='Meyer'}(r(\text{Ausleih}))$$

$$\pi_{\text{Titel}}(r(\text{Buch}))$$

$$\pi_{\text{InventarNr},\text{Titel}}(r(\text{Buch})) \bowtie \sigma_{\text{Name}='Meyer'}(r(\text{Ausleih}))$$

# Änderungskomponente

---

Änderungskomponente eines Datenbanksystems ermöglicht es,

- Tupel einzugeben,
- Tupel zu löschen und
- Tupel zu ändern.

# Sprachen und Sichten: SQL

---

```
select Buch.InventarNr, Titel, Name
from    Buch, Ausleih
where   Name = 'Meyer' and
          Buch.InventarNr = Ausleih.InventarNr
```

```
update Angestellte
set     Gehalt = Gehalt + 1000
where   Gehalt < 5000
```

```
insert into Buch values
    (4867, 'Wissensbanken', '3-876', 'Karajan')
```

```
insert into Kunde
    ( select LName, LAdr, 0 from Lieferant )
```

# Spracheinbettung

---

```
exec sql declare AktBuch cursor for  
    select ISBN, Titel, Verlagsname  
    from Buch  
    for update of ISBN, Titel;  
  
exec sql commit work;  
  
exec sql rollback work;
```



# Sichten in SQL

---

```
create view Meyers as  
select Buch.InventarNr, Titel, Name  
from Buch, Ausleih  
where Name = 'Meyer' and  
        Buch.InventarNr = Ausleih.InventarNr
```

# Überblick über behandelte Komponenten

---

- *Optimierer*
- *Dateiorganisationen und Zugriffspfade*
- *Organisation des Sekundärspeichers*
- *Transaktionsverwaltung*
- *Recovery-Komponente*

# Optimierer

---

- Äquivalenz von Algebra-Termen
  1.  $\sigma_{A=Konst} (REL1 \bowtie REL2)$  und  $A$  aus  $REL1$
  2.  $\sigma_{A=Konst} (REL1) \bowtie REL2$
- allgemeine Strategie: Selektionen möglichst früh, da sie Tupelanzahlen in Relationen verkleinern
- Beispiel:  $REL1$  100 Tupel,  $REL2$  50 Tupel  
intern: Tupel sequentiell abgelegt
  1.  $5000 (\bowtie) + 5000 (\sigma) = 10000$  Operationen
  2.  $100 (\sigma) + 10 \cdot 50 (\bowtie) = 600$  Operationen  
falls 10 Tupel in  $REL1$  die Bedingung  $A = Konst$  erfüllen

# Joins

---

- Merge-Join: *Verbund durch Mischen* von  $R_1$  und  $R_2$ 
  - ◆ insbesondere dann effizient, wenn eine oder beide Relation(en) sortiert nach den Verbund-Attributen vorliegen, d.h. für Verbund-Attribute  $X$  muss gelten:  
$$X := R_1 \cap R_2$$
  - ◆  $r_1$  und  $r_2$  werden nach  $X$  sortiert
  - ◆ Mischen von  $r_1$  und  $r_2$ , d.h., beide Relationen parallel sequentiell durchlaufen und passende Paare in das Ergebnis aufnehmen
- Nested-Loops-Join: doppelt Schleife über  $R_1$  und  $R_2$ 
  - ◆ liegt bei einer der beiden Relationen ein Zugriffspfad für  $X$  vor, dann innere Schleife durch Zugriff über diesen Zugriffspfad ersetzen

# Komplexität der Operationen

---

## ■ Selektion

- ◆ hash-basierte Zugriffsstruktur:  $O(1)$
- ◆ sequentieller Durchlauf:  $O(n)$
- ◆ in der Regel (baumbasierte Zugriffspfade):  $O(\log n)$

## ■ Verbund

- ◆ sortiert vorliegende Tabellen:  $O(n + m)$  (Merge Join)
- ◆ sonst: bis zu  $O(n * m)$  (Nested Loops Join)

## ■ Projektion

- ◆ vorliegender Zugriffspfad oder Projektion auf Schlüssel:  $O(n)$
- ◆ Duplikateliminierung durch Sortieren:  $O(n \log n)$

# Optimierungsarten

---

## ■ *Logische Optimierung:*

- ◆ nutzt nur algebraische Eigenschaften der Operationen
- ◆ *keine* Informationen über die Speicherungsstrukturen und Zugriffspfade
- ◆ Verwendung heuristischer Regeln anstelle exakter Optimierung
- ◆ Beispiele:
  - Entfernung redundanter Operationen
  - Verschieben von Operationen derart, daß Selektionen möglichst früh ausgeführt werden

~> *algebraische Optimierung*

# Optimierungsarten II

---

## ■ *Interne Optimierung:*

- ◆ Nutzung von Informationen über die vorhandenen Speicherungsstrukturen
- ◆ Auswahl der Implementierungsstrategie einzelner Operationen (Merge Join vs. Nested-Loops-Join)
- ◆ Beispiele:
  - Verbundreihenfolge anhand der Größe und Unterstützung der Relationen durch Zugriffspfade
  - Reihenfolge von Selektionen nach der Selektivität von Attributen und dem Vorhandensein von Zugriffspfaden

# Algebraische Optimierung

---

- *Entfernen redundanter Operationen* ( $r \bowtie r = r$ )

$$\begin{aligned} r(\text{BuchLangeWeg}) = \\ r(\text{Buch}) \bowtie \pi_{\text{ISBN}, \text{Datum}}( \\ \dots \sigma_{\text{Datum} < '31.12.1990'}(r(\text{Ausleihe}))) \end{aligned}$$

- ◆ *Anfrage an Sicht:*

$$\pi_{\text{Titel}}(r(\text{Buch}) \bowtie r(\text{BuchLangeWeg}))$$

- ◆ *Einsetzen der Sichtdefinition:*

$$\pi_{\text{Titel}}(r(\text{Buch}) \bowtie r(\text{Buch}) \bowtie \pi_{\dots}(\dots))$$



# Algebraische Optimierung II

---

## ■ *Verschieben von Selektionen*

$$\sigma_{\text{Autor}='Vossen'}(r(\text{Buch}) \bowtie \pi_{\text{ISBN,Datum}}(\dots))$$

Verbund auf kleineren Zwischenergebnissen:

$$(\sigma_{\text{Autor}='Vossen'}(r(\text{Buch}))) \bowtie \pi_{\text{ISBN,Datum}}(\dots)$$

Selektion und Verbund kommutieren
-----------------------------------

# Algebraische Optimierung III

---

## ■ *Reihenfolge von Verbunden*

$$(r(\text{Verlag}) \bowtie r(\text{Ausleihe})) \bowtie r(\text{Buch})$$

Nachteil: erster Verbund entartet zum kartesischen Produkt, da keine gemeinsamen Attribute

$$r(\text{Verlag}) \bowtie (r(\text{Ausleihe}) \bowtie r(\text{Buch}))$$

$\bowtie$ assoziativ und kommutativ
-------------------------------------

# Dateiorganisation und Zugriffspfade

---

Konzeptionelle Ebene		Interne Ebene		Platte
Relationen	→	Dateien (Files)	→	
Tupel	→	Sätze (Records)	→	Blöcke
Attributwerte	→	Felder	→	

# Zugriffspfade

---

- Primär- versus Sekundär-Index
- sequentielle Dateien, B-Bäume, Hashen
- eindimensional versus mehrdimensional
- spezielle Anwendungen

# Transaktionen und Recovery

---

- **Atomicity** (Atomarität oder *Ununterbrechbarkeit*)
  - ◆ Transaktion wird ganz oder gar nicht ausgeführt
- **Consistency** (Konsistenz oder *Integritätserhaltung*)
  - ◆ der von einer Transaktion hinterlassene neue Zustand genügt den Integritätsbedingungen
- **Isolation**
  - ◆ Ergebnis einer Transaktion muß einem isolierten Ablauf dieser Transaktion entsprechen, auch bei mehreren nebenläufigen Transaktionen
- **Durability** (*Dauerhaftigkeit* oder *Persistenz*)
  - ◆ nach Ende einer Transaktion stehen Ergebnisse dauerhaft in der Datenbank

# Transaktionen II

---

Das Ergebnis einer Transaktion soll so aussehen, als sei sie nach dem ACID-Prinzip abgelaufen.

# Datenelemente und Sperren

---

Spermodelle:

$T_1$  : **lock**  $A$ ;  
      **read**  $A$ ;  
       $A := A + 1$ ;  
      **write**  $A$ ;  
      **unlock**  $A$ ;

*Deadlocks:*

$T_1$  : **lock**  $A$ ;  
      ...;  
      **lock**  $B$ ;  
      ...;  
      **unlock**  $A$ ;  
      **unlock**  $B$ ;

$T_2$  : **lock**  $B$ ;  
      ...;  
      **lock**  $A$ ;  
      ...;  
      **unlock**  $B$ ;  
      **unlock**  $A$ ;

# Serielle Schedules

---

$T_1$ : **read**  $A$ ;  
     $A := A - 10$ ;  
    **write**  $A$ ;  
    **read**  $B$ ;  
     $B := B + 10$ ;  
    **write**  $B$ ;

$T_2$ : **read**  $B$ ;  
     $B := B - 20$ ;  
    **write**  $B$ ;  
    **read**  $C$ ;  
     $C := C + 20$ ;  
    **write**  $C$ ;

$T_1; T_2$ und $T_2; T_1$ sind seriell
--



# Begriff der Serialisierbarkeit

---

Ein Schedule heißt *serialisierbar*, wenn sein Ergebnis äquivalent zu dem eines seriellen Schedules ist.

Methoden:

- Serialisierbarkeitsgraphen
- Zwei-Phasen-Sperr-Protokoll
- Zeitmarkenverfahren

# Unterschiedliche Ablaufpläne

Schedule $S_1$		Schedule $S_2$		Schedule $S_3$	
$T_1$	$T_2$	$T_1$	$T_2$	$T_1$	$T_2$
<b>read</b> $A$		<b>read</b> $A$		<b>read</b> $A$	
$A - 10$			<b>read</b> $B$	$A - 10$	
<b>write</b> $A$		$A - 10$			<b>read</b> $B$
<b>read</b> $B$			$B - 20$	<b>write</b> $A$	
$B + 10$		<b>write</b> $A$			$B - 20$
<b>write</b> $B$			<b>write</b> $B$	<b>read</b> $B$	
	<b>read</b> $B$	<b>read</b> $B$			<b>write</b> $B$
	$B - 20$		<b>read</b> $C$	$B + 10$	
	<b>write</b> $B$	$B + 10$			<b>read</b> $C$
	<b>read</b> $C$		$C + 20$	<b>write</b> $B$	
	$C + 20$	<b>write</b> $B$			$C + 20$
	<b>write</b> $C$		<b>write</b> $C$		<b>write</b> $C$

# Kaskadierende Transaktionsabbrüche

	$T_1$	$T_2$	
	<b>lock</b> $A$ <b>read</b> $A$ $A := A - 1$ <b>write</b> $A$ <b>lock</b> $B$ <b>unlock</b> $A$	<b>lock</b> $A$ <b>read</b> $A$ $A := A \times 2$  <b>write</b> $A$ <b>unlock</b> $A$	$\leftarrow \text{commit } T_2$
<b>abort</b> $T_1 \rightarrow$	$B := B/A \downarrow$		

# Recovery

---

- stabiler vs. *instabiler* Speicher
- Log-Buch / Journal
- *Backward Recovery*: Änderungen *rückgängig* machen  
     $\leadsto$  UNDO
- *Forward Recovery*: Änderungen nachziehen  $\leadsto$  REDO
- Schattenspeicher

## 2. Architektur von Datenbanksystemen

---

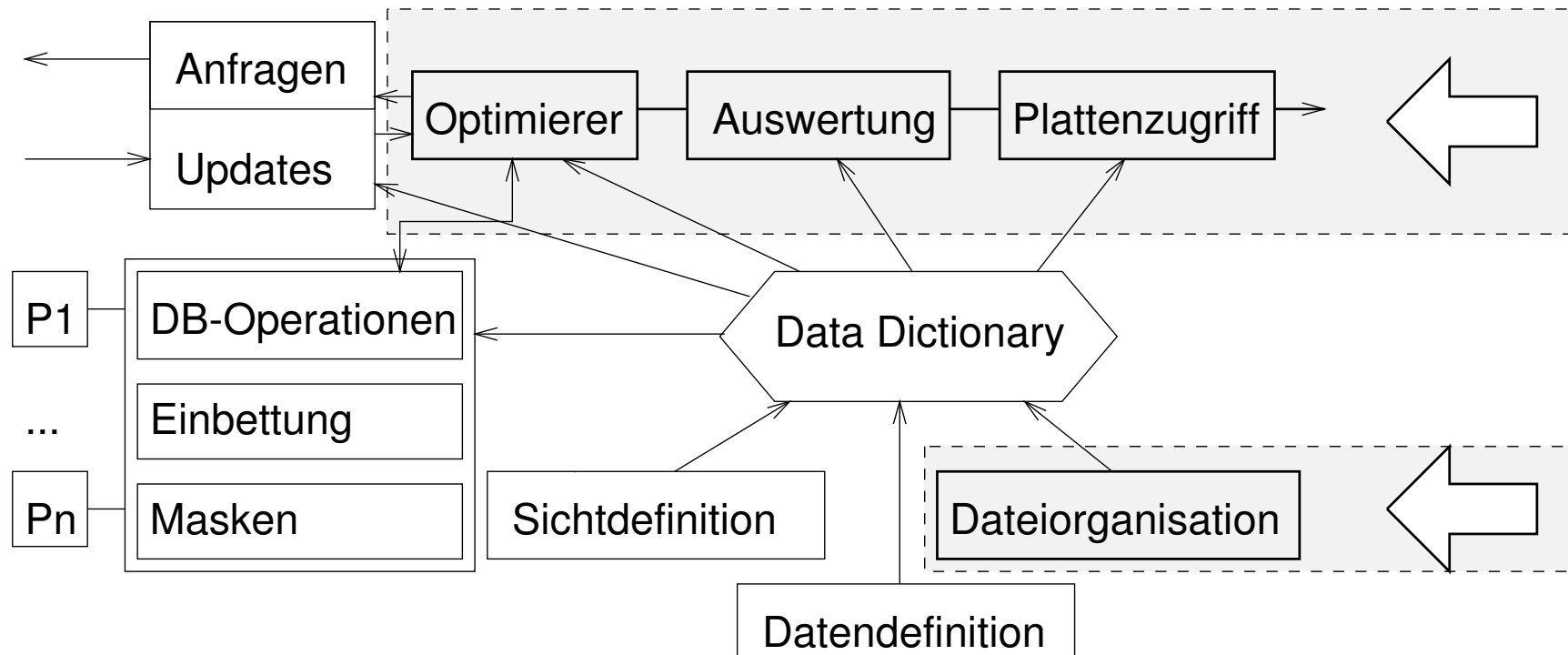
- Betrachtete Fragestellungen
- Schichtenmodell eines relationalen DBMS
- Hardware und Betriebssystem
- Pufferverwaltung
- Speichersystem
- Zugriffssystem
- Datensystem

# Betrachtete Fragestellungen

Externe Ebene

Konzeptuelle Ebene

Interne Ebene



# Fünf-Schichten-Architektur

---

- basierend auf Idee von Senko 1973
- Weiterentwicklung von Härder 1987
- Umsetzung im Rahmen des IBM-Prototyps *System R*
- genauere Beschreibung der Transformationskomponenten
  - ◆ schrittweise Transformation von Anfragen/Änderungen bis hin zu Zugriffen auf Speichermedien
  - ◆ Definition der Schnittstellen zwischen Komponenten

# 5-Schichten-Architektur: Schnittstellen I

---

- mengenorientierte Schnittstelle
  - ◆ deklarative DML auf Tabellen, Sichten, Zeilen
- satzorientierte Schnittstelle
  - ◆ Sätze, logische Dateien, logische Zugriffspfade
  - ◆ navigierender Zugriff
- interne Satzschnittstelle
  - ◆ Sätze, Zugriffspfade
  - ◆ Manipulation von Sätzen und Zugriffspfaden



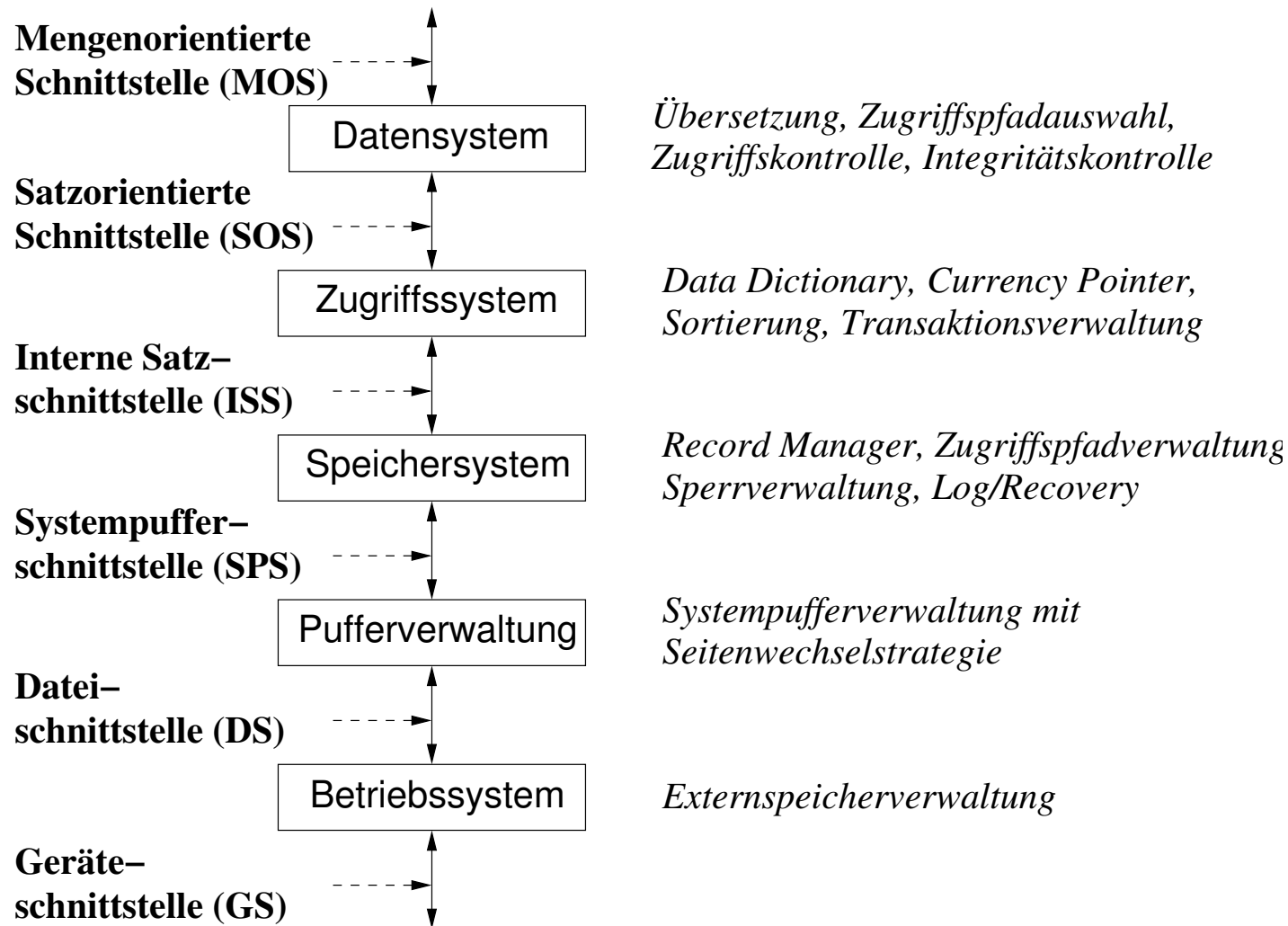
# 5-Schichten-Architektur: Schnittstellen II

---

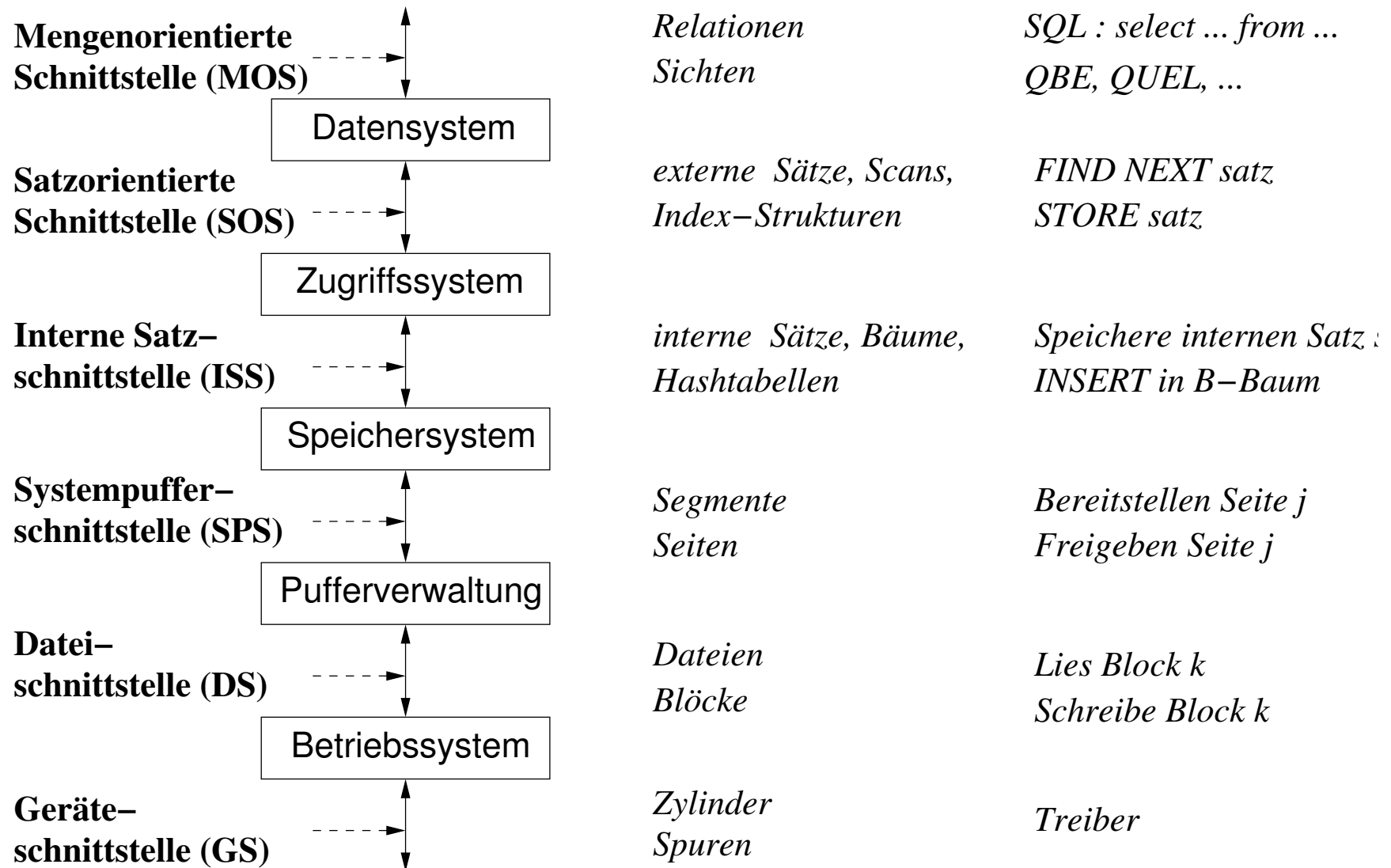
- Pufferschnittstelle
  - ◆ Seiten, Seitenadressen
  - ◆ Freigeben und Bereitstellen
- Datei- oder Seitenschnittstelle
  - ◆ Hole Seite, Schreibe Seite
- Geräteschnittstelle
  - ◆ Spuren, Zylinder
  - ◆ Armbewegungen

# 5-Schichten-Architektur: Funktionen

---



# 5-Schichten-Architektur: Objekte



# Erläuterungen (I)

---

- mengenorientierte Schnittstelle **MOS**:
  - ◆ deklarative Datenmanipulationssprache auf Tabellen und Sichten (etwa SQL)
- durch Datensystem auf satzorientierte Schnittstelle **SOS** umgesetzt:
  - ◆ navigierender Zugriff auf interner Darstellung der Relationen
  - ◆ manipulierte Objekte: typisierte Datensätze und interne Relationen sowie logische Zugriffspfade (Indexe)
  - ◆ Aufgaben des Datensystems: Übersetzung und Optimierung von SQL-Anfragen

# Erläuterungen (II)

---

- durch Zugriffssystem auf interne Satzchnittstelle **ISS** umgesetzt:
  - ◆ interne Tupel einheitlich verwalten, ohne Typisierung
  - ◆ Speicherstrukturen der Zugriffspfade (konkrete Operationen auf B\*-Bäumen und Hash-Tabellen), Mehrbenutzerbetrieb mit Transaktionen

# Erläuterungen III

---

- durch Speichersystem Datenstrukturen und Operationen der ISS auf interne Seiten eines virtuellen linearen Adressraums umsetzen
  - ◆ Manipulation des Adressraums durch Operationen der Systempufferschnittstelle **SPS**
  - ◆ Typische Objekte: interne Seiten, Seitenadressen
  - ◆ Typische Operationen: Freigeben und Bereitstellen von Seiten, Seitenwechselstrategien, Sperrverwaltung, Schreiben des Log-Buchs
- durch Pufferverwaltung interne Seiten auf Blöcke der Dateischnittstelle **DS** abbilden
  - ◆ Umsetzung der DS-Operationen auf Geräteschnittstelle erfolgt durch BS

# Hardware und Betriebssystem

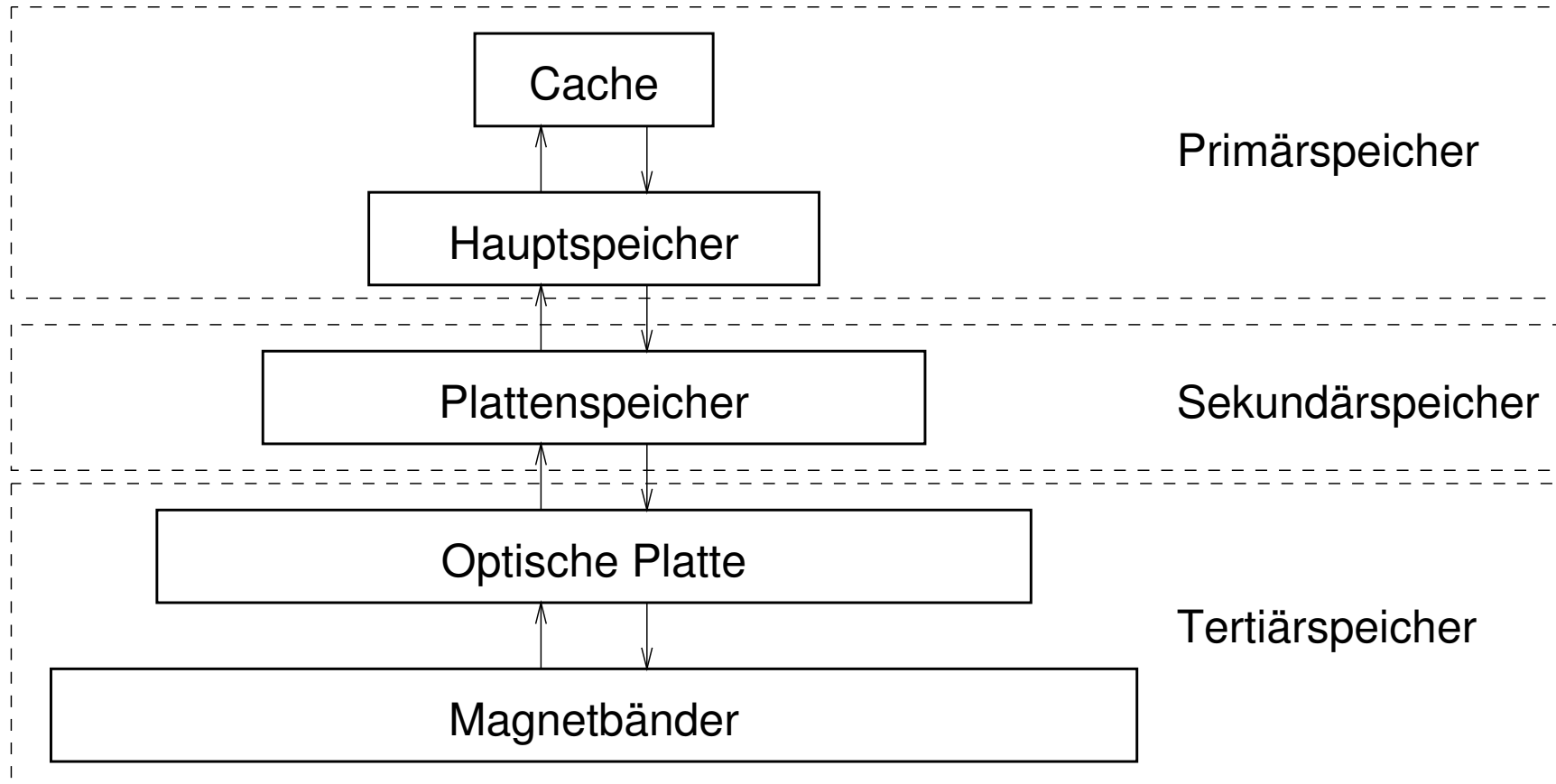
---

- Betriebssystemebene: Grundlage für datenbankbezogene Ebenen
  - ◆ benötigt: Treiberprogramme zum Zugriff auf die Daten von Medien; Caching-Mechanismen
- Prozessoren und Rechnerarchitekturen: klassische Industrie-Standards (aber: Datenbankmaschinen)
- Speichermedien: spezielle Anforderungen Speicherhierarchie

# Klassische Speicherhierarchie

---

Primär-, Sekundär- und Tertiärspeicher





# Eigenschaften von Speichermedien

---

	<b>Primär</b>	<b>Sekundär</b>	<b>Tertiär</b>
Geschwindigkeit	schnell	langsam	sehr langsam
Preis	teuer	preiswert	billig
Stabilität	flüchtig	stabil	stabil
Größe	klein	groß	sehr groß
Granulate	fein	grob	grob

# Primärspeicher

---

- Primärspeicher: Cache und Hauptspeicher
- sehr schnell, Zugriff auf Daten fein granular: jedes Byte adressierbar
- 32-Bit-Adressierung: nur  $2^{32}$  Bytes direkt adressierbar, somit Primärspeicher von der Größe stark eingeschränkt
- hohe Anschaffungskosten pro Byte
- flüchtiges Speichermedium (volatile, non-reliable)

# Sekundärspeicher

---

- Sekundärspeicher oder Online-Speicher
  - ◆ meist Plattenspeicher, nicht-flüchtig (stable, non-volatile, reliable)
  - ◆ weitaus größer, mehrere Gigabyte Speicherkapazität pro Medium
  - ◆ um Größenordnungen preiswerter
  - ◆ leider Daten nicht direkt verarbeitbar
  - ◆ Granularität des Zugriffs gröber: Blöcke, oft 512 Bytes (oder Mehrfaches davon)
  - ◆ Zugriffslücke: Faktor  $10^5$  langsamerer Zugriff
- erforderlich: intelligente Pufferverwaltung, gute Anfrageoptimierung
- *aber: neue Entwicklungen: Flash-Speicher*

# Tertiärspeicher

---

- Zur langfristigen Datensicherung (Archivierung) oder kurzfristigen Protokollierung (Journale) von Datenbeständen und Datenbankveränderungen
- Sekundärspeicher dazu zu teuer und zu klein
- mehrere hundert Gigabytes oder sogar Terabytes von Daten: Tertiärspeicher, Offline-Speicher, Archivspeicher
- üblich: optische Platten, Magnetbänder
- „Offline-Speicher“ meist Wechselmedium
- Nachteil: Zugriffslücke extrem groß: Zugriff auf das sequentielle Medium, Holen eines Bandes, Einlegen dieses Bandes (auch bei Automatisierung: Bandroboter, Jukeboxes)

# Angebotene Dienste

---

- Treiberprogramme zum Holen und Schreiben von Blöcken
- Zuordnung von Blöcken zu Seiten
- Ergänzen der Block-Informationen um Kontrollsummen, um Schreib- oder Lesefehler zu ermitteln
- Caching-Mechanismen, die bereits gelesene Daten im Hauptspeicher halten und verwalten
- Operationen des Dateisystems von Betriebssystemen (oft: Datenbanksysteme nutzen nur eine einzige Datei)

# Pufferverwaltung

---

- benötigte Blöcke des Sekundärspeichers im Hauptspeicher verwalten
- Speicherplatz für begrenzte Menge von Seiten im Hauptspeicher: *Puffer*
- Aufgabe der Pufferverwaltung: Verdrängung nicht mehr im Puffer benötigter Seiten (Seitenwechselstrategien)
- Unterschied: unter Verantwortung des Datenbanksystems verwalteter Puffer  $\leftrightarrow$  Cache auf der Betriebssystemebene

# Puffer

---

- je nach Hauptspeichergröße beträchtlicher Umfang (in üblichen Fällen bis zu 12 GB)
- trotzdem nur ganz geringer Bruchteil der Datenbank (weniger als 1%)
- alle Lese- und Schreib-Vorgänge von oder auf Seiten im Puffer
- dadurch Puffer oft Flaschenhals
- verfügbarer Hauptspeicher sehr groß und Datenbank relativ klein  $\Rightarrow$  gesamte Datenbank (etwa beim Start des Systems) in den Puffer:  
*Hauptspeicher-Datenbanken* (engl. *main memory databases*)

# Pufferverwaltung: Aufgaben

---

- Zuteilung von Speicherplatz für Seiten
- Suchen und Ersetzen von Seiten im Puffer
- Optimierung der Lastverteilung zwischen parallelen Transaktionen



# Ablauf eines Zugriffs auf eine Seite (I)

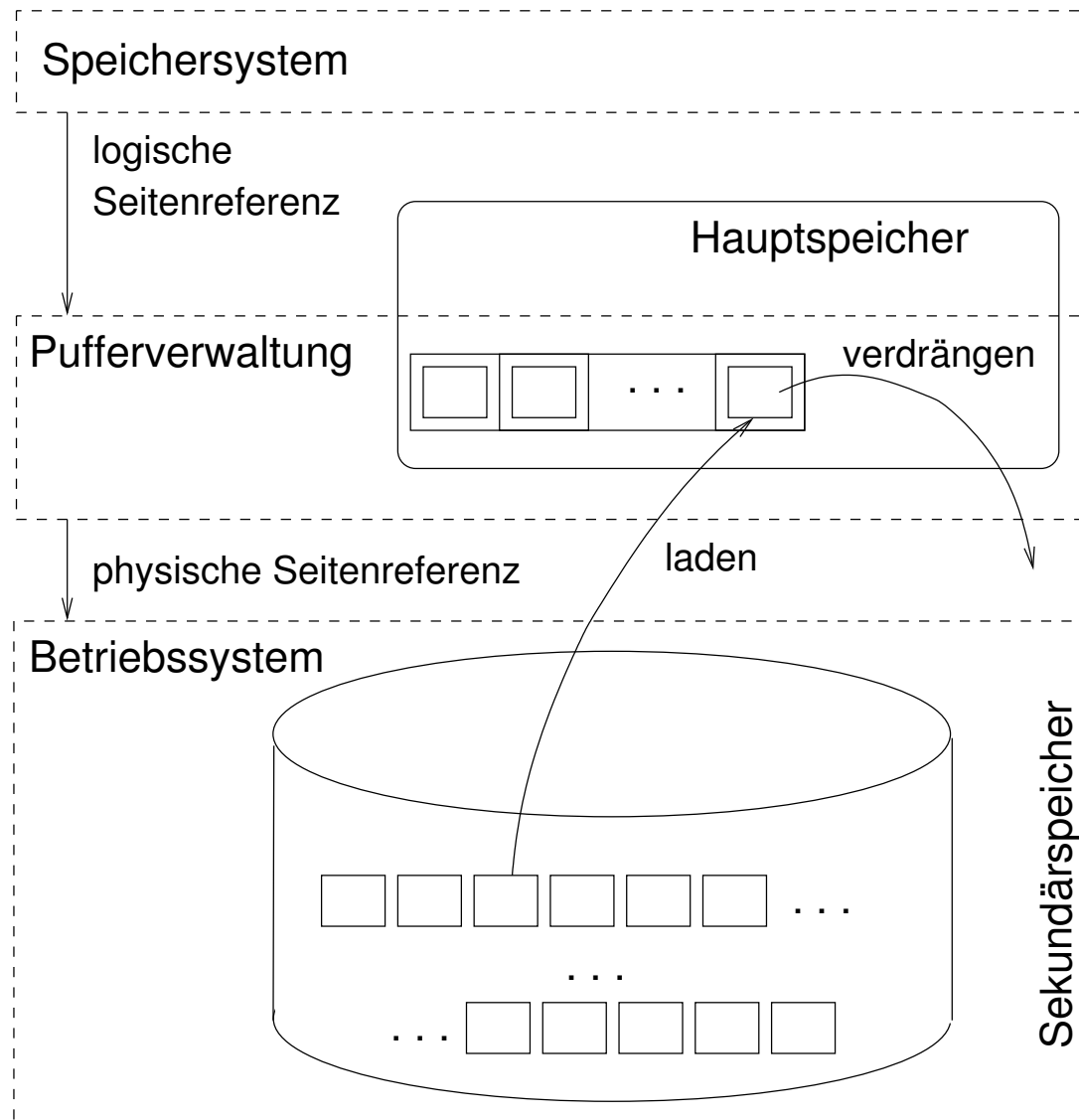
---

höhere Schicht (Speichersystem) fordert bei Pufferverwaltung Seite an (*logische Seitenreferenz*)

- angeforderte Seite im Puffer: wird dem Speichersystem zur Verfügung gestellt
- angeforderte Seite nicht im Puffer (*page fault*):  
*physische Seitenreferenz* durch Pufferverwaltung an Betriebssystemebene; i.a. bei gefülltem Puffer eine Seite aus dem Puffer verdrängen; falls die zu verdrängende Seite geändert wurde, vorher auf den Sekundärspeicher schreiben

Aufwand pro E/A-Operation: 2500 Instruktionen in CPU; 15 bis 30 ms für Zugriff auf Sekundärspeicher

# Ablauf eines Zugriffs auf eine Seite (II)



# Speichersystem

---

Puffer: Seiten (Byte-Container)  $\leftrightarrow$  Speichersystem: interne Datensätze  $\leftrightarrow$  Zugriffssystem: logische Datensätze, interne Tupel

Struktur	Systemkomponente
Tupel	Datensystem
internes Tupel oder logischer Datensatz	Zugriffssystem
interner Datensatz	Speichersystem

- Anwendungsobjekte im Speichersystem als interne Sätze
- Hilfsdaten wie Indexeinträge als interne Sätze

# Adressierung von Sätzen

---

- Problem bei Adressierung: Änderungen im Datenbestand  $\Rightarrow$  effiziente Änderungen von Adressen
- Bsp.: interne Sätze adressiert mit Offset  $x$  relativ zum Seitenanfang (interner Satz startet auf Byte  $x$ )  $\Rightarrow$  Änderungen auf dieser Seite haben Auswirkungen auf verwendete Tupeladresse
- Besser: *TID-Konzept* (Tupel-Identifizier)
  - ◆ Adresse: Seitennummer und Offset in einer Liste von Tupelzeigern am Anfang der Seite
  - ◆ Eintrag im Zeigerfeld bestimmt Offset des Satzes
  - ◆ ändert sich Position des internen Satzes auf der Seite  $\Rightarrow$  nur Eintrag lokal im Zeigerfeld verändern; alle „außen“ verwendeten Adressen bleiben stabil

# Satztypen

---

- *Nicht-Spannsätze (unspanned records)* auf maximal eine Seite; Satz zu groß für eine in Bearbeitung befindliche Seite  $\Rightarrow$  von Freispeicherverwaltung eine neue Seite anfordern
- *Spannsätze (spanned records)* können mehrere Seiten überspannen; Satz zu groß  $\Rightarrow$  Beginn des Satzes auf dieser Seite, Überlauf auf neuer Seite speichern
- *Sätze fester Länge*: für bestimmten Tupeltyp feste Anzahl von Bytes (bei *string* alle Attributwerte mit gleicher Anzahl Bytes speichern)
- *Sätze variabler Länge*: nur die wirklich benötigte Anzahl von Bytes speichern (Anzahl der Bytes pro Datensatz variabel)

# Zugriffssystem

---

- Zugriffssystem abstrahiert von interner Darstellung der Datensätze auf Seiten
- *logische Datensätze, interne Tupel*
- interne Tupel können Elemente einer Dateidarstellung der konzeptuellen Relation oder Elemente eines Zugriffspfads auf die konzeptuellen Relationen sein
- interne Tupel bestehen aus Feldern (entsprechen Attributen bei konzeptuellen Tupeln)
- Operationen im Zugriffssystem sind typischerweise *Scans* (interne Cursor auf Dateien oder Zugriffspfaden)

# Indexdateien

---

- Zugriffspfad auf eine Datei selbst wieder Datei:  
*Indexdatei*
- Index enthält neben Attributwerten der konzeptuellen Relation, über die ein schneller Zugriff auf die Relation verwirklicht werden soll, eine Liste von Tupeladressen
- zugeordnete Adressen verweisen auf Tupel, die den indizierten Attributwert beinhalten
- Index über Primärschlüssel  $\Rightarrow$  Liste der Tupeladressen einelementig: *Primärindex*
- Index über beliebige andere Attributmenge:  
*Sekundärschlüssel* (obwohl Attributwerte gerade **keine** Schlüsseleigenschaft besitzen müssen)
- Index über Sekundärschlüssel: *Sekundärindex*

# Dateioperationen

---

- Einfügen eines Datensatzes (**insert**)
- Löschen eines Datensatzes (**remove** oder **delete**)
- Modifizieren eines Datensatzes (**modify**)
- Suchen und Finden eines Satzes (**lookup** oder **fetch**)



# Dateioperationen: Arten des lookup

---

- Gegeben Attributwert für bestimmtes Feld, gesucht interne Tupel, die diesen Attributwert besitzen: *single-match query*.
- Gegeben Wertekombination für bestimmte Feldkombination, gesucht alle Tupel, die diese Attributwerte besitzen:
  - ◆ Werte für alle Felder der (Index-)Datei: *exact-match query* (*single match query* ist einfacher Spezialfall)
  - ◆ Werte nur für einige Felder der (Index-)Datei: *partial-match query*
- Gegeben Wertintervall für ein oder mehrere Attribute, gesucht alle internen Tupel, die Attributwerte in diesem Intervall besitzen: *range query*

# Zugriff auf Datensätze

---

- Datensätze in Abhängigkeit vom Primärschlüsselwert in einer Datei
  - ◆ geordnet oder
  - ◆ gehasht (gestreut)gespeichert  $\Rightarrow$  schneller Zugriff über Primärschlüssel
- schneller Zugriff über andere Attributmengen (Sekundärschlüssel) standardmäßig über Indexdateien realisiert

# Dateiorganisationen und Zugriffspfade (I)

---

- *Primärschlüssel / Sekundärschlüssel:*
  - ◆ Primärschlüssel-Zugriff (nur eine Tupeladresse pro Attributwert)
  - ◆ Sekundärschlüssel-Zugriff (mehrere Tupeladressen pro Attributwert möglich)
  - ◆ oft: Primärschlüssel bestimmt Dateiorganisationsform, Sekundärschlüssel bestimmt Zugriffspfade (Indexdateien)
- *eindimensional / mehrdimensional:*
  - ◆ Unterstützung des Zugriffs für feste Feldkombination (*exact-match*)
  - ◆ Unterstützung des Zugriffs für variable Feldkombination (*partial-match*)

# Dateiorganisationen und Zugriffspfade II

---

- *statisch / dynamisch:*

- ◆ statische Dateiorganisationsform oder Zugriffspfad nur optimal bei einer bestimmten Anzahl von zu verwaltenden Datensätzen
- ◆ dynamische Dateiorganisationsformen oder Zugriffspfade unabhängig von der Anzahl der Datensätze (automatische, effiziente Anpassung an wachsende oder schrumpfende Datenmengen)

# Beispiele

---

## ■ B-Baum

- ◆ dynamischer, eindimensionaler Zugriffspfad
- ◆ in den meisten Datenbanksystemen über mehrere Attribute einer Datei definierbar
- ◆ aber nur ein *exact-match* auf dieser Feldkombination möglich

## ■ klassisches Hash-Verfahren

- ◆ statische, eindimensionale Dateiorganisationsform
- ◆ bei wachsenden Tupelmengen immer mehr Kollisionen zu erwarten

# Datensystem (I)

---

- *Optimierung*: mengenorientierte Anfrage (SQL) muß durch System optimiert werden
  - ◆ Umformung des Anfrageausdrucks in einen effizienter zu bearbeitenden Ausdruck (*Query Rewriting, Konzeptuelle oder Logische Optimierung*)
  - ◆ Auswahl des effizientesten Anfrageausdrucks nach Kostenschätzungen (*Kostenbasierte Optimierung*)
  - ◆ *Interne Optimierung*: Auswahl der zur Anfragebearbeitung sinnvollen Zugriffspfade und Auswertungsalgorithmen für jeden relationenalgebraischen Operator

# Datensystem (II)

---

- *ZugriffspfadAuswahl*: bestimmt die internen Strukturen, die bei Abarbeitung einer Anfrage benutzt werden sollen
- *Auswertung*: Wahl der Auswertungsalgorithmen kann Antwortzeit auf eine Anfrage entscheidend beeinflussen; im Zusammenspiel mit der ZugriffspfadAuswahl muß Datensystem die Auswertungsalgorithmen auswählen

# 3. Verwaltung des Hintergrundspeichers

---

- Speichermedien
- Speicherarrays: RAID
- Sicherungsmedien: Tertiärspeicher
- Struktur des Hintergrundspeichers
- Seiten, Sätze und Adressierung
- Pufferverwaltung im Detail
- Kryptographische Methoden



# Speichermedien

---

- verschiedene Zwecke:
  - ◆ Daten zur Verarbeitung bereitstellen
  - ◆ Daten langfristig speichern (und trotzdem schnell verfügbar halten)
  - ◆ Daten sehr langfristig und preiswert archivieren unter Inkaufnahme etwas längerer Zugriffszeiten
- in diesem Abschnitt:
  - ◆ Speicherhierarchie
  - ◆ Magnetplatte
  - ◆ Kapazität, Kosten, Geschwindigkeit

# Speicherhierarchie

---

1. Extrem schneller Prozessor mit Registern
2. Sehr schneller *Cache-Speicher*
3. Schneller *Hauptspeicher*
4. Langsamer *Sekundärspeicher* mit wahlfreiem Zugriff
5. Sehr langsamer *Nearline-Tertiärspeicher* bei dem die Speichermedien automatisch bereitgestellt werden
6. Extrem langsamer *Offline-Tertiärspeicher*, bei dem die Speichermedien per Hand bereitgestellt werden

Tertiärspeicher: CD-R (Compact Disk Recordable), CD-RW (Compact Disk ReWritable), DVD (Digital Versatile Disks), Magnetbänder etwa DLT (Digital Linear Tape)

# Cache-Hierarchie (I)

---

- Eigenschaften der Speicherhierarchie
  - ◆ Ebene  $x$  (etwa Ebene 3, der Hauptspeicher) hat wesentlich schnellere Zugriffszeit als Ebene  $x + 1$  (etwa Ebene 4, der Sekundärspeicher)
  - ◆ aber gleichzeitig einen weitaus höheren Preis pro Speicherplatz
  - ◆ und deshalb eine weitaus geringere Kapazität
  - ◆ Lebensdauer der Daten erhöht sich mit der Höhe der Ebenen

# Cache-Hierarchie (II)

---

- *Zugriffslücke* (Unterschiede zwischen den Zugriffsgeschwindigkeiten auf die Daten) vermindern  
⇒ Cache-Speicher speichern auf Ebene  $x$  Daten von Ebene  $x + 1$  zwischen:
  - ◆ *Cache* (Hauptspeicher-Cache) schnellere Halbleiterspeicher-Technologie für die Bereitstellung von Daten an Prozessor (Ebene 2 in der Speicherhierarchie)
  - ◆ *Plattenspeicher-Cache* im Hauptspeicher: *Puffer*
  - ◆ Cache beim Zugriff auf Daten im WWW über HTTP: Teil des Plattenspeichers, der Teile der im Internet bereitgestellten Daten zwischenspeichert

# Zugriffslücke

---

- Magnetplatten pro Jahr 70% mehr Speicherdichte
- Magnetplatten pro Jahr 7% schneller
- Prozessorleistung pro Jahr um 70% angestiegen
- Zugriffslücke zwischen Hauptspeicher und Magnetplattenspeicher beträgt  $10^5$
- Größen:
  - ◆ *ns* für Nanosekunden (also  $10^{-9}$  Sekunden, *ms* für Millisekunden ( $10^{-3}$  Sekunden))
  - ◆ KB (KiloByte =  $10^3$  Bytes), MB (MegaByte =  $10^6$  Bytes), GB (GigaByte =  $10^9$  Bytes) und TB (TeraByte =  $10^{12}$  Bytes)

# Zugriffslücke in Zahlen (2005)

---

Speicherart	typische Zugriffszeit	typische Kapazität
Cache-Speicher	6 ns	512 KB bis 32 MB
Hauptspeicher	60 ns	32 MB bis 1 GB
— Zugriffslücke $10^5$ —		
Magnetplatten-speicher	12 ms	1 GB bis 10 GB
Platten-Farm oder -Array	12 ms	im TB-Bereich

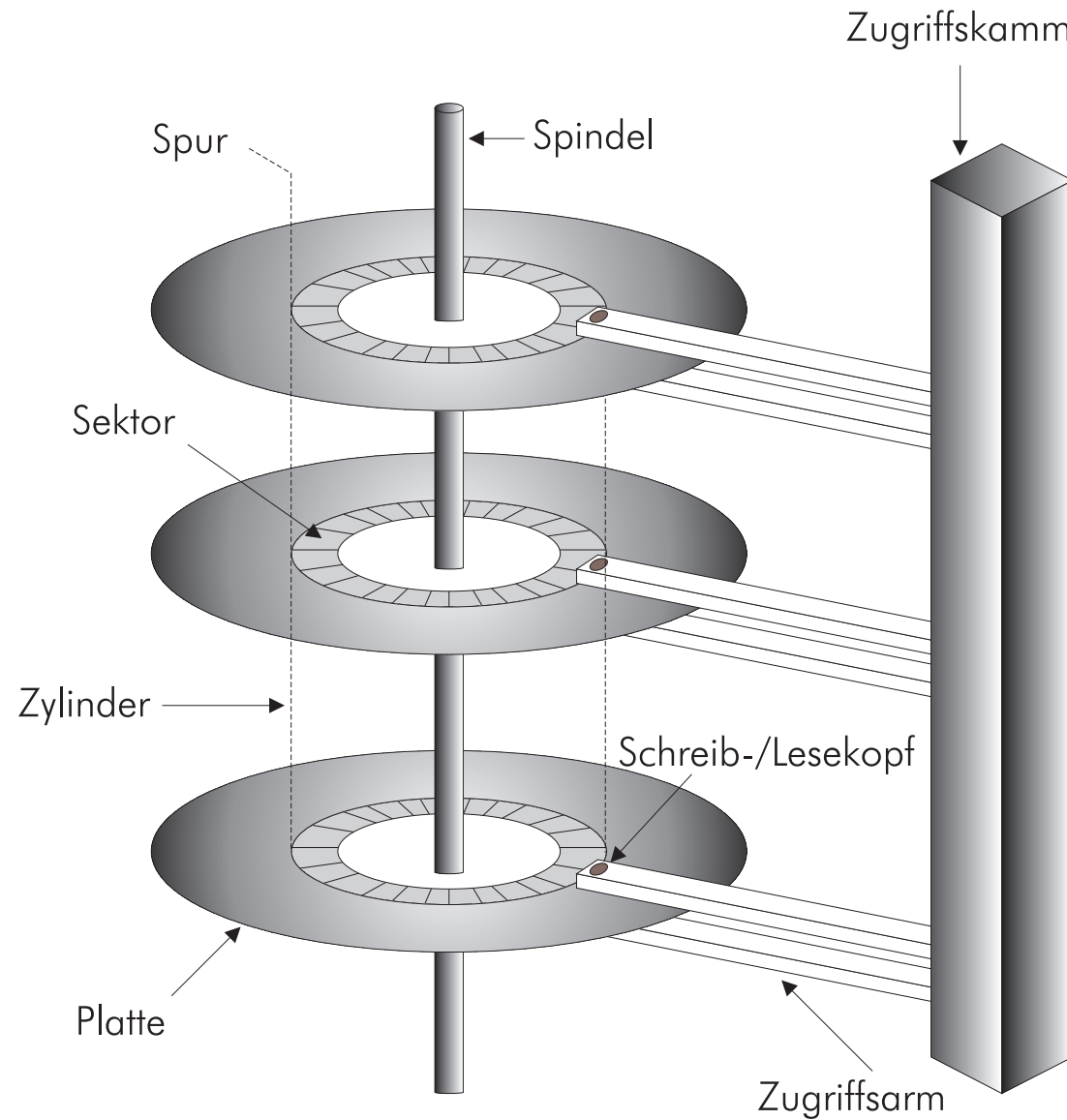
# Lokalität des Zugriffs

---

- Caching-Prinzip funktioniert nicht, wenn immer neue Daten benötigt werden
- in den meisten Anwendungsfällen: *Lokalität* des Zugriffs
- D.h., Großteil der Zugriffe (in den meisten Fällen über 90%) auf Daten aus dem jeweiligen Cache
- Deshalb: Pufferverwaltung des Datenbanksystems wichtiges Konzept

# Die Magnetplatte

---





# Aufzeichnungskomponente

---

- Plattenstapel mit 5 bis 10 Platten
- bis zu 10.000 Umdrehungen pro Minute
- für jede Plattenoberfläche (also etwa 10 bis 20) Schreib-/Lesekopf
- Plattenoberfläche: konzentrische Kreise (*Spuren*)
- übereinander angeordnete Spuren aller Plattenoberflächen: *Zylinder*
- Spuren bestehen aus *Sektoren* (512 Bytes, 1 KB)
- Sektor enthält Daten zur Selbstkorrektur von Fehlern — Paritätsbits oder *Error Correcting Codes* (ECC)
- kleinste Zugriffseinheit auf die Platte: *Block* (Vielfaches der Sektorgröße)

# Positionierungskomponente

---

- Adressierung von Blöcken über Zylinder-, Spur- und Sektornummer
- Zugriffszeit (vom „Auftrag“ bis zur Übertragung des Blockinhaltes):
  - ◆ *Zugriffsbewegungszeit* (seek time)
  - ◆ *Umdrehungswartezeit* (Latenzzeit, latency time)
  - ◆ *Übertragungszeit* (Lesezeit, data-transfer time)

# Typische Merkmale von Magnetplatten

---

<b>Merkmal</b>	<b>1998</b>	<b>1985</b>	<b>1970</b>
mittlere Zugriffsbewegungszeit	8 ms	16 ms	30 ms
Umdrehungszeit	6 ms	16.7 ms	16.7 ms
Spurkapazität	100 KB	47 KB	13 KB
Plattenoberflächen	20	15	19
Zylinder	5000	2655	411
Kapazität	10 GB	1.89 GB	93.7 MB

# Controller und Weiterentwicklungen

---

## Controller

- Übertragungsrate von 40 MB pro Sekunde
- IDE- oder SCSI-Controller

## Weiterentwicklungen

- *Platten-Farmen* lose Kopplung weniger, großer Platten; Betriebssystem oder Datenbanksystem übernimmt selbst die Verteilung der Daten auf die Platten
- *Platten-Arrays*, insb. RAID-Systeme (*Redundant Array of Inexpensive Disks*); billige Standardplatten in hoher Anzahl (8 bis 128) unter einem intelligenten Controller  
⇒ Erhöhung der Fehlertoleranz (Zuverlässigkeit) oder Erhöhung der Parallelität des Zugriffs (Effizienzsteigerung)

# Speicherkapazität und Kosten I

---

Größe	Information oder Medium
1 KB	= 1.000 (genauer: 1024 Byte)
0.5 KB	Buchseite als Text
30 KB	eingescannte, komprimierte Buchseite
1 MB	= 1.000.000 (...)
5 MB	Die Bibel als Text
20 MB	eingescanntes Buch
500 MB	CD-ROM; Oxford English Dictionary
1 GB	= 1.000.000.000 (...)
4.7 GB	Digital Versatile Disk (DVD)
10 GB	große Festplatte
10 GB	komprimierter Spielfilm
100 GB	ein Stockwerk einer Bibliothek
200 GB	Kapazität eines Videobandes

# Speicherkapazität und Kosten II

---

Größe	Information oder Medium
1 TB	= 1.000.000.000.000 (...)
1 TB	Bibliothek mit 1M Bänden
11 TB	größtes Data Warehouse (Wal-Mart)
20 TB	großes Speicher-Array
20 TB	Library of Congress Bände als Text gespeichert
1 PB	= 1.000.000.000.000.000 (...)
1 PB	Eingescannte Bände einer Nationalen Bibliothek
15 PB	weltweite Plattenproduktion in 1996
200 PB	weltweite Magnetbandproduktion in 1996

# Speichermedium und Preis (ca. 2005)

---

Speicherart	Speichermedium	Preis (Euro/MB)
Hauptspeicher	64-MB-SDRAMs	0.5000 Euro
Sekundärspeicher	10-GB-IDE-HD	0.0275 Euro
Tertiärspeicher	opt. Platte 5 GB	0.0160 Euro
	650-MB-CD-RW	0.0150 Euro
	650-MB-CD-R	0.0015 Euro
	70-GB-DLT-Band	0.0010 Euro

# Aktuelle Entwicklungen

---

- Klassische Speicherhierarchie im Umbruch
- Flash-Speicher
  - ◆ bekannt aus Digitalkameras, MP3-Player, PDA, etc.
  - ◆ blockweises Löschen vor jedem Schreiben
    - sequentielles Lesen wie Disk
    - wahlfreies Lesen deutlich schneller
    - Schreiben langsamer
    - Lebensdauer beschränkt auf 100.000 bis 1.000.000 Schreibvorgänge
  - ◆ 256 GB Chips für 40 Dollar vorhergesagt für 2012
    - dann immer noch ca. um Faktor 10 teurer als Disk
- RAM-Speicher
  - ◆ noch schneller, wahlfrei
  - ◆ Hauptspeicherdatenbanken



# Speicherarrays: RAID

---

- Kopplung billiger Standard-Magnetplatten unter einem speziellen Controller zu einem einzigen logischen Laufwerk
- Verteilung der Daten auf die verschiedenen physischen Festplatten übernimmt Controller
- zwei gegensätzliche Ziele:
  - ◆ Erhöhung der Fehlertoleranz (Ausfallsicherheit, Zuverlässigkeit) durch Redundanz
  - ◆ Effizienzsteigerung durch Parallelität des Zugriffs

# Erhöhung der Fehlertoleranz

---

- Nutzung zusätzlicher Platten zur Speicherung von Duplikaten (Spiegeln) der eigentlichen Daten  $\Rightarrow$  bei Fehler: Umschalten auf Spiegelplatte
- bestimmte RAID-Levels (1, 0+1) erlauben eine solche Spiegelung
- Alternative: Kontrollinformationen wie Paritätsbits nicht im selben Sektor wie die Originaldaten, sondern auf einer anderen Platte speichern
- RAID-Levels 2 bis 6 stellen durch Paritätsbits oder Error Correcting Codes (ECC) fehlerhafte Daten wieder her
- ein Paritätsbit kann einen Plattenfehler entdecken und bei Kenntnis der fehlerhaften Platte korrigieren

# Erhöhung der Effizienz (I)

---

- Datenbank auf mehrere Platten verteilen, die parallel angesteuert werden können  $\Rightarrow$  Zugriffszeit verringert sich fast linear mit der Anzahl der verfügbaren Platten
- Verteilung
  - ◆ bitweise (bei 8 Platten kann dann ein Byte auf diese Platten verteilt werden)
  - ◆ byteweise
  - ◆ blockweise

# Erhöhung der Effizienz (II)

---

- höhere RAID-Levels (ab Level 3) verbinden Fehlerkorrektur und block- oder bitweises Verteilen von Daten
- Unterschiede:
  - ◆ *schnellerer Zugriff* auf bestimmte Daten
  - ◆ *höherer Durchsatz* für viele parallel anstehende Transaktionen durch eine Lastbalancierung des Gesamtsystems

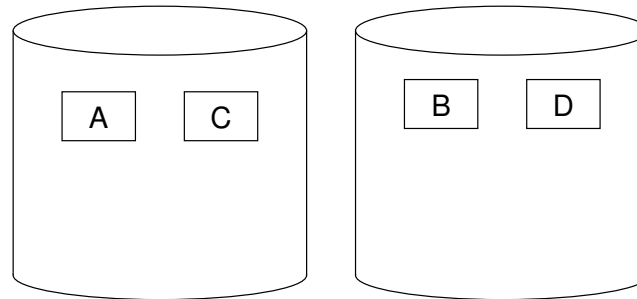
# RAID-Levels 0, 1 und 0+1

---

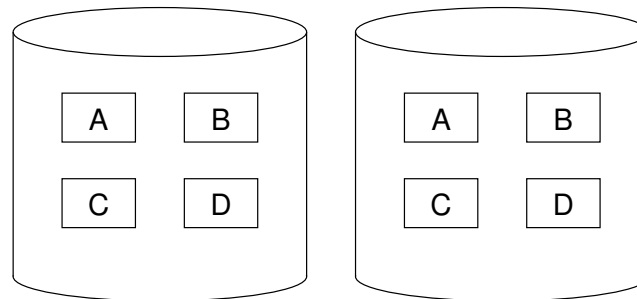
Blöcke



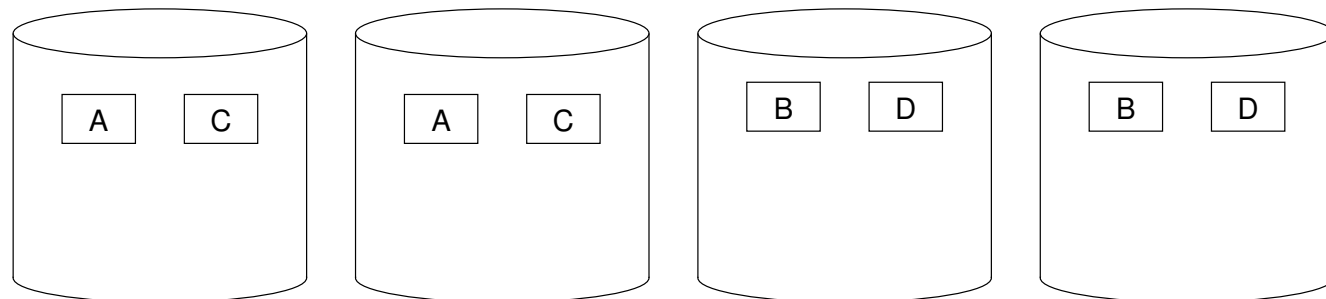
RAID 0



RAID 1



RAID 0 + 1



# RAID-Level 0

---

- Datenblöcke im Rotationsprinzip auf die im RAID-System zur Verfügung stehenden physischen Platten verteilen (*Striping*)
- hier: blockweises Striping
- Vorteile:
  - ◆ Daten aufeinanderfolgender Blöcke parallel lesen
  - ◆ Lastbalancierung bei vielen parallel anstehenden Lesetransaktionen (benötigte Blöcke wahrscheinlich auf verschiedenen Platten)
- Nachteile:
  - ◆ keine Beschleunigung des wahlfreien Zugriffs auf genau einen Block
  - ◆ keinerlei Redundanz oder Kontrollinformation

# RAID-Level 1

---

- zur Speicherung anstehenden Blöcke einfach auf mehrere Platten spiegeln
- Nachteile
  - ◆ einzige Effizienzsteigerung: Lastbalancierung beim Lesen von parallel anstehenden Transaktionen
- Vorteile
  - ◆ bei Fehlern auf eine der anderen Spiegelplatten umschalten
  - ◆ danach fehlerhafte Platte ersetzen und wieder mit den korrekten Daten füllen

# RAID-Level 0+1

---

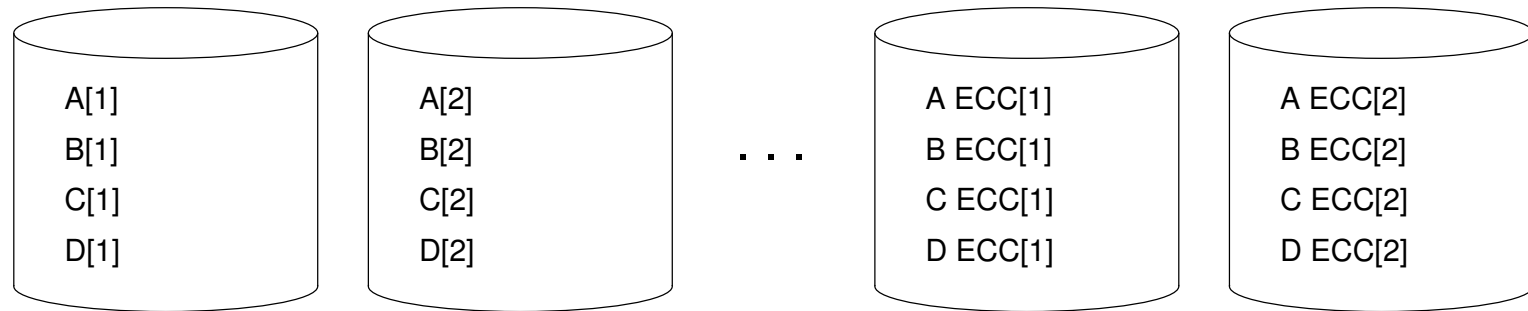
- Kombination von 0 und 1 soll Vorteile der beiden Levels vereinigen
- folgende RAID-Levels: Kombination von Striping zur Effizienzsteigerung und Maßnahmen zur Fehlerkorrektur beibehalten
- jedoch unter Vermeidung der Verdoppelung des Speicherplatzbedarfs
- statt Spiegelplatten nun Kontrollinformationen wie Paritätsbits oder Error Correcting Codes (ECC)



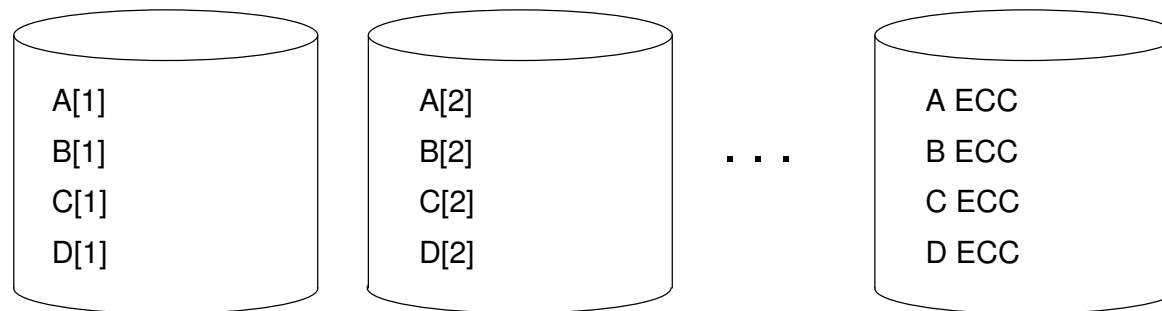
# RAID Levels 2 und 3

---

RAID 2



RAID 3



# RAID-Level 2

---

- Striping auf Bitebene
- Paritätsbits oder erweiterte ECC auf zusätzlichen Platten
- Einlesen benötigt 8 parallele Leseoperationen
- Zugriff nicht effizienter, aber Gesamtdurchsatz verachtfacht

# RAID-Level 3

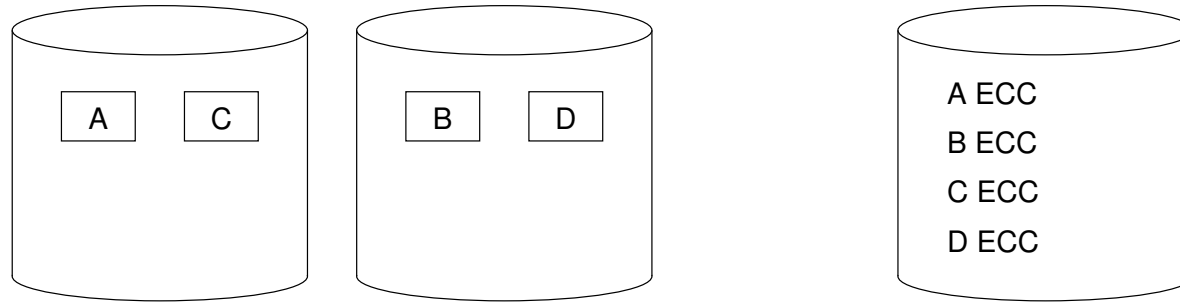
---

- Prinzip: erkennt Platte einen Fehler, so kann ein Byte mit Hilfe der anderen 7 Bits und **einem** Paritätsbit rekonstruiert werden
- Paritätsbits nehmen nur eine einzige, dedizierte Platte ein
- Kontrollinformation auf das wesentliche beschränkt, Plattenplatz weiter reduziert

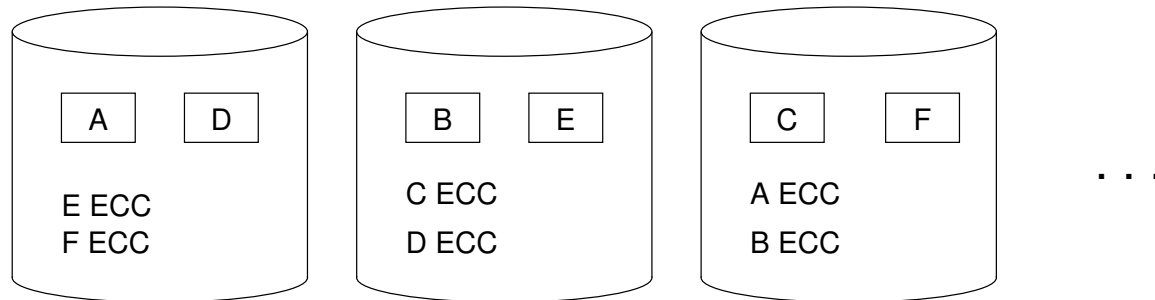
# RAID-Level 4 bis 6

---

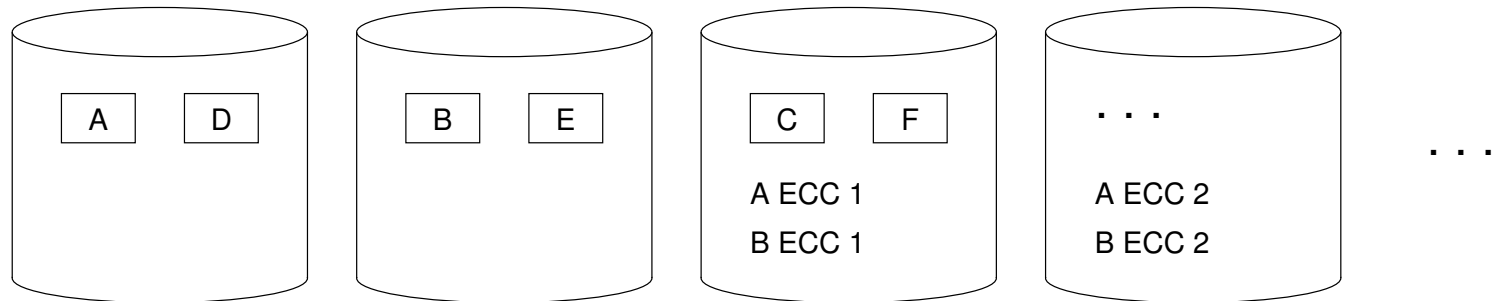
RAID 4



RAID 5



RAID 6



# RAID-Level 4

---

- blockweises Striping
- eine dedizierte Platte zur Aufnahme der Paritätsbits
- kleinere Datenmengen effizienter lesbar, da nur eine physische Platte betroffen ist

# RAID-Level 5

---

- keine zusätzliche, dedizierte Platte zur Speicherung der Paritätsbits
- sondern verteilt Paritätsbits auf die vorhandenen Datenplatten
- damit Flaschenhals von 3 und 4 beseitigt: bisher jedes Schreiben immer auch auf Paritätsplatte  $\Rightarrow$  jede Schreiboperation wartet auf dedizierte Platte
- durch Verwendung unterschiedlicher Platten Last besser verteilt

# RAID-Level 6

---

- baut auf dem RAID-Level 5 auf
- mehr Kontrollinformationen auf Datenplatten
- mehr als ein Plattenfehler korrigierbar
- kaum implementiert

# Übersicht RAID-Levels

---

	0	1	0+1	2	3	4	5	6
Striping blockweise	✓		✓			✓	✓	✓
Striping bitweise				✓	✓			
Kopie		✓	✓					
Parität				✓	✓	✓	✓	✓
Parität dediz. Platte					✓	✓		
Parität verteilt							✓	
Erkennen mehrerer Fehler								✓



# Vergleich der einzelnen RAID-Level (I)

---

## Anwendungsprofil

- Anzahl der Leseoperationen
- Anzahl der Schreiboperationen
- gewünschte Anforderungen an Ausfallsicherheit

# Vergleich der einzelnen RAID-Level (II)

---

## Eigenschaften

- schnellste Fehlerkorrektur: Level 1 (gut für Log-Dateien)
- nur Effizienzsteigerung: Level 0 (gut für Video-Server)
- 3 und 5 Verbesserungen von 2 und 4
- 3 und 5 für Arbeit mit sehr großen Datenmengen
- 3 steigert Gesamtdurchsatz
- 5 verbessert wahlfreien Zugriff
- 6 wäre Verbesserung von 5, aber selten umgesetzt
- Archivmedien werden durch RAID-Systeme nicht (!) ersetzt

# Sicherungsmedien: Tertiärspeicher

---

- weniger oft benutzte Teile der Datenbank, die eventuell sehr großen Umfang haben (Text, Multimedia) „billiger“ speichern als auf Magnetplatten
- aktuell benutzte Datenbestände zusätzlich sichern (archivieren)

Tertiärspeicher: Medium austauschbar

- *offline*: Medien manuell wechseln
- *nearline*: Medien automatisch wechseln (*Jukeboxes*, *Bandroboter*)

# Optische Platten

---

- CD-ROM, CD-R, CD-RW; DVD, DVD-R, DVD-RW, DVD+RW, ...
- Unterschiede: ausführbare Operationen, Speicherkapazität
- Speicherkapazität CD: 650 MB
- Speicherkapazität DVD: 4.7 GB bis 17 GB Daten
- Optische Platten und Laufwerke relativ preiswert
- Zugriff trotz „wahlfreier“ Zugriffsmöglichkeit mit 250 ms sehr langsam
- Lebensdauer mit etwa 30 Jahren vergleichsweise hoch
- als Speicher für selten, aber regelmäßig benötigte Daten beliebt

# Bänder

---

- sehr billiges Medium mit preiswerten Laufwerken
- Zugriff noch langsamer als bei optischen Platten
- sequentieller Zugriff: Transferrate von 1 bis 10 MB pro Sekunde (akzeptabel)
- wahlfreier Zugriff: im Durchschnitt im Minutenbereich
- sehr hohe Kapazität im Gigabyte-Bereich (DLT (Digital Linear Tape) komprimiert 70 GB)
- als Archivspeicher beliebt

# Jukeboxes und Roboter

---

- Herunterfahren / Zurückspulen,
- Medium aus dem Laufwerk holen,
- Medium lagern,
- Neues Medium aufnehmen,
- Laden des Mediums und Hochfahren

kosten über 20 Sekunden bei optischen Platten und über zwei Minuten bei Bändern

- Typische Kapazitäten: Bandroboter mit über 10.000 Cartridges (Petabyte-Bereich möglich)

# Langzeitarchivierung

---

Lebensdauer, Teilaspekte:

- physische Haltbarkeit des Mediums garantiert die *Unversehrtheit* der Daten
- Vorhandensein von Geräten und Treibern garantiert die *Lesbarkeit* von Daten
- zur Verfügung stehende Metadaten garantieren die *Interpretierbarkeit* von Daten
- Vorhandensein von Programmen, die auf den Daten arbeiten können, garantieren die *Wiederverwendbarkeit* von Daten

# Physische Haltbarkeit: Unversehrtheit

---

- 10 Jahre für Magnetbänder, 30 Jahre für optische Platten
- im Vergleich zu den klassischen Archivierungsmedien wie Papier enttäuschend gering
- trotzdem Unversehrtheit der Daten weitaus länger gewährleistet als andere Aspekte



# Lesbarkeit

---

- Welche Systeme können heutzutage noch Bänder, Disketten, Lochstreifen und Lochkarten der sechziger und siebziger Jahre lesen?
- neue Geräte müssen dazu viele Parameter wie Aufzeichnungsformate, Sektorenanordnung, Art der Paritätsinformationen etc. über Jahrzehnte beibehalten
- Verfahren nicht normiert oder Normung in ständiger Weiterentwicklung

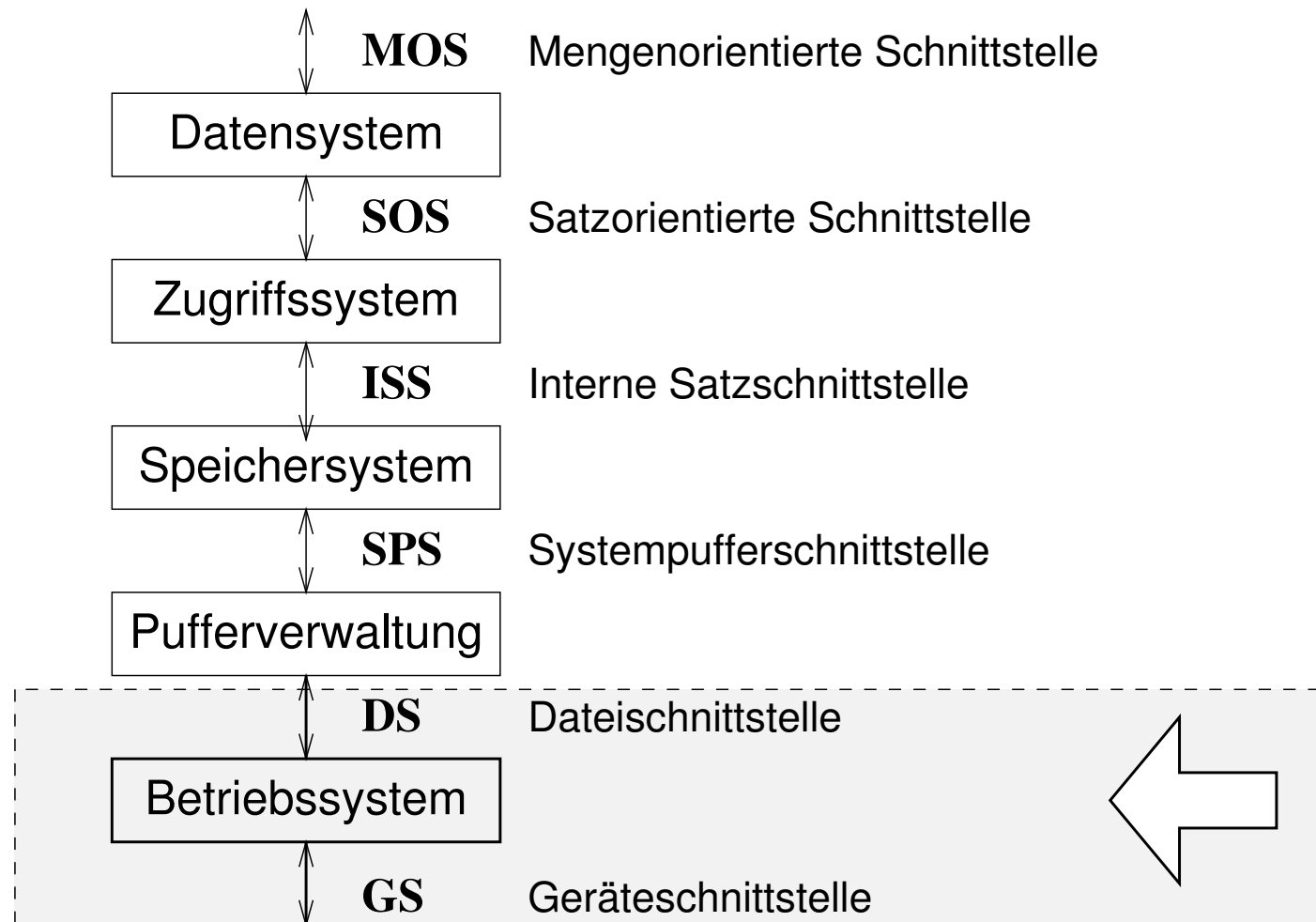
# Interpretierbarkeit

---

- Codes für die Darstellung von Zeichen (EBCDIC, ASCII, 16-Bit-Unicode)
- Dokumentformate: Welche Software kann heute noch formatierte Dokumente lesen, die in einer Textverarbeitungs-Software der siebziger oder frühen achtziger Jahre erstellt wurde?
- Ausweg: Speicherung in offengelegten Standards wie Austauschformaten (ODIF), Seitenbeschreibungssprachen (Postscript) oder Markup-Sprachen (HTML, XML)

# Verwaltung des Hintergrundspeichers

Abstraktion von Speicherungs- oder Sicherungsmediums  
Modell: Folge von Blöcken



# Betriebssystemdateien (I)

---

- jede Relation oder jeder Zugriffspfad in genau einer Betriebssystem-Datei
- ein oder mehrere BS-Dateien, DBS verwaltet Relationen und Zugriffspfade selbst innerhalb dieser Dateien
- DBS steuert selbst Magnetplatte an und arbeitet mit den Blöcken in ihrer Ursprungsform (*raw device*)

# Betriebssystemdateien (II)

---

Warum nicht immer BS-Dateiverwaltung?

- Betriebssystemunabhängigkeit
- In 32-Bit-Betriebssystemen: Dateigröße 4 GB maximal
- BS-Dateien auf maximal einem Medium
- betriebssystemseitige Pufferverwaltung von Blöcken des Sekundärspeichers im Hauptspeicher genügt nicht den Anforderungen des Datenbanksystems

# Blöcke und Seiten

---

- Zuordnung der physischen Blöcke zu *Seiten*
- meist mit festen Faktoren: 1, 2, 4 oder 8 Blöcke einer Spur auf eine Seite
- hier: „ein Block — eine Seite“
- höhere Schichten des DBS adressieren über Seitennummer

# Dienste

---

- Allokation oder Deallokation von Speicherplatz
- Holen oder Speichern von Seiteninhalten
- Allokation möglichst so, daß logisch aufeinanderfolgende Datenbereiche (etwa einer Relation) auch möglichst in aufeinanderfolgenden Blöcken der Platte gespeichert werden
- Nach vielen Update-Operationen:  
*Reorganisationsmethoden*
- *Freispeicherverwaltung*: doppelt verkettete Liste von Seiten

# Abbildung der Datenstrukturen

---

- Abbildung der konzeptuellen Ebene auf interne Datenstrukturen
- Unterstützung durch *Metadaten* (im Data Dictionary, etwa das interne Schema)

Konz. Ebene	Interne Ebene	Dateisystem/Platte
Relationen →	Log. Dateien →	Phys. Dateien
Tupel →	Datensätze →	Seiten/Blöcke
Attributwerte →	Felder →	Bytes



# Varianten der Abbildungen

---

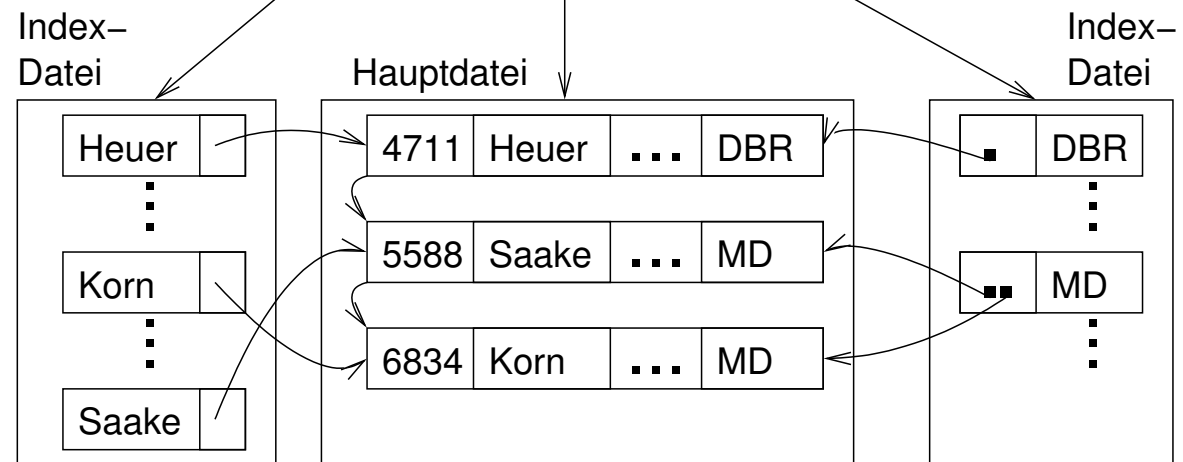
- Beispiel 1: jede Relation in je einer logischen Datei, diese insgesamt in einer einzigen physischen Datei
- Beispiel 2: Cluster-Speicherung – mehrere Relationen in einer logischen Datei

# Übliche Form der Speicherung (I)

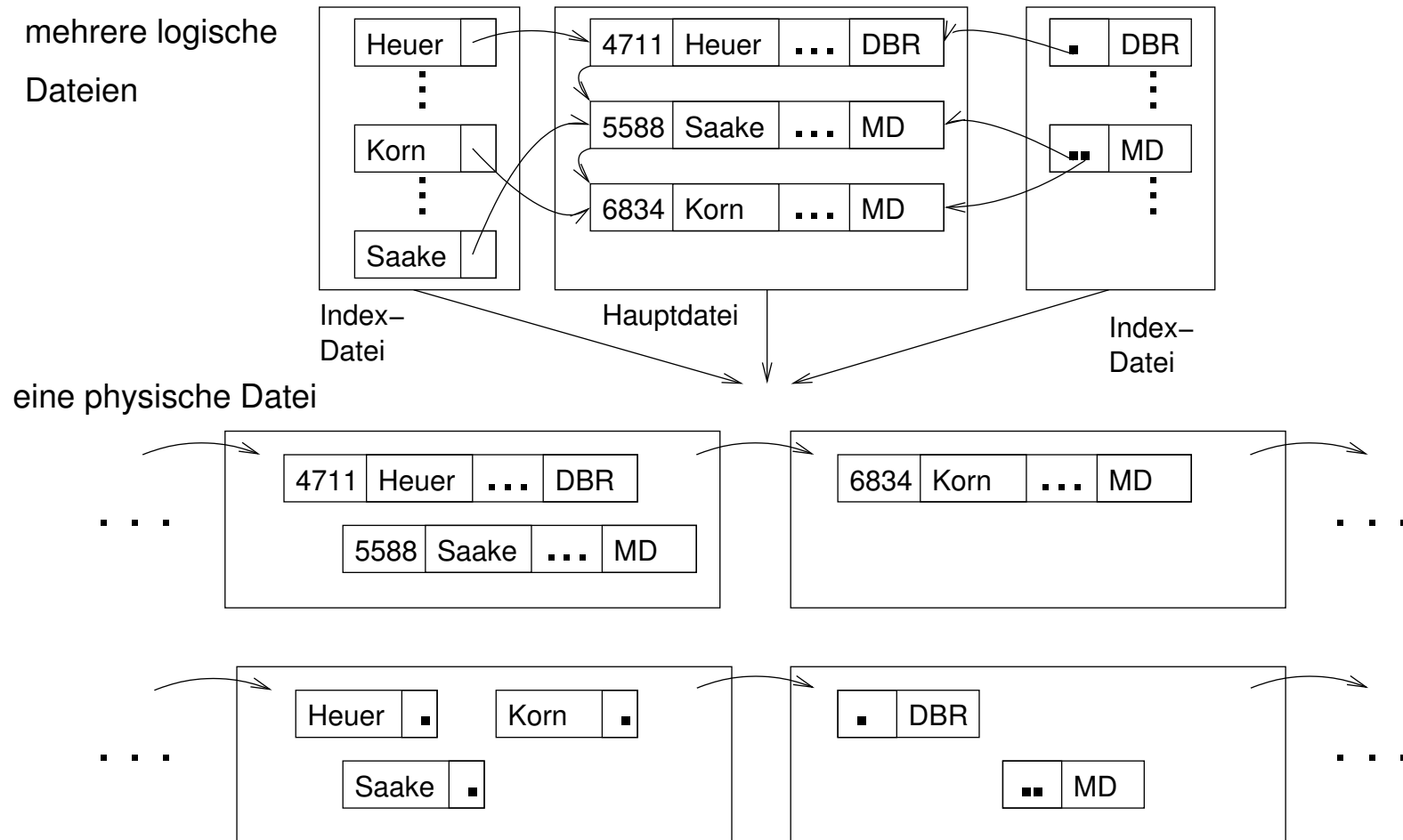
Eine Relation

PANr	Nachname	. . .	Ort
4711	Heuer	. . .	DBR
5588	Saake	. . .	MD
6834	Korn	. . .	MD
⋮	⋮		⋮

mehrere logische  
Dateien



# Übliche Form der Speicherung (II)



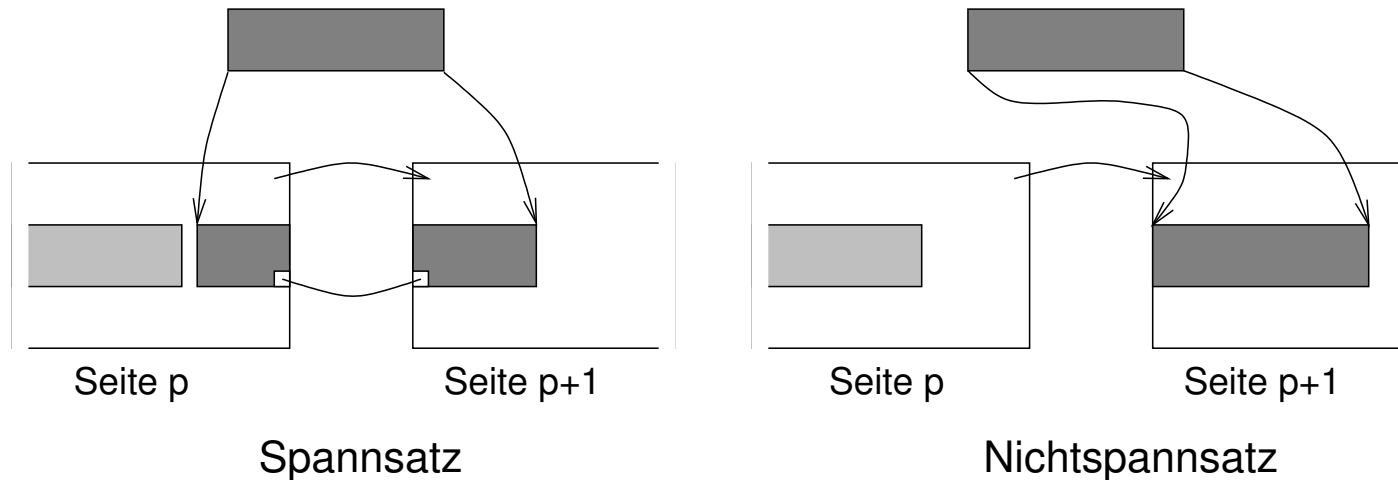
# Einpassen von Datensätzen auf Blöcke

---

- Datensätze (eventuell variabler Länge) in die aus einer fest vorgegebenen Anzahl von Bytes bestehenden Blöcke einpassen: *Blocken*
- Blocken abhängig von variabler oder fester Feldlänge der Datenfelder
  - ◆ Datensätze mit variabler Satzlänge: höherer Verwaltungsaufwand beim Lesen und Schreiben, Satzlänge immer wieder neu ermitteln
  - ◆ Datensätze mit fester Satzlänge: höherer Speicheraufwand

# Blockungstechniken

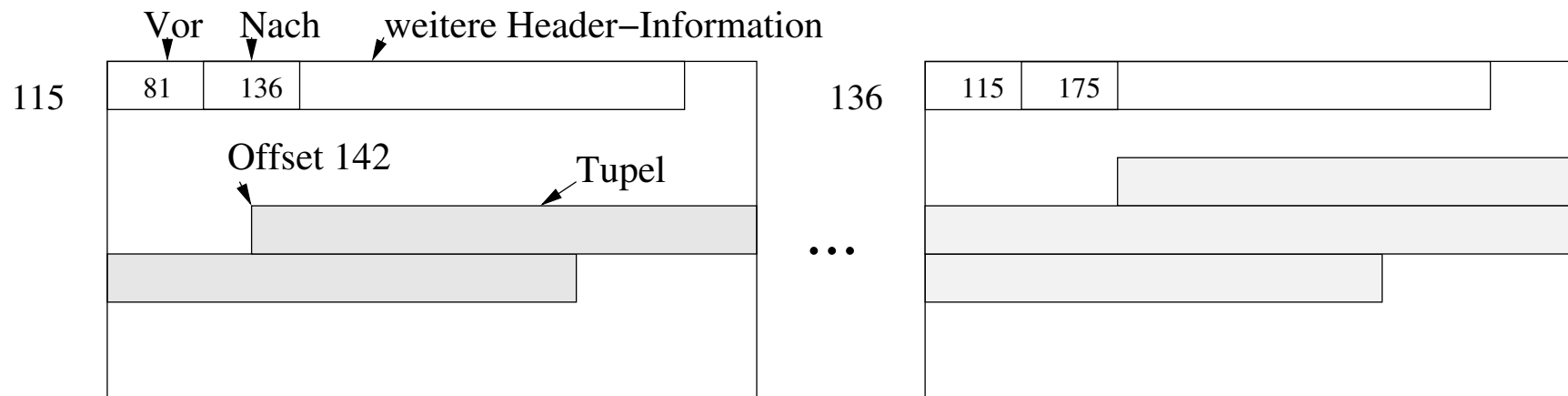
---



- *Nichtspannsätze*: jeder Datensatz in maximal einem Block
- *Spannsätze*: Datensatz eventuell in mehreren Blöcken
- Standard: Nichtspannsätze (nur im Falle von BLOBs oder CLOBs Spannsätze üblich)

# Seiten, Sätze und Adressierung

- Struktur der Seiten: doppelt verkettete Liste
- freie Seiten in Freispeicherverwaltung



- *Header*
  - ◆ Informationen über Vorgänger- und Nachfolger-Seite
  - ◆ eventuell auch Nummer der Seite selbst
  - ◆ Informationen über Typ der Sätze
  - ◆ freier Platz
- *Datensätze*
- un belegte Bytes

# Seite: Adressierung der Datensätze

---

- adressierbare Einheiten
  - ◆ Zylinder
  - ◆ Spuren
  - ◆ Sektoren
  - ◆ Blöcke oder Seiten
  - ◆ Datensätze in Blöcken oder Seiten
  - ◆ Datenfelder in Datensätzen
- Beispiel: Adresse eines Satzes durch Seitennummer und Offset (relative Adresse in Bytes vom Seitenanfang)

(115, 142)



# Seitenzugriff als Flaschenhals

---

- Maß für die Geschwindigkeit von Datenbankoperationen: Anzahl der Seitenzugriffe auf dem Sekundärspeicher (wegen Zugriffslücke)
- Faustregel: Geschwindigkeit des Zugriffs  $\Leftarrow$  Qualität des Zugriffspfad  $\Leftarrow$  Anzahl der benötigten Seitenzugriffe
- Hauptspeicheroperationen nicht beliebig vernachlässigbar

jetzt: Satztypen

# Pinned records (Fixierte Sätze)

---

- *Fixierte Datensätze* sind an ihre Position gebunden
- Bsp.: Verweise mit Zeigern aus anderer, schwer auffindbarer Seite, etwa Verschieben des Datensatzes (115, 142) auf Seite 136
- Gefahr: ins Leere verweisende Zeiger (*dangling pointers*)

# Unpinned records (Unfixierte Sätze)

---

- *unfixierte Sätze*: Verweise mit „logischen Zeigern“
- etwa: Matrikelnummer bei Studenten-Datensatz
- diesen an zentraler Stelle (etwa in Indexdatei zur Studenten-Relation) auf aktuelle Adresse umsetzen
- Nachteil: Verschieben des Datensatzes benötigt neben dem Laden der beiden direkt betroffenen Seite (Quell- und Zielseite der Verschiebeoperation) auch noch Holen der Indexseite
- Nachteil wird beim TID-Konzept (s.u.) behoben

# Sätze fester Länge

---

SQL: Datentypen fester und variabler Länge

- *char(n)* Zeichenkette der festen Länge  $n$
- *varchar(n)* Zeichenkette variabler Länge mit der Maximallänge  $n$

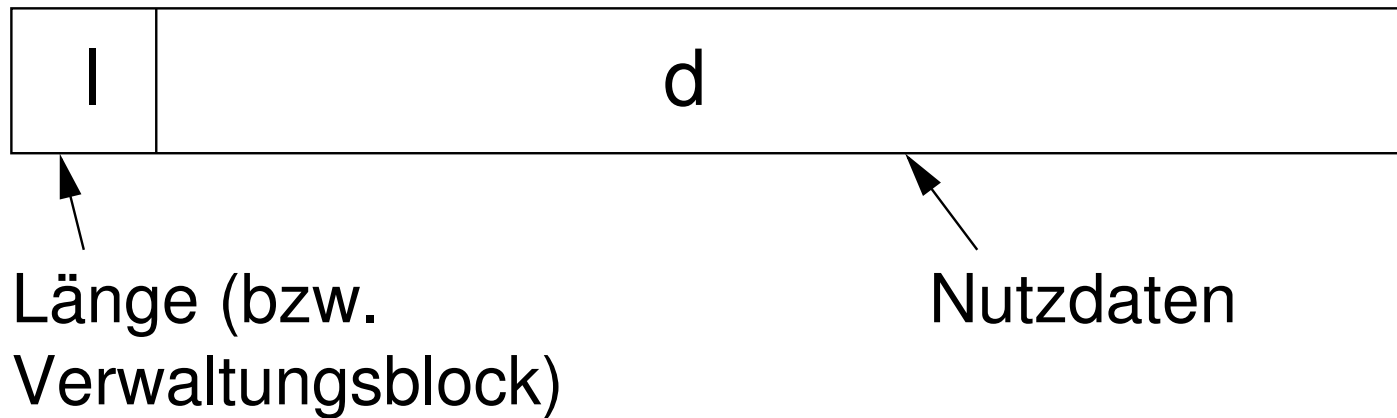
Aufbau der Datensätze, falls alle Datenfelder feste Länge:

1. *Verwaltungsblock* mit
  - Typ eines Satzes (wenn unterschiedliche Satztypen auf einer Seite möglich)
  - Löschrbit
2. *Freiraum* zur Justierung des Offset
3. *Nutzdaten* des Datensatzes

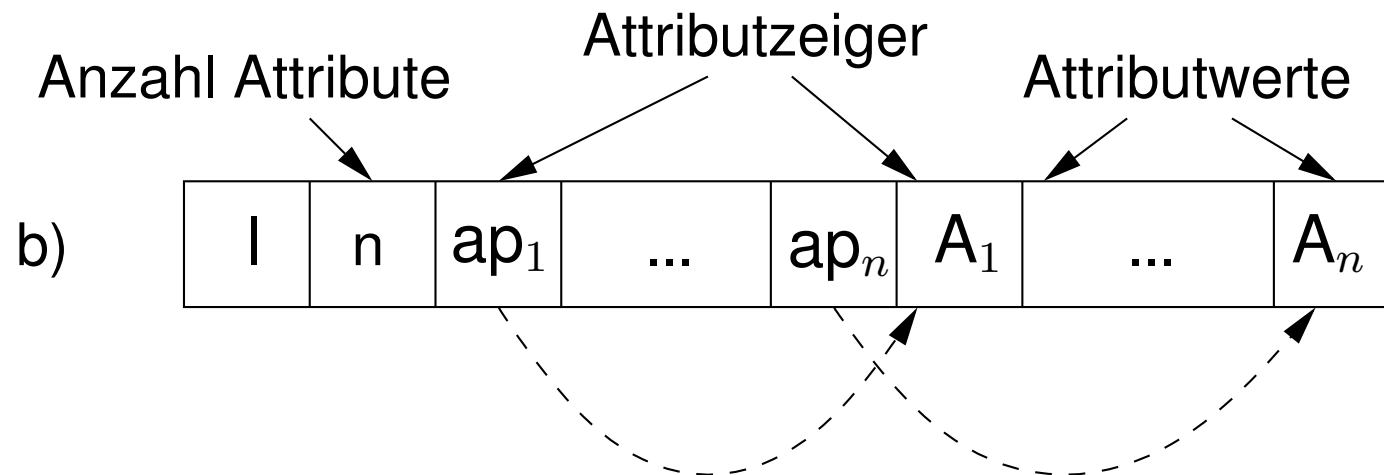
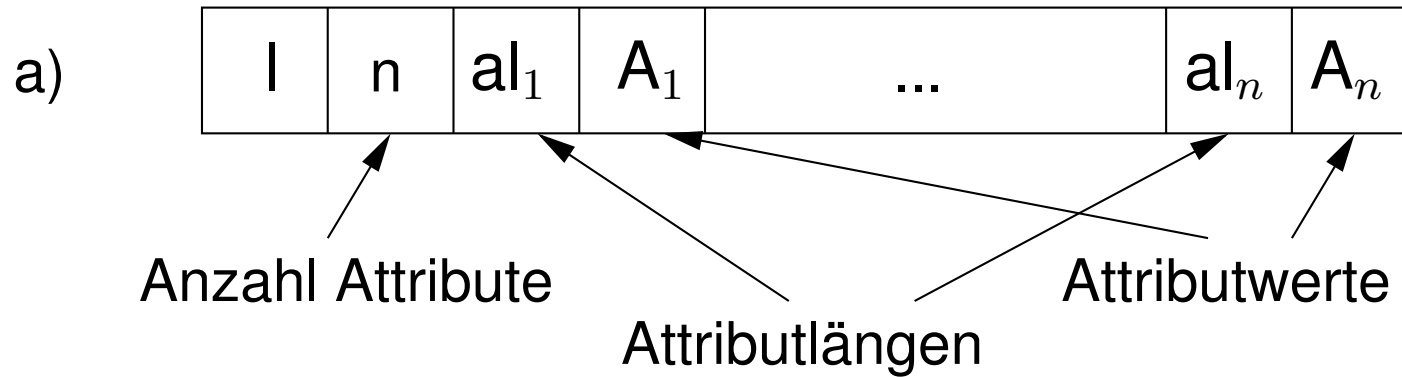
# Sätze variabler Länge

---

- im Verwaltungsblock nötig: Satzlänge  $l$ , um die Länge des Nutzdaten-Bereichs  $d$  zu kennen



# Sätze variabler Länge (II)



# Speicherung von Sätzen variabler Länge

---

- *Strategie a)*: Jedes Datenfeld variabler Länge  $A_i$  beginnt mit einem *Längenzeiger*  $al_i$ , der angibt, wie lang das folgende Datenfeld ist
- *Strategie b)*: Am Beginn des Satzes wird nach dem Satz-Längenzeiger  $l$  und der Anzahl der Attribute ein Zeigerfeld  $ap_1, \dots, ap_n$  für alle variabel langen Datenfelder eingerichtet
- Vorteil Strategie b): leichtere Navigation innerhalb des Satzes (auch für Sätze in Seiten  $\Rightarrow$  TID)

# Anwendung variabel langer Datenfelder

---

„Wiederholgruppen“: Liste von Werten des gleichen Datentyps

- Zeichenketten variabler Länge wie *varchar(n)* sind Wiederholgruppe mit *char* als Basisdatentyp, mathematisch also die Kleene'sche Hülle *(char)\**
- Mengen- oder listenwertige Attributwerte, die im Datensatz selbst denormalisiert gespeichert werden sollen (Speicherung als geschachtelte Relation oder Cluster-Speicherung), bei einer Liste von *integer*-Werten wäre dies *(integer)\**
- Adreßfeld für eine Indexdatei, die zu einem Attributwert auf mehrere Datensätze zeigt (*Sekundärindex*), also *(pointer)\**



# Große unstrukturierte Sätze

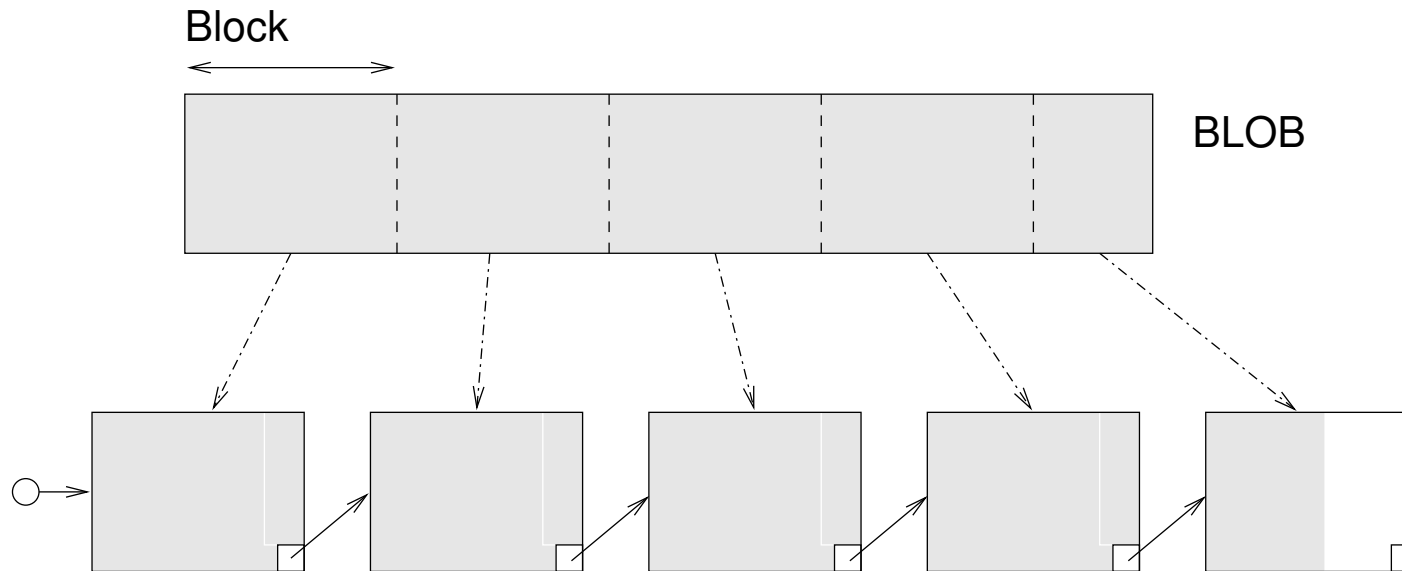
---

- RDBS-Datentypen für sehr große, unstrukturierte Informationen:
  - ◆ *Binary Large Objects (BLOBs)*: Byte-Folgen wie Bilder, Audio- und Videosequenzen
  - ◆ *Character Large Objects (CLOBs)*: Folgen von ASCII-Zeichen (unstrukturierter ASCII-Text)
- lange Felder überschreiten i.a. Grenzen einer Seite, deshalb nur Nicht-BLOB-Felder auf der Originalseite speichern

# BLOB-Speicherung: Lösung 1

---

- Als Attributwert Zeiger: Zeiger zeigt auf Beginn einer Seiten- oder Blockliste, die BLOB aufnimmt



- Vorteil bei Einfügungen, Löschungen, Modifikationen
- Nachteil bei wahlfreiem Zugriff in das BLOB hinein

# BLOB-Speicherung: Lösung 2

---

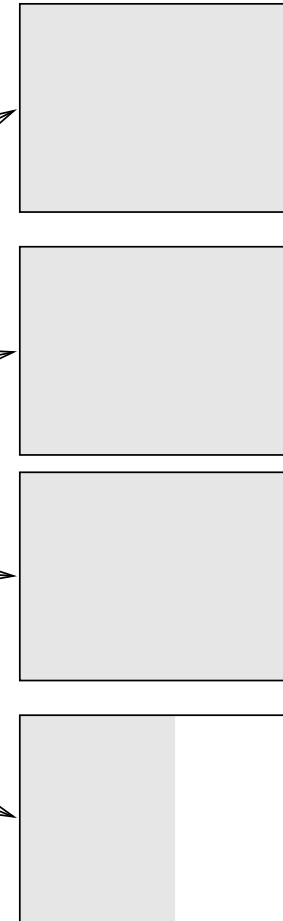
- Als Attributwert BLOB-Directory:
  - ◆ BLOB-Größe
  - ◆ weitere Verwaltungsinformationen
  - ◆ mehrere Zeiger, die auf die einzelnen Seiten verweisen
- Vorteil: schneller Zugriff auf Teilbereiche des BLOBs
- Nachteil: festgelegte, begrenzte Maximalgröße des BLOBs (Gigabyte-BLOB; 8-Byte-Adressierung, Seitengröße 1 KB  $\Rightarrow$  8 MB für ein BLOB-Directory)
- effizienter: B-Baum zur Speicherung von BLOBs (s.u.)

# BLOB-Speicherung: Lösung 2

---

BLOB-Directory

BLOB Größe	
Verwaltungs-Inform.	
Zeiger auf Block 1	
Zeiger auf Block 2	
Zeiger auf Block 3	
...	
Zeiger auf Block k	



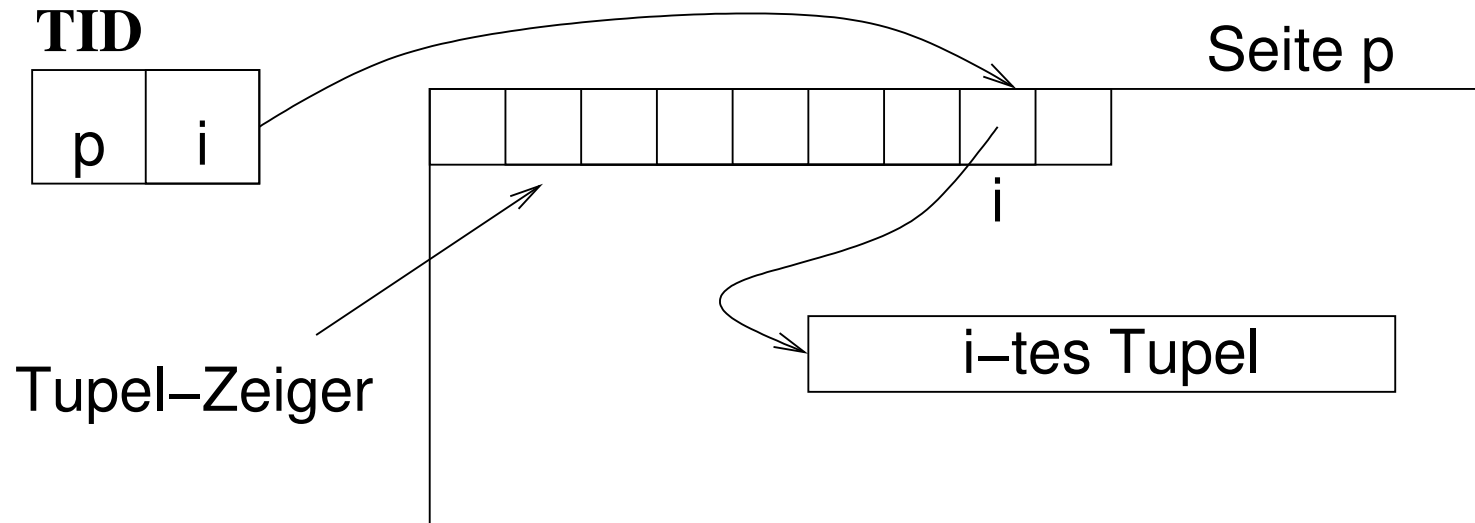
# Adressierung: TID-Konzept

---

- *Tupel-Identifizier* (TID) ist Datensatz-Adresse bestehend aus Seitennummer und Offset
- Offset verweist innerhalb der Seite bei einem Offset-Wert von  $i$  auf den  $i$ -ten Eintrag in einer Liste von *Tupel-Zeigern*, die am Anfang der Seite stehen
- Jeder Tupel-Zeiger enthält Offsetwert
- Verschiebung auf der Seite: sämtliche Verweise von außen bleiben unverändert
- Verschiebungen auf eine andere Seite: statt altem Datensatz neuer TID-Zeiger
- diese zweistufige Referenz aus Effizienzgründen nicht wünschenswert: Reorganisation in regelmäßigen Abständen

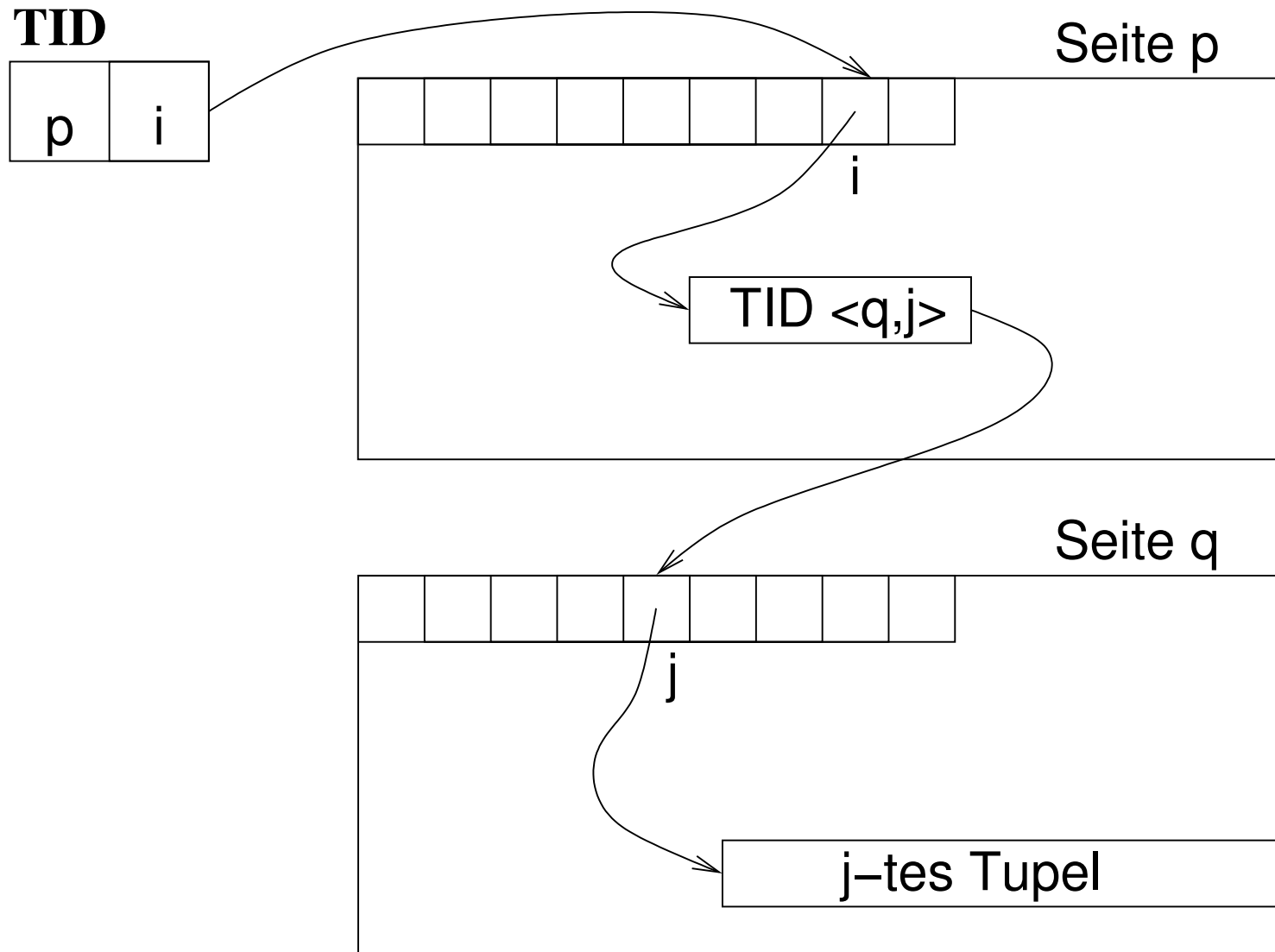
# TID-Konzept: einstufige Referenz

---



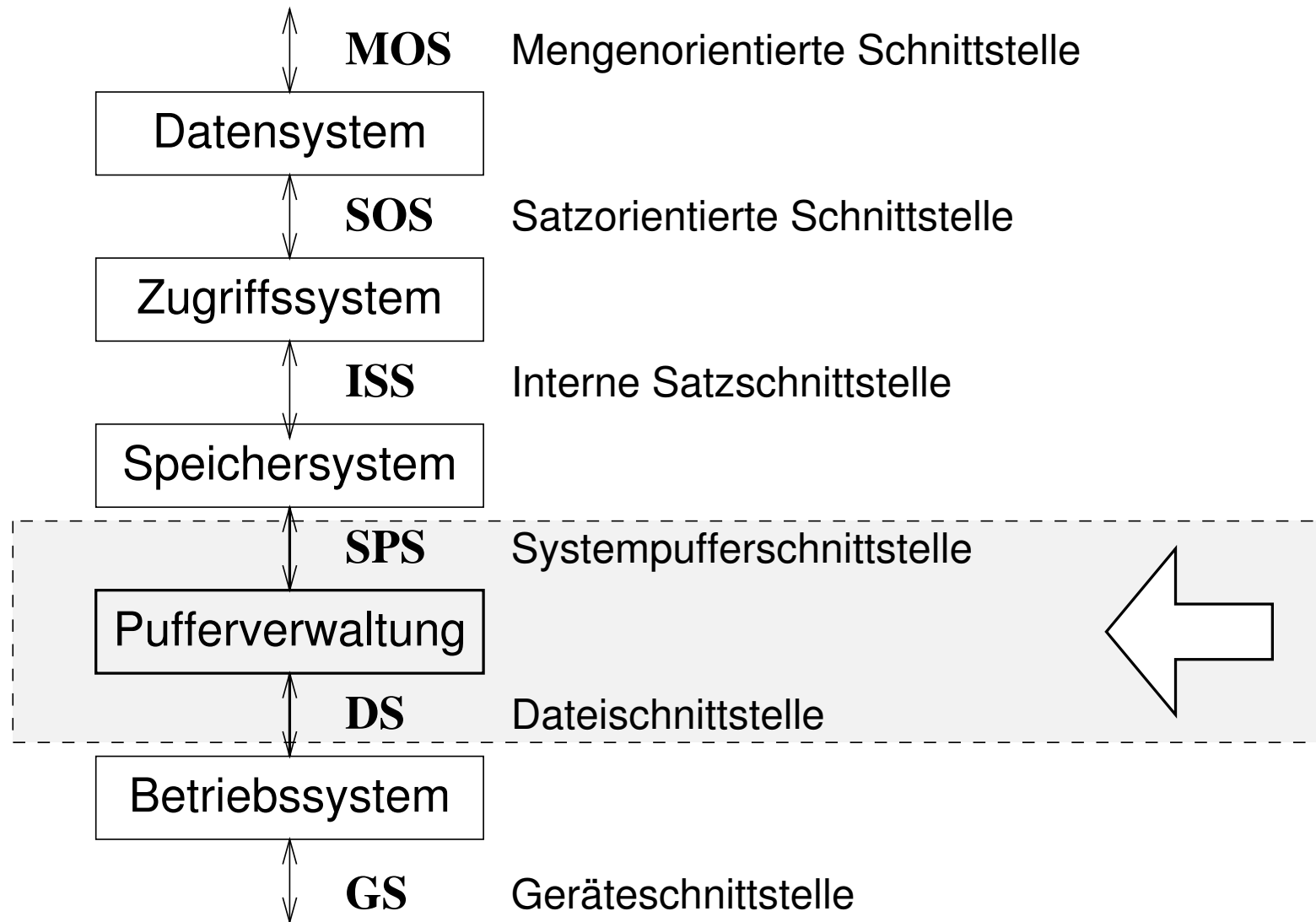
# TID-Konzept: zweistufige Referenz

---



# Pufferverwaltung im Detail

---





# Aufgaben der Pufferverwaltung

---

- *Puffer*: ausgezeichneter Bereich des Hauptspeichers
- in *Pufferrahmen* gegliedert, jeder Pufferrahmen kann Seite der Platte aufnehmen
- Aufgaben:
  - ◆ Pufferverwaltung muß angeforderte Seiten im Puffer suchen  $\Rightarrow$  effiziente *Suchverfahren*
  - ◆ parallele Datenbanktransaktionen: geschickte *Speicherzuteilung* im Puffer
  - ◆ Puffer gefüllt: adäquate *Seitenersetzungsstrategien*
  - ◆ *Unterschiede zwischen einem Betriebssystem-Puffer und einem Datenbank-Puffer*
  - ◆ spezielle Anwendung der Pufferverwaltung: *Schattenspeicherkonzept*

# Suchen einer Seite

---

- *Direkte Suche*: ohne Hilfsmittel linear suchen
- *Indirekte Suche*:
  - ◆ *unsortierte und sortierte Tabelle*: alle Seiten im Puffer vermerkt
  - ◆ *verkettete Liste*: schnelleres sortiertes Einfügen möglich
  - ◆ *Hash-Tabelle*: bei geschickt gewählter Hashfunktion günstigster Such- und Änderungsaufwand

# Speicherzuteilung im Puffer

---

bei mehreren parallel anstehenden Transaktionen

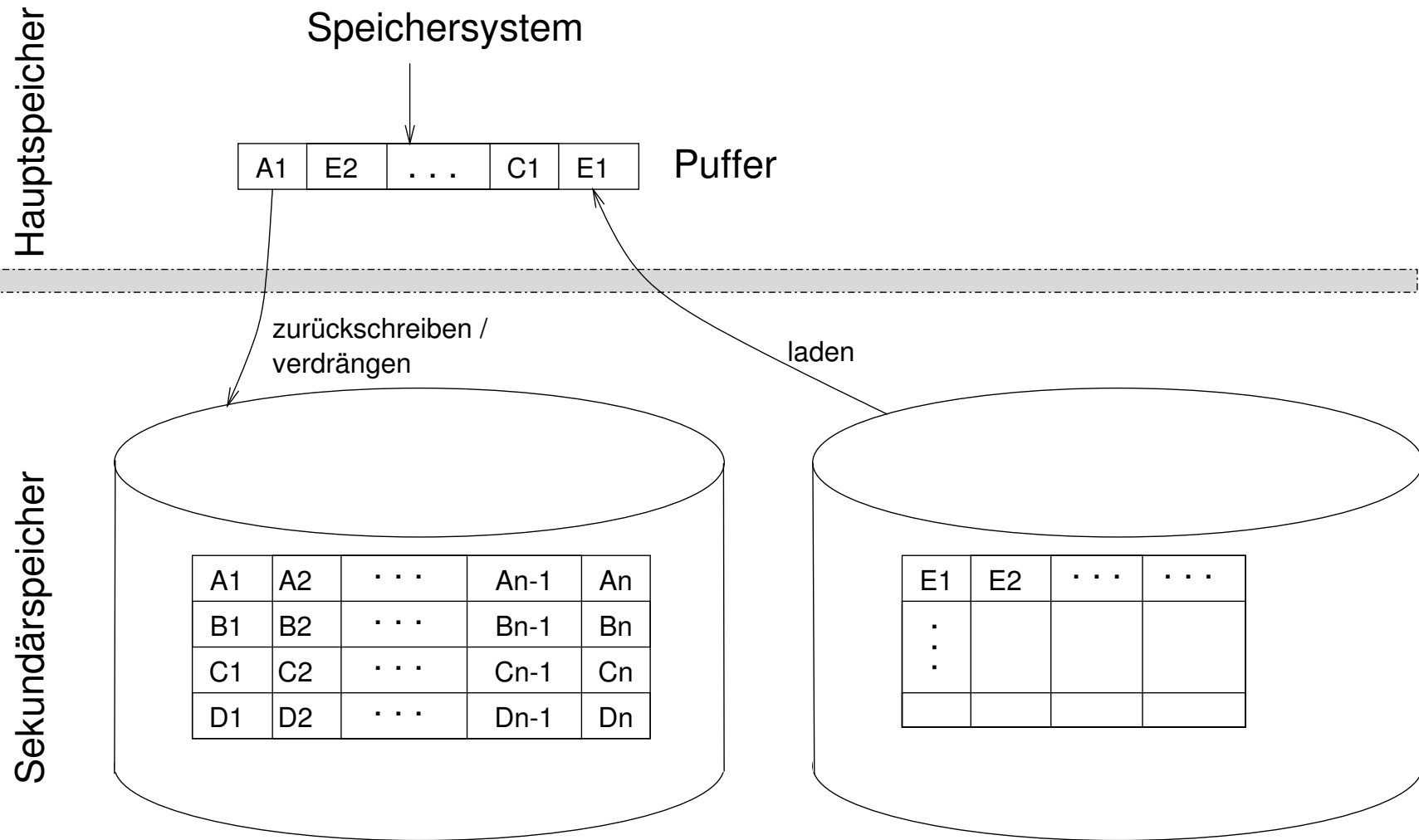
- *Lokale Strategien*: Jeder Transaktion bestimmte disjunkte Pufferteile verfügbar machen (Größe statisch vor Ablauf der Transaktionen oder dynamisch zur Programmlaufzeit entscheiden)
- *Globale Strategien*: Zugriffsverhalten aller Transaktionen insgesamt bestimmt Speicherzuteilung (gemeinsam von mehreren Transaktionen referenzierte Seiten können so besser berücksichtigt werden)
- *Seitentypbezogene Strategien*: Partition des Puffers: Pufferrahmen für Datenseiten, Zugriffspfadseiten, Data-Dictionary-Seiten, ...

# Seitenersetzungsstrategien

---

- Speichersystem fordert Seite  $E_2$  an, die nicht im Puffer vorhanden ist
- Sämtliche Pufferrahmen sind belegt
- vor dem Laden von  $E_2$  Pufferrahmen freimachen
- nach den unten beschriebenen Strategien Seite aussuchen
- Ist Seite in der Zwischenzeit im Puffer verändert worden, so wird sie nun auf Platte *zurückgeschrieben*
- Ist Seite seit Einlagerung in den Puffer nur gelesen worden, so kann sie überschrieben werden (*verdrängt*)

# Seitenersetzung schematisch



# Seitenersetzung: Verfahren (I)

---

- *Demand-paging-Verfahren*: genau eine Seite im Puffer durch angeforderte Seite ersetzen
- *Prepaging-Verfahren*: neben der angeforderten Seite auch weitere Seiten in den Puffer einlesen, die eventuell in der Zukunft benötigt werden (z.B. bei BLOBs sinnvoll)
- *optimale Strategie*: Welche Seite hat maximale Distanz zu ihrem nächsten Gebrauch? (nicht realisierbar, zukünftiges Referenzverhalten nicht vorhersehbar)

~> Realisierbare Verfahren besitzen keine Kenntnisse über das zukünftige Referenzverhalten

- *Zufallsstrategie*: jeder Seite gleiche Wiederbenutzungswahrscheinlichkeit zuordnen

# Seitenersetzung: Verfahren (II)

---

- Gute, realisierbare Verfahren sollen vergangenes Referenzverhalten auf Seiten nutzen, um Erwartungswerte für Wiederbenutzung schätzen zu können
  - ◆ besser als Zufallsstrategie
  - ◆ Annäherung an optimale Strategie

# Merkmale gängiger Strategien

---

- *Alter* der Seite im Puffer:
  - ◆ Alter einer Seite nach Einlagerung (die globale Strategie (G))
  - ◆ Alter einer Seite nach dem letztem Referenzzeitpunkt (die Strategie des jüngsten Verhaltens (J))
  - ◆ Alter einer Seite wird nicht berücksichtigt (—)
- *Anzahl* der Referenzen auf Seite im Puffer:
  - ◆ Anzahl aller Referenzen auf eine Seite (die globale Strategie (G))
  - ◆ Anzahl nur der letzten Referenzen auf eine Seite (die Strategie des jüngsten Verhaltens (J))
  - ◆ Anzahl der Referenzen wird nicht berücksichtigt (—)



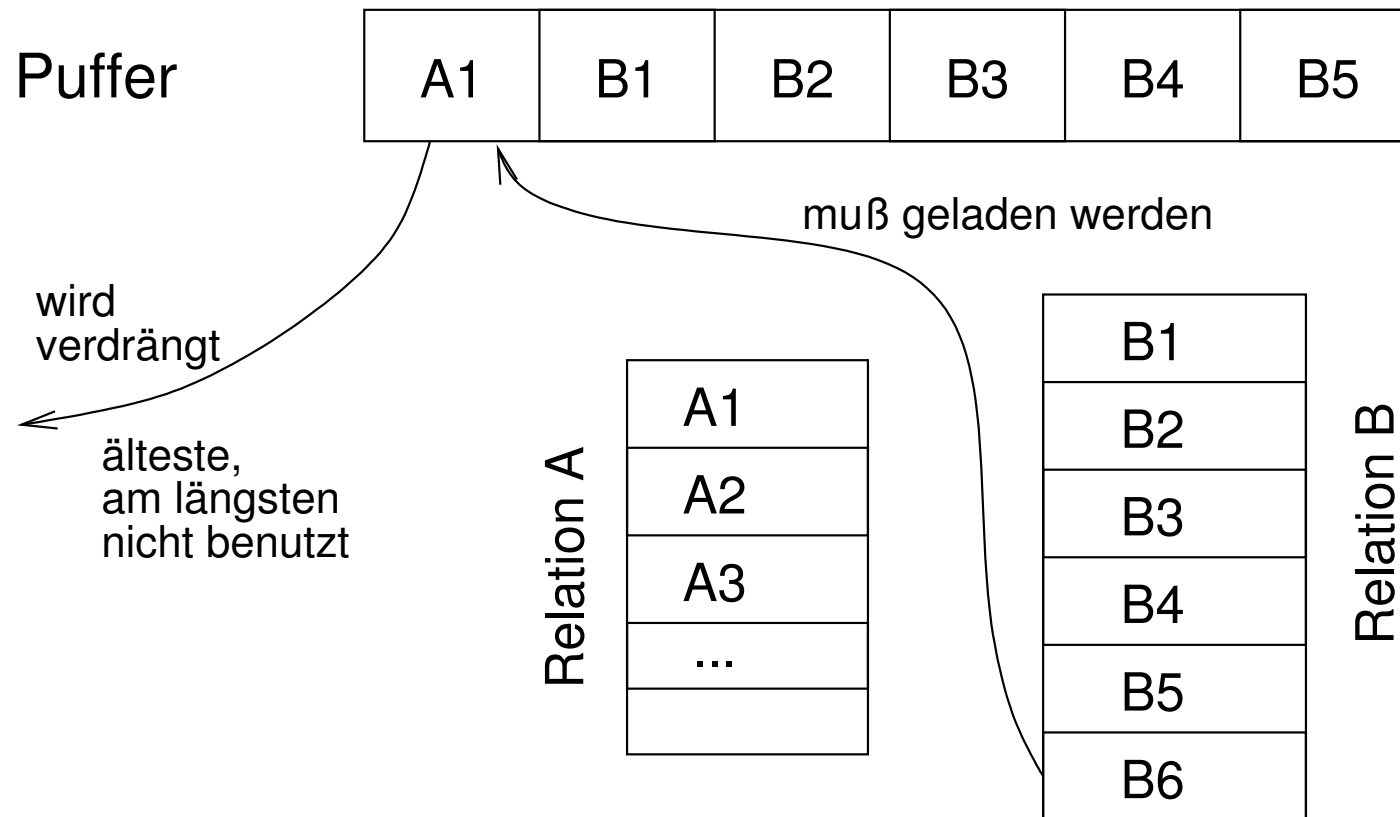
# Klassifikation gängiger Strategien

---

Verfahren	Prinzip	Alter	Anzahl
FIFO	älteste Seite ersetzt	G	–
LFU (least frequently used)	Seite mit geringster Häufigkeit ersetzen	–	G
LRU (least recently used)	Seite ersetzen, die am längsten nicht referenziert wurde (System R)	J	J
DGCLOCK (dyn. generalized clock)	Protokollierung der Ersetzungshäufigkeiten wichtiger Seiten	G	JG
LRD (least reference density)	Ersetzung der Seite mit geringster Referenzdichte	JG	G

# Mangelnde Eignung des BS-Puffers (I)

Natürlicher Verbund von Relationen  $A$  und  $B$  (zugehörige Folge von Seiten:  $A_i$  bzw.  $B_j$ )  $\leadsto$  Implementierung: *Nested-Loop*



# Mangelnde Eignung des BS-Puffers (II)

---

- FIFO:  $A_1$  verdrängt, da älteste Seite im Puffer
- LRU:  $A_1$  verdrängt, da diese Seite nur im ersten Schritt beim Auslesen des Vergleichstupels benötigt wurde

## Problem

- im nächsten Schritt wird gerade  $A_1$  wieder benötigt
- weiteres „Aufschaukeln“: um  $A_1$  laden zu können, muß  $B_1$  entfernt werden (im nächsten Schritt benötigt) usw.

## Seitenersetzungsstrategien von Datenbanksystemen

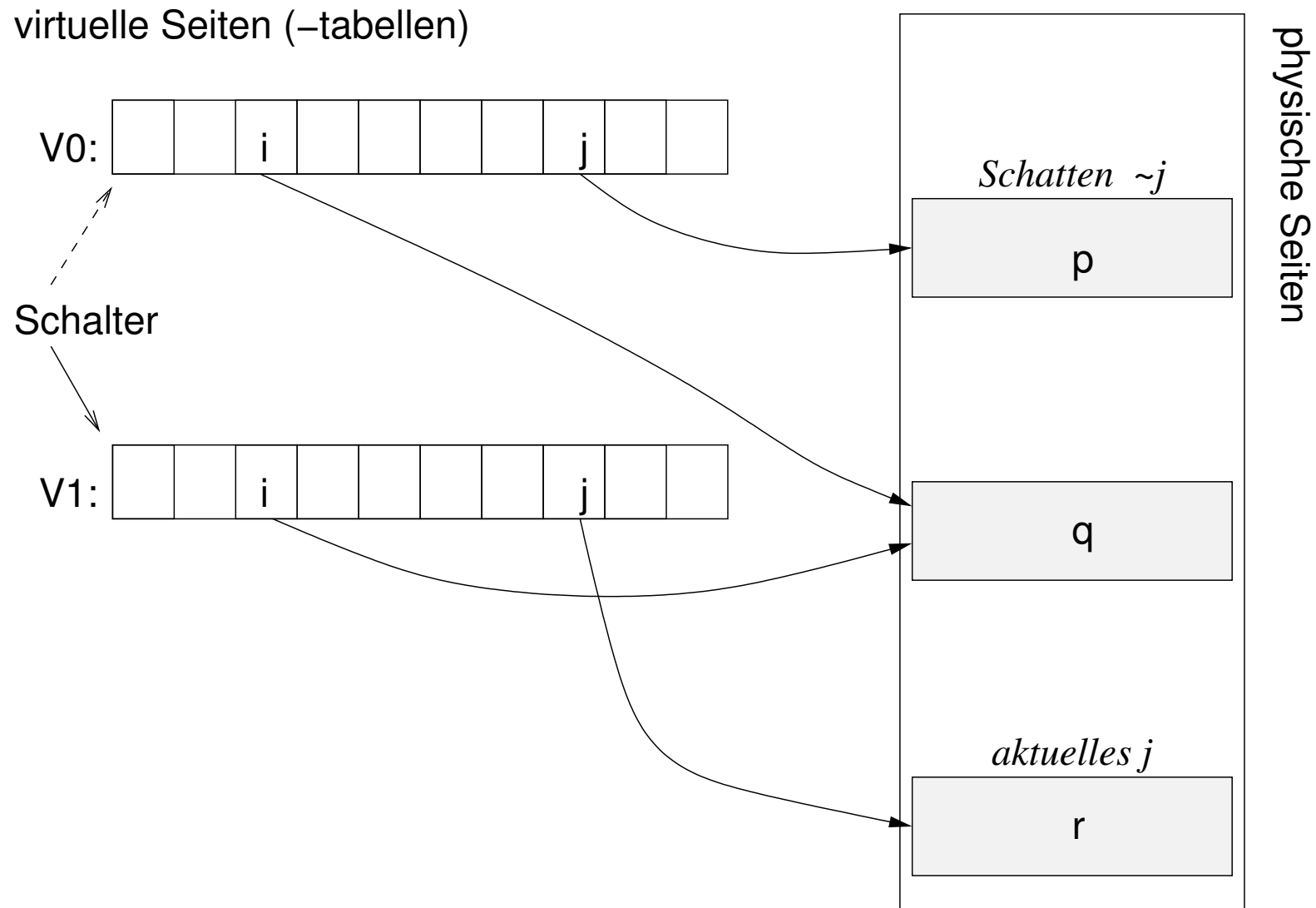
- *Fixieren von Seiten*
- *Freigeben von Seiten*
- *Zurückschreiben einer Seite (z.B. am Transaktionsende!)*

# Schattenspeicherkonzept (I)

---

- Modifikation des Pufferkonzeptes
- Muß Seite im Verlauf einer Transaktion auf Platte zurückgeschrieben werden: nicht auf die Originalseite, sondern auf neue Seite
- statt einer Hilfsstruktur zwei parallele Hilfsstrukturen
- Hilfsstrukturen: *virtuelle Seitentabellen*

# Schattenspeicherkonzept (II)



# Schattenspeicherkonzept (III)

---

- bei Transaktionsabbruch, nur Zugriff auf Ursprungsversion  $V_0$  notwendig, um den alten Zustand der Seiten vor Beginn der Transaktion wiederherzustellen
- Umschalten zwischen den beiden Versionen der virtuellen Seitentabellen durch einen „Schalter“
- Transaktion erfolgreich beendet  $\Rightarrow V_1$  wird Originalversion und  $V_0$  verweist dann auf die neuen Schattenseiten
- Nachteil: ehemals zusammenhängende Seitenbereiche einer Relation werden nach Änderungsoperationen auf der Datenbank über den Sekundärspeicher „verstreut“

# Kryptographische Verfahren

---

- unerlaubten Zugriff auf Datenbank verhindern
  - ◆ im DBS: Rechtevergabe durch **grant**
  - ◆ auf BS-Ebene: Datei verschlüsseln gegen „Dump“
  - ◆ Netzwerk: Datei verschlüsseln, falls Daten übertragen werden, sichere Kanäle (SSL)
- übliche Verfahren
  - ◆ Data Encryption Standard
  - ◆ Public-Key-Verfahren: RSA, PGP

## 4. Dateiorganisation / Zugriffsstrukturen

---

- Klassifikation der Speichertechniken
- Statische Verfahren (Heap, indexsequentiell, indiziert-nichtsequentiell)
- Baumverfahren (B-Bäume und Varianten)
- Hashverfahren
- Bitmap-Indexe
- Mehrdimensionale Speichertechniken
- Cluster-Bildung
- Physische Datendefinition und Umsetzung in SQL-Systemen
- Datenkatalog / Data Dictionary



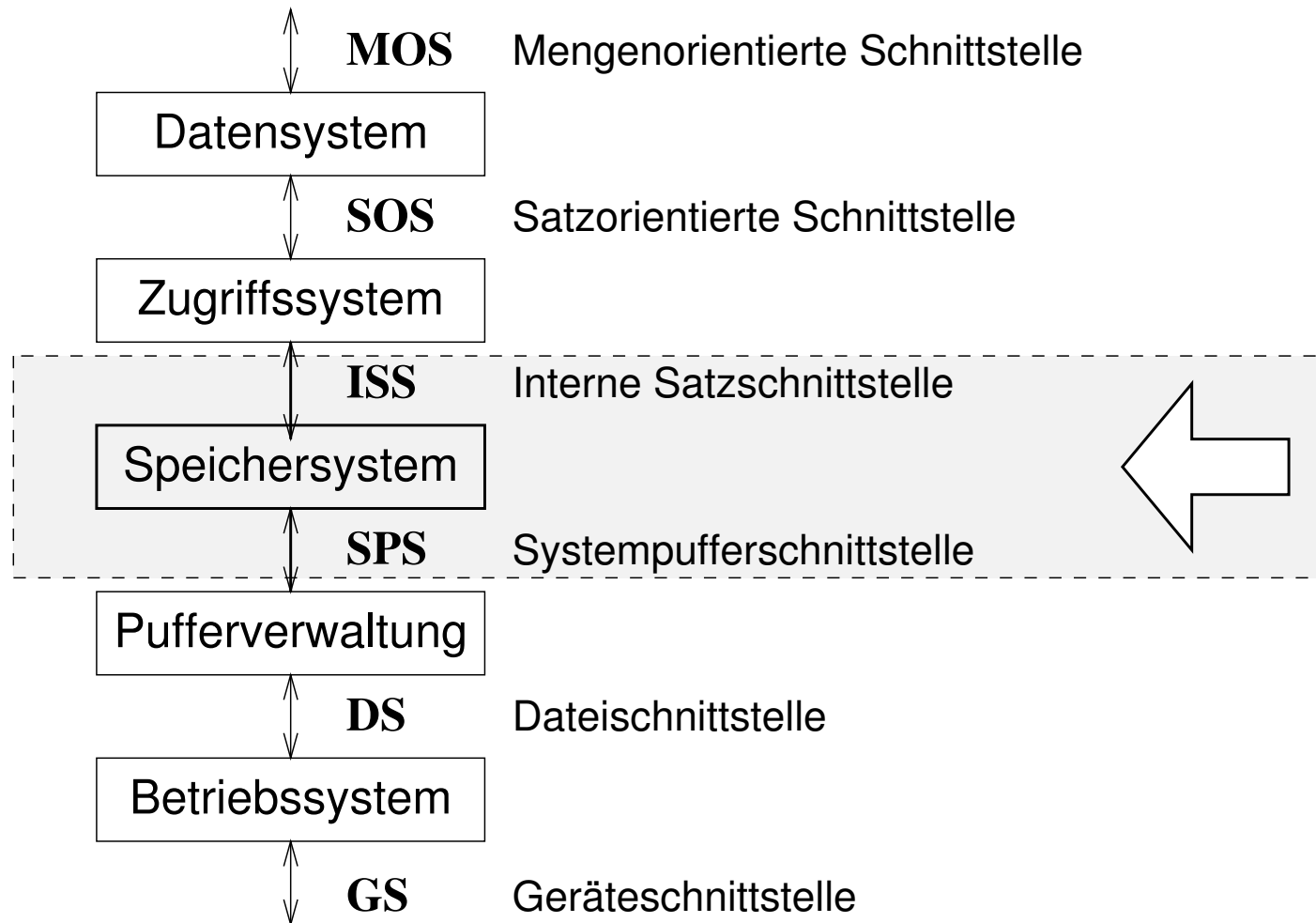
# Einordnung in 5-Schichten-Architektur

---

- *Speichersystem* fordert über Systempufferschnittstelle Seiten an
- interpretiert diese als *interne Datensätze*
- interne Realisierung der logischen Datensätze mit Hilfe von Zeigern, speziellen Indexeinträgen und weiteren Hilfsstrukturen
- *Zugriffssystem* abstrahiert von der konkreten Realisierung

# Einordnung (II)

---



# Klassifikation der Speichertechniken

---

Kriterien für *Zugriffsstrukturen* oder *Zugriffsverfahren*:

- organisiert interne Relation selbst (*Dateiorganisationsform*) oder zusätzliche Zugriffsmöglichkeit auf bestehende interne Relation (*Zugriffspfad*)
- Art der Zuordnung von gegebenen Attributwerten zu Datensatz-Adressen
- Arten von Anfragen, die durch Dateiorganisationsformen und Zugriffspfade effizient unterstützt werden können

# Primärschlüssel vs. Sekundärschlüssel

---

Unterscheidungsmerkmal: Art des/der unterstützten Attribute(s)

- *Primärschlüssel*: wesentliche Eigenschaften
  - ◆ Duplikatfreiheit
  - ◆ identifizierende Attributmenge
  - ◆ Verbundoperationen oft über Primärschlüssel
- *Sekundärschlüssel* beliebige andere Attributmengen
  - ◆ meist keine Schlüsseleigenschaft
  - ◆ meist kein identifizierendes Merkmal
  - ◆ Unterstützung bestimmter Anfragen (etwa Selektionen über Sekundärschlüsseln)

# Weitere Unterscheidungsmerkmale

---

- Normalfall: Primärschlüssel über *Primärindex* oder *Dateiorganisationsform* unterstützt, kann *geclusterter Index* sein
- Normalfall: Sekundärschlüssel über *Sekundärindex* oder *Zugriffspfad* unterstützt, Index ist nicht *geclustert*
- *dünnbesetzter Index* eignet sich nur für Primärschlüssel
- *dichtbesetzter Index* muß bei Sekundärschlüssel eingesetzt werden

# Primärindex vs. Sekundärindex

---

- *Primärindex*: Zugriffspfad auf interne Relation, der Dateiorganisationsform der internen Relation ausnutzen kann
- interne Relation  $\Rightarrow$  Primärindex kann Sortierung ausnutzen  $\Rightarrow$  *geclusteter Index* oder *dünnbesetzter Index* möglich
- Primärindex im Normalfall über Primärschlüssel, aber auch über Sekundärschlüssel denkbar
- *Sekundärindex*: jeder weitere Zugriffspfad auf interne Relation, der Dateiorganisationsform der internen Relation nicht ausnutzen kann
- Sekundärindex: *nicht-geclusteter Index* oder *dichtbesetzter Index*

# Primärindex vs. Sekundärindex (II)

---

- pro interne Relation ein Primärindex, mehrere Sekundärindexe
- einige RDBS: Sekundärindexe als Zugriffspfad  $\Rightarrow$  kein Index nutzt Art der Speicherung der internen Relation aus

# Dateiorganisation vs. Zugriffspfad (I)

---

- *Dateiorganisationsform*: Form der Speicherung der internen Relation
  - ◆ unsortierte Speicherung von internen Tupeln: *Heap-Organisation*
  - ◆ sortierte Speicherung von internen Tupeln: *sequentielle Organisation*
  - ◆ gestreute Speicherung von internen Tupeln: *Hash-Organisation*
  - ◆ Speicherung von internen Tupeln in mehrdimensionalen Räumen: *mehrdimensionale Dateiorganisationsformen*
- üblich: Sortierung oder Hash-funktion über Primärschlüssel
- sortierte Speicherung plus zusätzlicher Primärindex über Sortierattributen: *index-sequentielle Organisationsform*



# Dateiorganisation vs. Zugriffspfad (II)

---

*Zugriffspfad*: über grundlegende Dateiorganisationsform hinausgehende Zugriffsstruktur, etwa Indexdatei

- Einträge  $(K, K \uparrow)$
- $K$  der Wert eines Primär- oder Sekundärschlüssels
- $K \uparrow$  Datensatz oder Verweis auf Datensatz
- $K$ : *Suchschlüssel*, genauer: *Zugriffsattribute* und *Zugriffsattributwerte*

# Dateiorganisation vs. Zugriffspfad (III)

---

Eintrag  $K \uparrow$ :

- $K \uparrow$  ist *Datensatz selbst*: Zugriffspfad wird Dateiorganisationsform
- $K \uparrow$  ist *Adresse eines internen Tupels*: Primärschlüssel; Sekundärschlüssel mit  $(K, K \uparrow_1), \dots, (K, K \uparrow_n)$  für denselben Zugriffsattributwert  $K$
- $K \uparrow$  ist *Liste von Tupeladressen*: Sekundärschlüssel; nachteilig ist variable Länge der Indexeinträge

Tupeladressen: TIDs, nur Seitenadressen, ...

Indexdatei kann selbst in Dateiorganisationsform gespeichert und mit Zugriffspfaden versehen werden

# Dünn- vs. dichtbesetzter Index

---

- *dünnbesetzter Index*: nicht für jeden Zugriffsattributwert  $K$  ein Eintrag in Indexdatei
  - ◆ interne Relation sortiert nach Zugriffsattributen: im Index reicht ein Eintrag pro Seite  $\Rightarrow$  Index verweist mit  $(K_1, K_1 \uparrow)$  auf *Seitenanführer*, nächste Indexeintrag  $(K_2, K_2 \uparrow)$
  - ◆ Datensatz mit Zugriffsattributwert  $K_?$  mit  $K_1 \leq K_? < K_2$  ist auf Seite von  $K_1 \uparrow$  zu finden
- *indexsequentielle Datei*: sortierte Datei mit dünnbesetztem Index als Primärindex

# Dünn- vs. dichtbesetzter Index (II)

---

- *dichtbesetzter Index*: für jeden Datensatz der internen Relation ein Eintrag in Indexdatei
- Primärindex kann dichtbesetzter Index sein, wenn Dateiorganisationsform Heap-Datei, aber auch bei Sortierung (*geclusterter Index*)

# Geclusterter vs. nicht-geclusterter Index

---

- *geclusterter Index*: in der gleichen Form sortiert wie interne Relation
- Bsp.: interne Relation `Studenten` Matrikelnummern sortiert  $\Rightarrow$  Indexdatei über dem Attribut `Matrikelnummer` üblicherweise geclustert

# Geclusterter vs. nicht-geclusterter Index

---

- *nicht-geclusterter Index*: anders organisiert als interne Relation
- Bsp.: über `Studienfach` ein Sekundärindex, Datei selbst nach Matrikelnummern sortiert
- Primärindex dünnbesetzt und geclustert sein
- jeder dünnbesetzte Index ist auch geclusterter Index, aber nicht umgekehrt
- Sekundärindex kann nur dichtbesetzter, nicht-geclusterter Index sein (auch: invertierte Datei)

# Schlüsselzugriff vs. -transformation

---

- *Schlüsselzugriff*: Zuordnung von Primär- oder Sekundärschlüsselwerten zu Adressen in Hilfsstruktur wie Indexdatei
  - ◆ Bsp.: indexsequentielle Organisation, B-Baum, KdB-Baum, ...
- *Schlüsseltransformation*: berechnet Tupeladresse aufgrund Formel aus Primär- oder Sekundärschlüsselwerten (statt Indexeinträgen nur Berechnungsvorschrift gespeichert)
  - ◆ Bsp.: *Hash-Verfahren*

# Ein-Attribut- vs. Mehr-Attribut-Index

---

- *Ein-Attribut-Index (non-composite index)*: Zugriffspfad über einem einzigen Zugriffsattribut
- *Mehr-Attribut-Index (composite index)*: Zugriffspfad über mehreren Attributen
- Bsp.: Attribute `VORNAME` und `PLZ` unterstützen
  - ◆ entweder zwei Ein-Attribut-Indexe
  - ◆ oder ein Zwei-Attribut-Index über beiden Attributen
- Vorteil Mehr-Attribut-Index: bei *exact-match* nur ein Indexzugriff (weniger Seitenzugriffe)
- Mehr-Attribut-Index: Ausführungsart bestimmt, ob neben *exact-match* auch noch *partial-match* effizient unterstützt wird (eindimensional oder mehrdimensional)



# Ein- vs. mehrdim. Zugriffsstruktur

---

- Ein-Attribut-Index immer *eindimensionale Zugriffsstruktur*: Zugriffsattributwerte definieren lineare Ordnung in eindimensionalem Raum
- Mehr-Attribut-Index ist eindimensionale oder *mehrdimensionale Zugriffsstruktur*
  - ◆ *eindimensionaler Fall*: Kombinationen der verschiedenen Zugriffsattributwerte konkateniert als einen einzigen Zugriffsattributwert betrachten (wieder lineare Ordnung in eindimensionalem Raum)

# Ein- vs. mehrdim. Zugriffsstruktur (II)

---

- Bsp. eindimensionaler Fall: (Vorname, PLZ) unterstützt keinen *partial-match* nach PLZ

Vorname	PLZ	Adresse
Andreas	18209	•
Christa	69121	•
Gunter	39106	•

- *mehrdimensionaler Fall*: Menge der Zugriffsattributwerte spannt mehrdimensionalen Raum auf  $\Rightarrow$  bei *partial-match* bestimmt horizontale oder vertikale Gerade im Raum die Treffermenge
- Bsp.: Anfragen auch nach PLZ möglich

# Statische vs. dynamische Struktur

---

- *statische Zugriffsstruktur*: optimal nur bei bestimmter (fester) Anzahl von verwaltenden Datensätzen
  - ◆ Bsp. 1: Adreßtransformation für Personalausweisnummer  $p$  von Personen mit  $p \bmod 5$   
5 Seiten, Seitengröße 1 KB, durchschnittliche Satzlänge 200 Bytes, Gleichverteilung der Personalausweisnummern  
⇒ für 25 Personen optimal, für 10.000 Personen nicht mehr ausreichend

# Statische vs. dynamische Struktur (II)

---

- *statische Zugriffsstruktur:*
  - ◆ Bsp. 2: Telefonbücher der Telekommunikationsfirma *Klingel-und-Tuut*  
Indexstruktur dreistufig: Bereichsverzeichnis, in Telefonbuch Index auf Ort, innerhalb eines Ortes Nutzer nach Nachnamen sortiert  
*Klingel-und-Tuut* hat 30 Kunden insgesamt  $\Rightarrow$  nur eine Seite nötig, keine drei Indexstufen nötig  
*Klingel-und-Tuut* wird Weltkonzern und hat 3 Milliarden Kunden: dreistufige Indexstruktur nicht mehr ausreichend, mindestens vierte Stufe (Länderverzeichnisse) nötig

# Statische vs. dynamische Struktur (III)

---

- *Dynamische Zugriffsstrukturen* unabhängig von der Anzahl der Datensätze optimal
  - ◆ dynamische Adreßtransformationsverfahren verändern dynamisch Bildbereich der Transformation
  - ◆ dynamische Indexverfahren verändern dynamisch Anzahl der Indexstufen  $\Rightarrow$  in DBS üblich

# Beispiele für Klassifikationen

---

- Beispiel für dynamisches und eindimensionales Verfahren: *B-Baum* (beliebtester Zugriffspfad in relationalen Datenbanksystemen)
- wird mit **create index** meistens angelegt
- nur *exact-match*, kein *partial-match*
- Beispiel für statisches und eindimensionales Verfahren: klassisches *Hashverfahren*

# Anforderung an Speichertechniken

---

- dynamisches Verhalten
- Effizienz beim Einzelzugriff (Schlüsselsuche beim Primärindex)
- Effizienz beim Mehrfachzugriff (Schlüsselsuche beim Sekundärindex)
- Ausnutzung für sequentiellen Durchlauf (Sortierung, geclusterter Index)
- Clustering
- Anfragetypen: *exact-match*, *partial-match*, *range queries* (Bereichsanfragen)

# Statische Verfahren

---

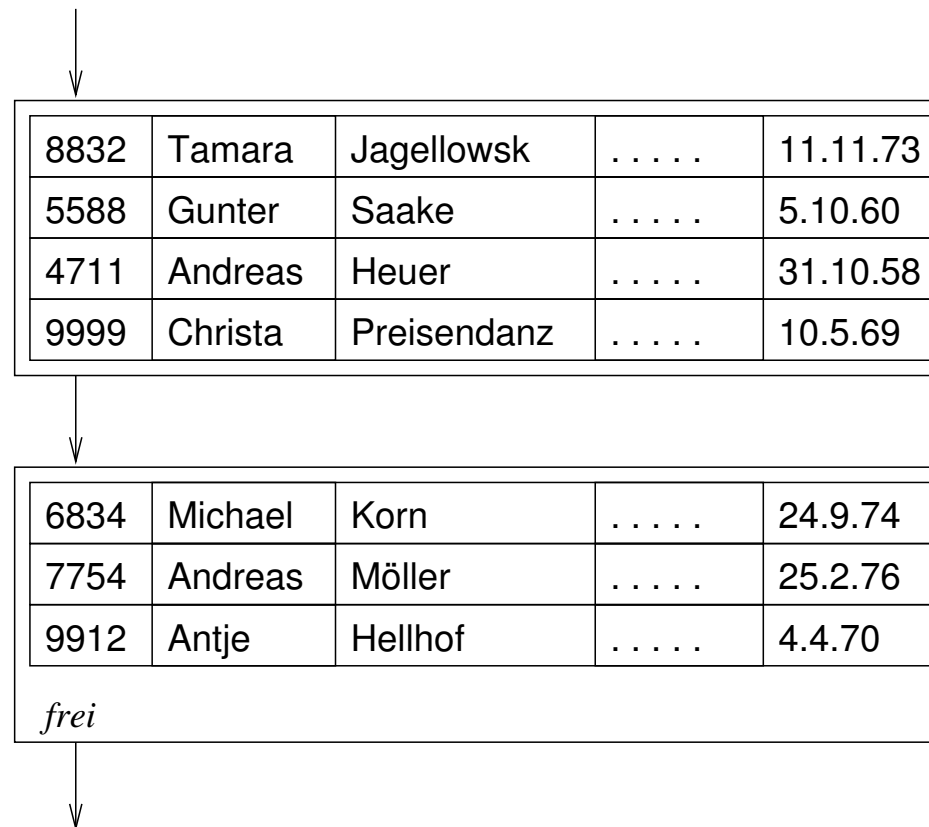
- Heap, indexsequentiell, indiziert-nichtsequentiell
- oft grundlegende Speichertechnik in RDBS
- *direkten Organisationsformen*: keine Hilfsstruktur, keine Adreßberechnung (Heap, sequentiell)
- statische Indexverfahren für Primärindex und Sekundärindex



# Heap-Organisation

---

- völlig unsortiert speichern
- physische Reihenfolge der Datensätze ist zeitliche Reihenfolge der Aufnahme von Datensätzen



# Heap: Operationen

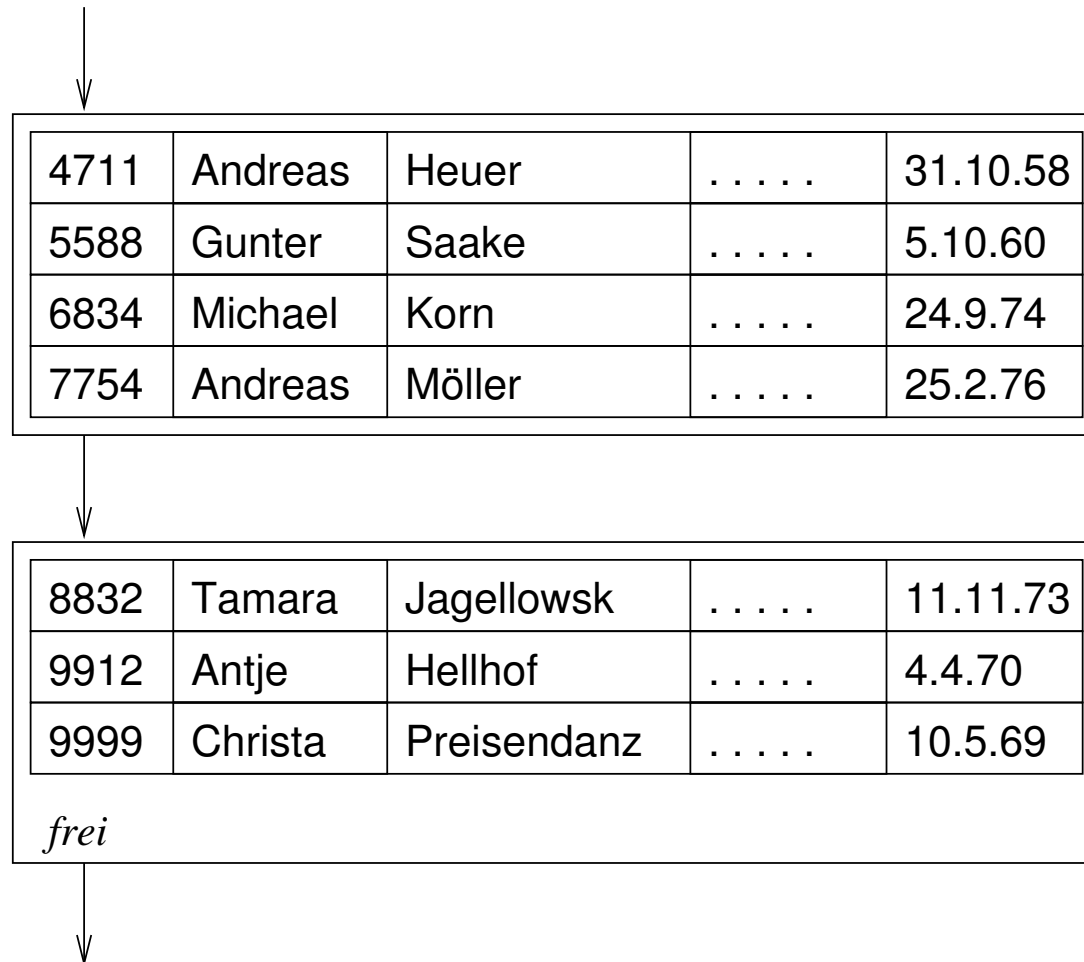
---

- **insert**: Zugriff auf letzte Seite der Datei. Genügend freier Platz  $\Rightarrow$  Satz anhängen. Sonst nächste freie Seite holen
- **delete**: **lookup**, dann Löschbit auf 0 gesetzt
- **lookup**: sequentielles Durchsuchen der Gesamtdatei, maximaler Aufwand (Heap-Datei meist zusammen mit Sekundärindex eingesetzt; oder für sehr kleine Relationen)
- Komplexitäten: Neuaufnahme von Daten  $O(1)$ , Suchen  $O(n)$

# Sequentielle Speicherung

---

## ■ sortiertes Speichern der Datensätze



# Sequentielle Datei: Operationen

---

- **insert**: Seite suchen, Datensatz einsortieren  $\Rightarrow$  beim Anlegen oder sequentiellen Füllen einer Datei jede Seite nur bis zu gewissem Grad (etwa 66%) füllen
- **delete**: Aufwand bleibt
- Folgende Dateiorganisationsformen:
  - ◆ schnelleres **lookup**
  - ◆ mehr Platzbedarf (durch Hilfsstrukturen wie Indexdateien)
  - ◆ mehr Zeitbedarf bei **insert** und **delete**
- klassische Indexform: indexsequentielle Dateiorganisation

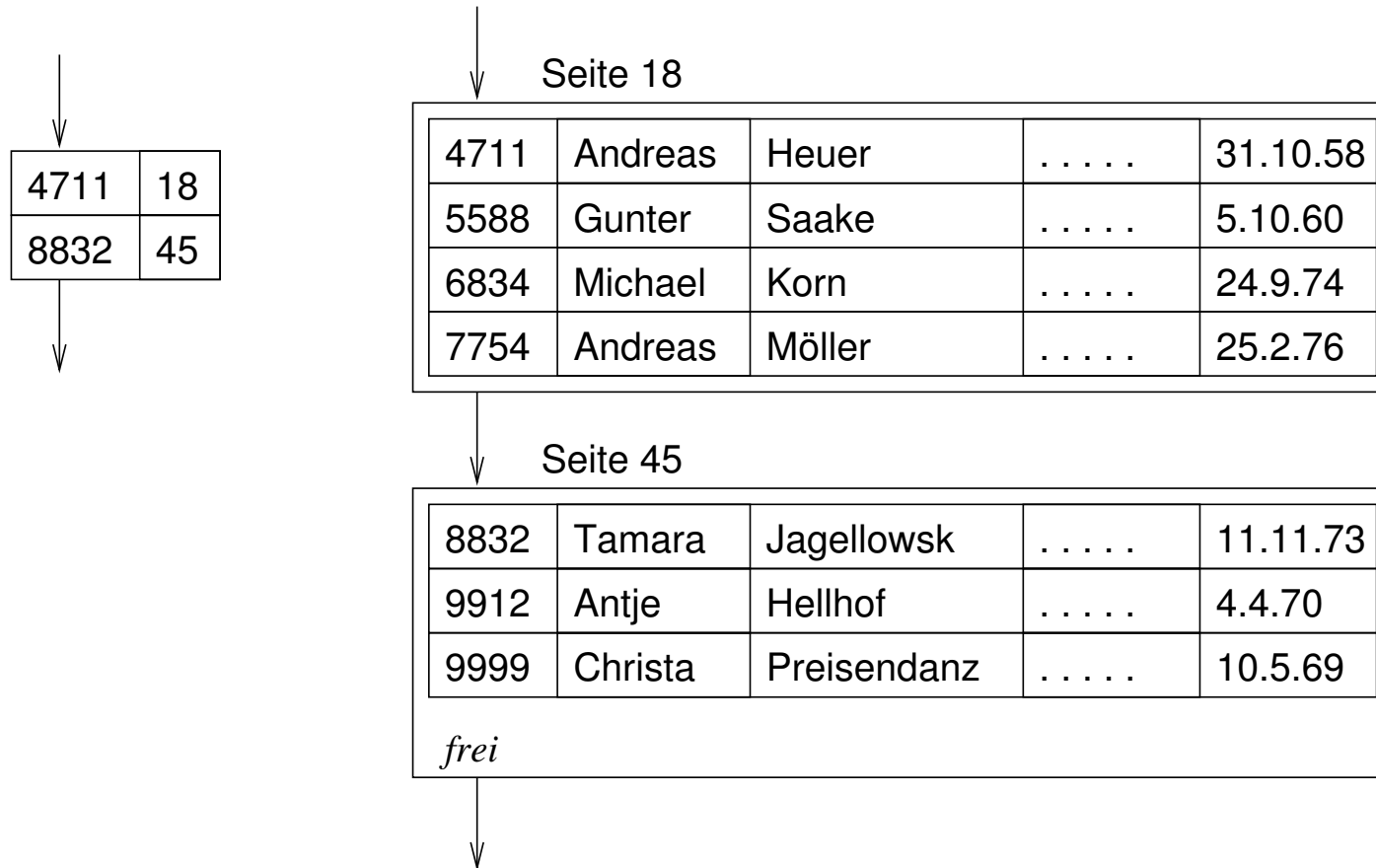
# Indexsequentielle Dateiorganisation

---

- Kombination von sequentieller Hauptdatei und Indexdatei: *indexsequentielle Dateiorganisationsform*
- Indexdatei kann geclusterter, dünnbesetzter Index sein
- mindestens zweistufiger Baum
  - ◆ Blattebene ist *Hauptdatei* (Datensätze)
  - ◆ jede andere Stufe ist *Indexdatei*

# Indexsequentielle Dateiorganisation (II)

---



# Aufbau der Indexdatei

---

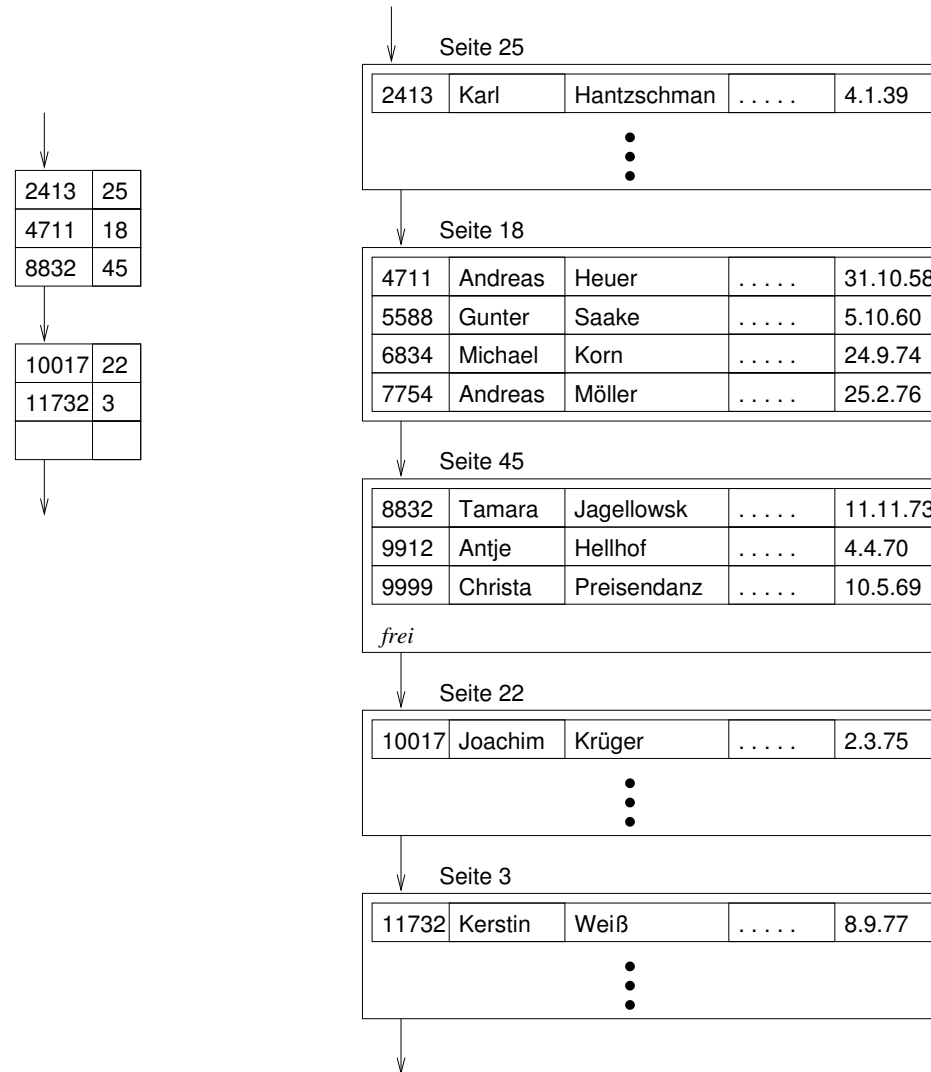
- Datensätze in Indexdatei:

(Primärschlüsselwert, Seitennummer)

zu jeder Seite der Hauptdatei genau ein  
Index-Datensatz in Indexdatei

- Problem: „Wurzel“ des Baumes bei einem einstufigen  
Index nicht nur eine Seite

# Aufbau der Indexdatei (II)



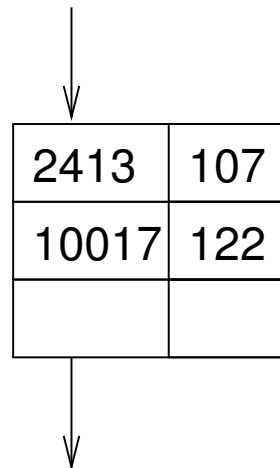


# Mehrstufiger Index

---

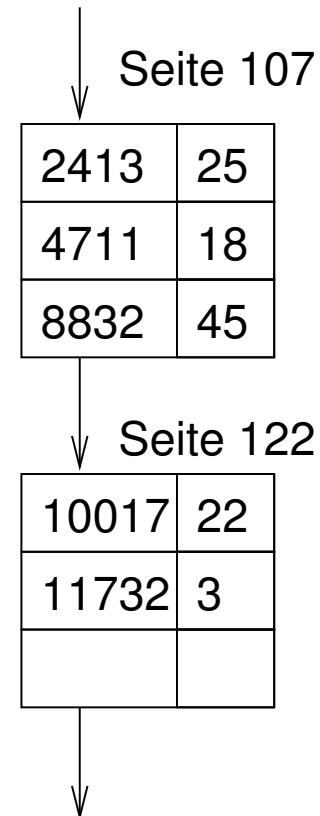
- Optional: Indexdatei wieder indexsequentiell verwalten
- Idealerweise: Index höchster Stufe nur noch eine Seite

Indexdatei 2. Stufe



2413	107
10017	122

Indexdatei 1. Stufe



2413	25
4711	18
8832	45

10017	22
11732	3

# lookup bei indexsequentiellen Dateien

---

**lookup**-Operation sucht Datensatz zum Zugriffsattributwert  $w$

Indexdatei sequentiell durchlaufen, dabei  $(v_1, s)$  im Index gesucht mit  $v_1 \leq w$ :

- $(v_1, s)$  ist letzter Satz der Indexdatei, dann kann Datensatz zu  $w$  höchstens auf dieser Seite gespeichert sein (wenn er existiert)
- nächster Satz  $(v_2, s')$  im Index hat  $v_2 > w$ , also muß Datensatz zu  $w$ , wenn vorhanden, auf Seite  $s$  gespeichert sein

$(v_1, s)$  *überdeckt* Zugriffsattributwert  $w$

# insert bei indexsequentiellen Dateien

---

- **insert**: zunächst mit **lookup** Seite finden
- Falls Platz, Satz sortiert in gefundener Seite speichern; Index anpassen, falls neuer Satz der erste Satz in der Seite
- Falls kein Platz, neue Seite von Freispeicherverwaltung holen; Sätze der „zu vollen“ Seite gleichmäßig auf alte und neue Seite verteilen; für neue Seite Indexeintrag anlegen
- Alternativ neuen Datensatz auf Überlaufseite zur gefundenen Seite

# delete bei indexsequentiellen Dateien

---

- **delete**: zunächst mit **lookup** Seite finden
- Satz auf Seite löschen (Löschbit auf 0)
- erster Satz auf Seite: Index anpassen
- Falls Seite nach Löschen leer: Index anpassen, Seite an Freispeicherverwaltung zurück

# Probleme indexsequentieller Dateien

---

- *stark wachsende Dateien*: Zahl der linear verketteten Indexseiten wächst; automatische Anpassung der Stufenanzahl nicht vorgesehen
- *stark schrumpfende Dateien*: nur zögernde Verringerung der Index- und Hauptdatei-Seiten
- *unausgeglichene Seiten* in der Hauptdatei (unnötig hoher Speicherplatzbedarf, zu lange Zugriffszeit)

# Indiziert-nichtsequentieller Zugriffspfad

---

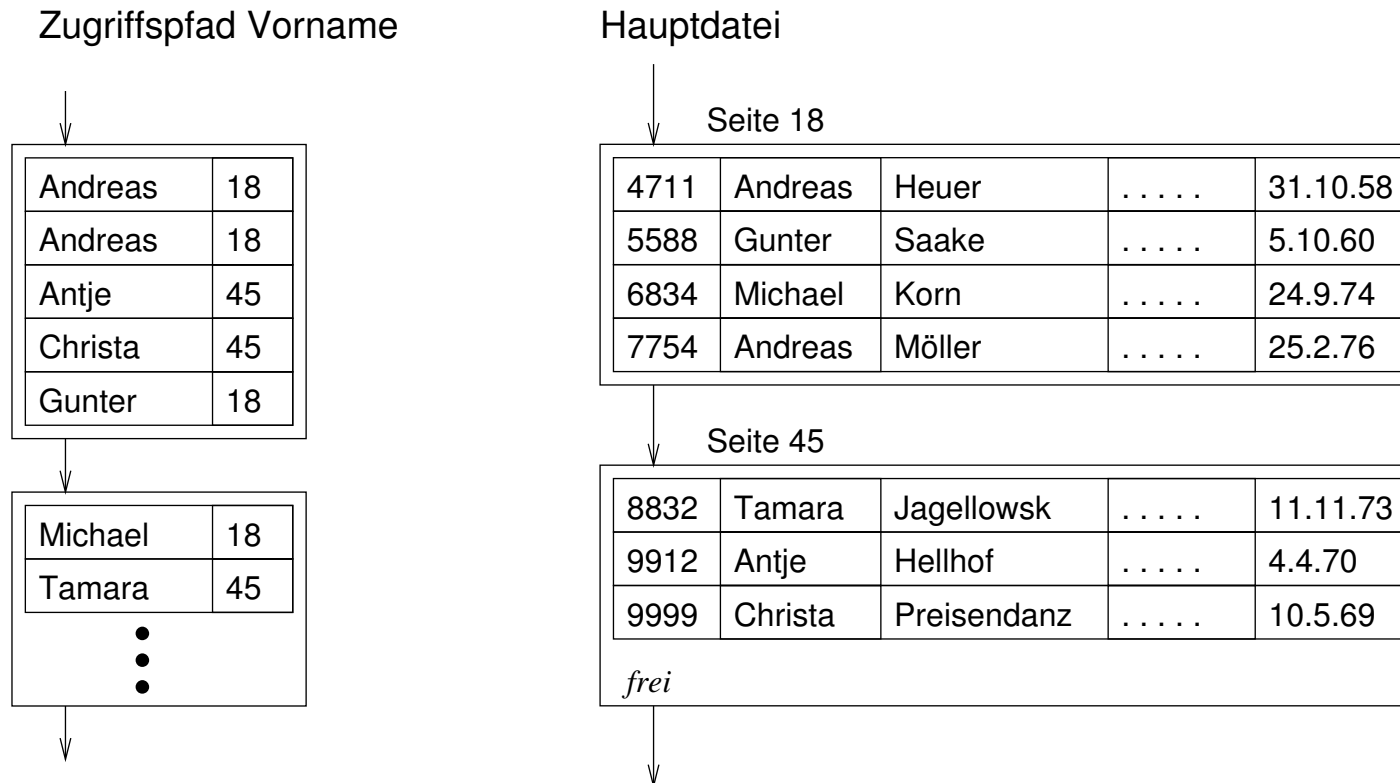
- zur Unterstützung von Sekundärschlüsseln
- mehrere Zugriffspfade dieser Form pro Datei möglich
- einstufig oder mehrstufig: höhere Indexstufen wieder indexsequentiell organisiert

# Aufbau der Indexdatei

---

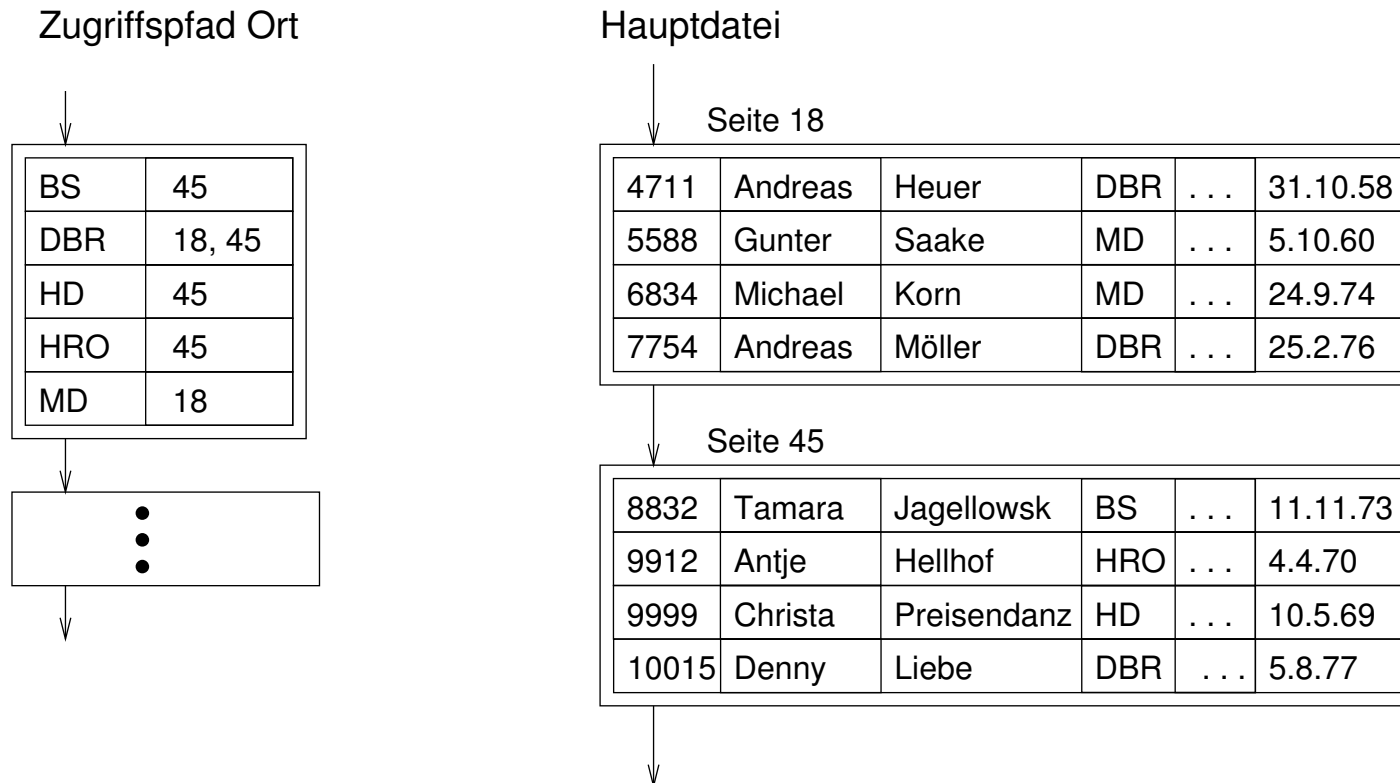
- Sekundärindex, dichtbesetzter und nicht-geclusteter Index
- zu jedem Satz der Hauptdatei Satz  $(w, s)$  in der Indexdatei
- $w$  Sekundärschlüsselwert,  $s$  zugeordnete Seite
  - ◆ entweder für ein  $w$  mehrere Sätze in die Indexdatei aufnehmen
  - ◆ oder für ein  $w$  Liste von Adresse in der Hauptdatei angeben

# Aufbau der Indexdatei (II)





# Aufbau der Indexdatei (III)



# Operationen

---

- **lookup:**  $w$  kann mehrfach auftreten, Überdeckungstechnik nicht benötigt
- **insert:** Anpassen der Indexdateien
- **delete:** Indexeintrag entfernen

# Baumverfahren

---

- Stufenanzahl dynamisch verändern
- wichtigste Baumverfahren: *B-Bäume* und ihre Varianten
- B-Baum-Varianten sind noch „allgegenwärtiger“ in heutigen Datenbanksystemen als SQL
- SQL nur in der relationalen und objektrelationalen Datenbanktechnologie verbreitet; B-Bäume überall als Grundtechnik eingesetzt

# B-Bäume

---

- Ausgangspunkt: ausgeglichener, balancierter Suchbaum
- *Ausgeglichen* oder *balanciert*: alle Pfade von der Wurzel zu den Blättern des Baumes gleich lang
- Hauptspeicher-Implementierungsstruktur: binäre Suchbäume, beispielsweise AVL-Bäume von Adelson-Velskii und Landis
- Datenbankbereich: Knoten der Suchbäume zugeschnitten auf Seitenstruktur des Datenbanksystems
- mehrere Zugriffsattributwerte auf einer Seite
- *Mehrweg-Bäume*

# Prinzip des B-Baumes

---

- *B-Baum* von Bayer (B für balanciert, breit, buschig, Bayer, **NICHT**: binär)
- dynamischer, balancierter Indexbaum, bei dem jeder Indexeintrag auf eine Seite der Hauptdatei zeigt

Mehrwegebaum völlig ausgeglichen, wenn

1. alle Wege von Wurzel bis zu Blättern gleich lang
2. jeder Knoten gleich viele Indexeinträge

vollständiges Ausgleichen zu teuer, deshalb  
B-Baum-Kriterium:

*Jede Seite außer der Wurzelseite enthält zwischen  $m$  und  $2m$  Daten*

# Eigenschaften des B-Baumes

---

$n$  Datensätze in der Hauptdatei  $\Rightarrow$  in  $\log_m(n)$   
Seitenzugriffen von der Wurzel zum Blatt

- Durch Balancierungskriterium wird Eigenschaft nahe an der vollständigen Ausgeglichenheit erreicht (1. Kriterium vollständig erfüllt, 2. Kriterium näherungsweise)
- Kriterium garantiert 50% Speicherplatzausnutzung
- einfache, schnelle Algorithmen zum Suchen, Einfügen und Löschen von Datensätzen (Komplexität von  $O(\log_m(n))$ )

# Eigenschaften des B-Baumes (II)

---

- B-Baum als Primär- und Sekundärindex geeignet
- Datensätze direkt in die Indexseiten  $\Rightarrow$  Dateiorganisationsform
- Verweist man aus Indexseiten auf Datensätze in den Hauptseiten  $\Rightarrow$  Sekundärindex

# Definition B-Baum

---

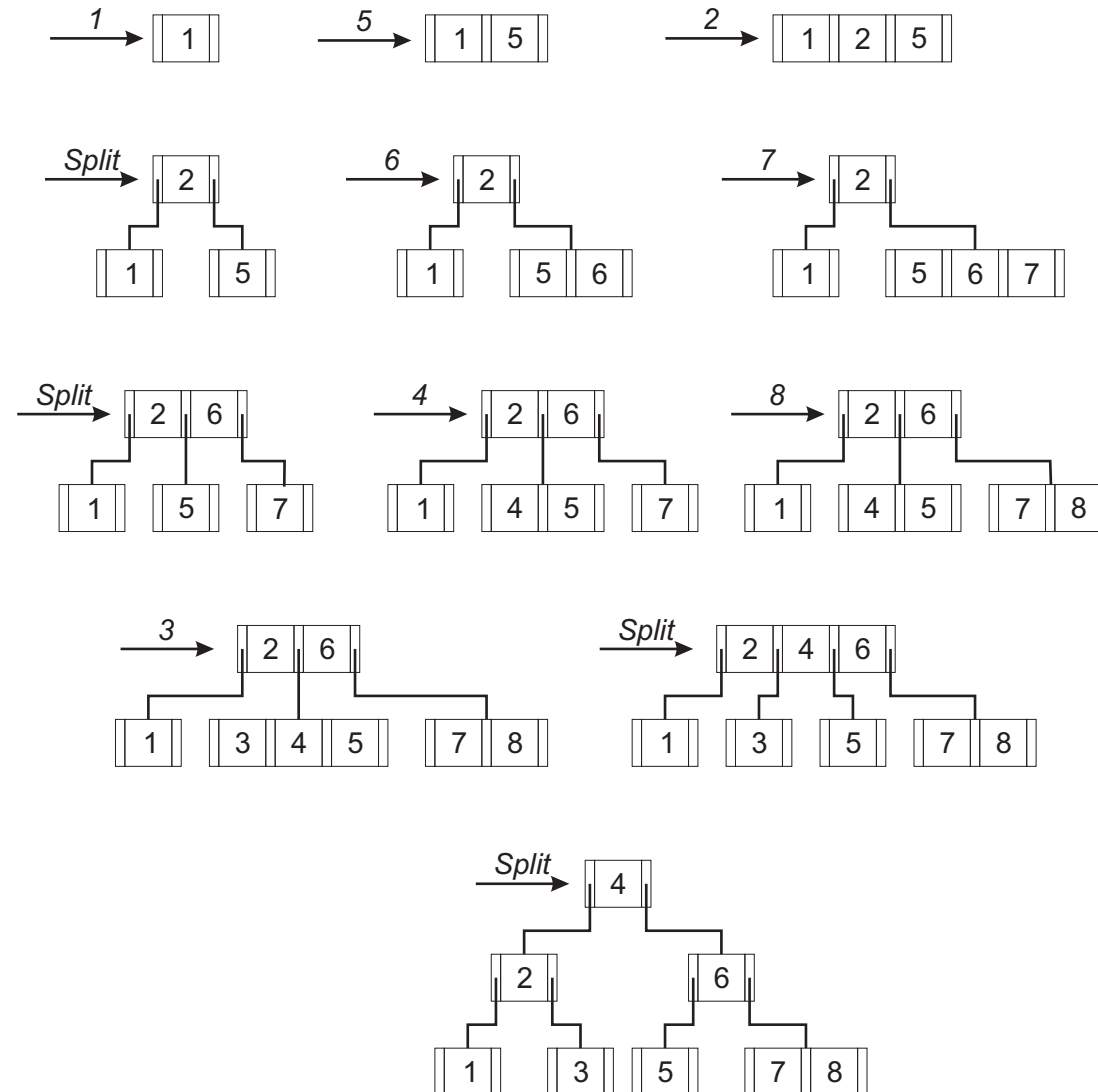
*Ordnung* eines B-Baumes ist minimale Anzahl der Einträge auf den Indexseiten außer der Wurzelseite  
Bsp.: B-Baum der Ordnung 8 faßt auf jeder inneren Indexseite zwischen 8 und 16 Einträgen

Def.: Ein Indexbaum ist ein B-Baum der Ordnung  $m$ , wenn er die folgenden Eigenschaften erfüllt:

1. Jede Seite enthält höchstens  $2m$  Elemente.
2. Jede Seite, außer der Wurzelseite, enthält mindestens  $m$  Elemente.
3. Jede Seite ist entweder eine Blattseite ohne Nachfolger oder hat  $i + 1$  Nachfolger, falls  $i$  die Anzahl ihrer Elemente ist.
4. Alle Blattseiten liegen auf der gleichen Stufe.



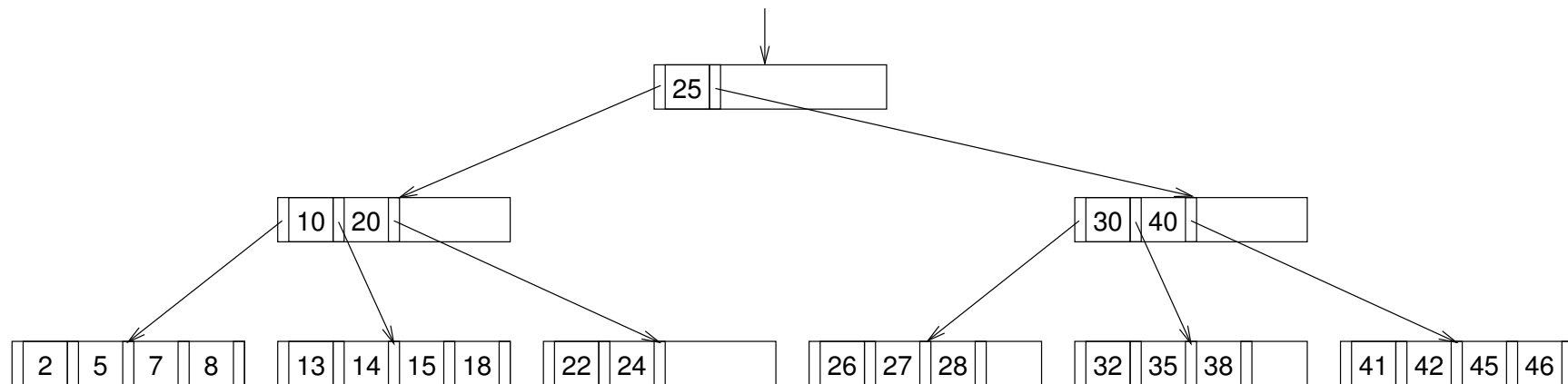
# Einfügen in einen B-Baum: Beispiel



# Suchen in B-Bäumen

---

- **lookup** wie in statischen Indexverfahren
- Startend auf Wurzelseite Eintrag im B-Baum ermitteln, der den gesuchten Zugriffsattributwert  $w$  überdeckt  $\Rightarrow$  Zeiger verfolgen, Seite nächster Stufe laden
- Suchen: 38, 20, 6



# Einfügen in B-Bäumen

---

Einfügen eines Wertes  $w$

- mit **lookup** entsprechende Blattseite suchen
- passende Seite  $n < 2m$  Elemente,  $w$  einsortieren
- passende Seite  $n = 2m$  Elemente, neue Seite erzeugen,
  - ◆ ersten  $m$  Werte auf Originalseite
  - ◆ letzten  $m$  Werte auf neue Seite
  - ◆ mittleres Element auf entsprechende Indexseite nach oben
- eventuell dieser Prozeß rekursiv bis zur Wurzel

# Löschen in B-Bäumen

---

bei weniger als  $m$  Elementen auf Seite: Unterlauf  
Löschen eines Wertes  $w$ : Bsp.: 24; 28, 38, 35

- mit **lookup** entsprechende Seite suchen
- $w$  auf Blattseite gespeichert  $\Rightarrow$  Wert löschen, eventuell Unterlauf behandeln
- $w$  nicht auf Blattseite gespeichert  $\Rightarrow$  Wert löschen, durch lexikographisch nächstkleineres Element von einer Blattseite ersetzen, eventuell auf Blattseite Unterlauf behandeln

# Löschen in B-Bäumen (II)

---

## Unterlaufbehandlung

- Ausgleichen mit der benachbarten Seite (benachbarte Seite  $n$  Elemente mit  $n > m$ )
- oder Zusammenlegen zweier Seiten zu einer (Nachbarseite  $n = m$  Elemente), das „mittlere“ Element von Indexseite darüber dazu, auf Indexseite eventuell Unterlauf behandeln

# Komplexität der Operationen

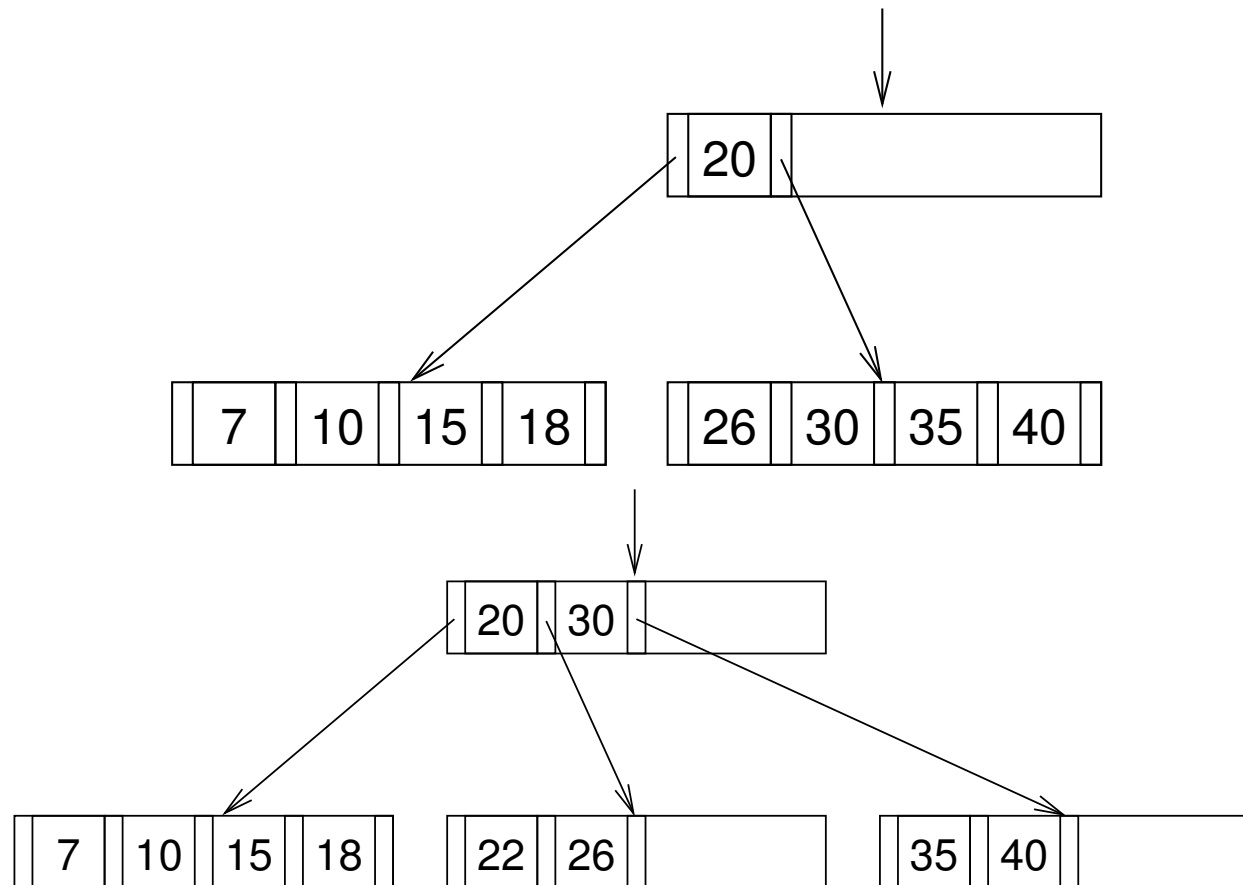
---

- Aufwand beim Einfügen, Suchen und Löschen im B-Baum immer  $O(\log_m(n))$  Operationen
- entspricht genau der „Höhe“ des Baumes
- Konkret: Seiten der Größe 4 KB, Zugriffsattributwert 32 Bytes, 8-Byte-Zeiger: zwischen 50 und 100 Indexeinträge pro Seite; Ordnung dieses B-Baumes 50
- 1.000.000 Datensätze:  $\log_{50}(1.000.000) = 4$  Seitenzugriffe im schlechtesten Fall
- Wurzelseite jedes B-Baumes normalerweise im Puffer: drei Seitenzugriffe

# Einfügen und Löschen im B-Baum

---

Einfügen des Elementes 22; Löschen von 22



# Varianten

---

- $B^+$ -Bäume: Hauptdatei als letzte (Blatt-)Stufe des Baumes integrieren
- $B^*$ -Bäume: Aufteilen von Seiten vermeiden durch „Shuffle“
- Präfix-B-Bäume: Zeichenketten als Zugriffsattributwerte, nur Präfix indexieren

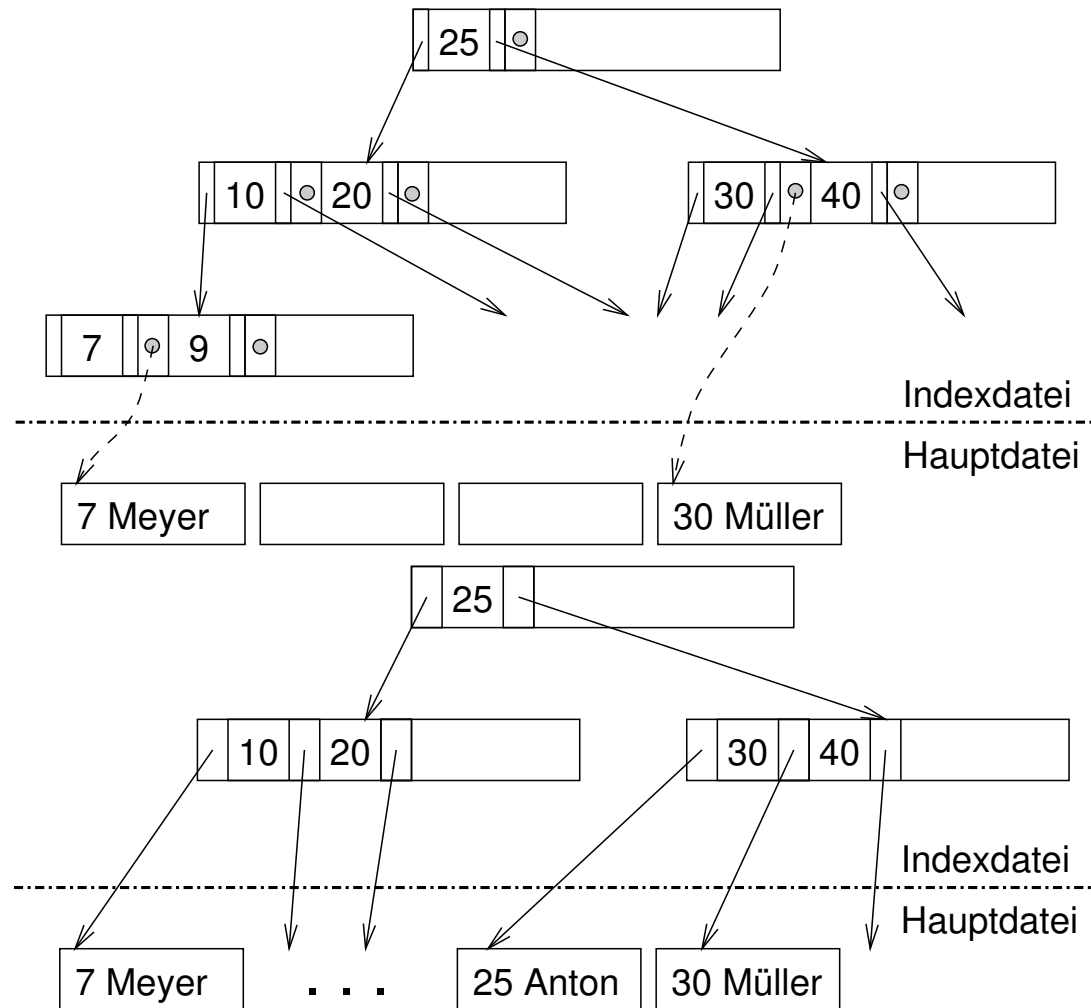


# B<sup>+</sup>-Baum

---

- in der Praxis am häufigsten eingesetzte Variante des B-Baumes: effizientere Änderungsoperationen, Verringerung der Baumhöhe
- integriert Datensätze der Hauptdatei auf den Blattseiten des Baumes
- in inneren Knoten nur noch Zugriffsattributwert und Zeiger auf nachfolgenden Seite der nächsten Stufe

# B-Baum und B<sup>+</sup>-Baum im Vergleich



# Ordnung; Operationen

---

- *Ordnung* für B<sup>+</sup>-Baum:  $(x, y)$ ,  $x$  Mindestbelegung der Indexseiten,  $y$  Mindestbelegung der Datensatz-Seiten
- **delete** gegenüber B-Baum effizienter („Ausleihen“ eines Elementes von der Blattseite entfällt)
- Zugriffsattributwerte in inneren Knoten können sogar stehenbleiben
- häufig als Primärindex eingesetzt
- B<sup>+</sup>-Baum ist dynamische, mehrstufige, indexsequentielle Datei

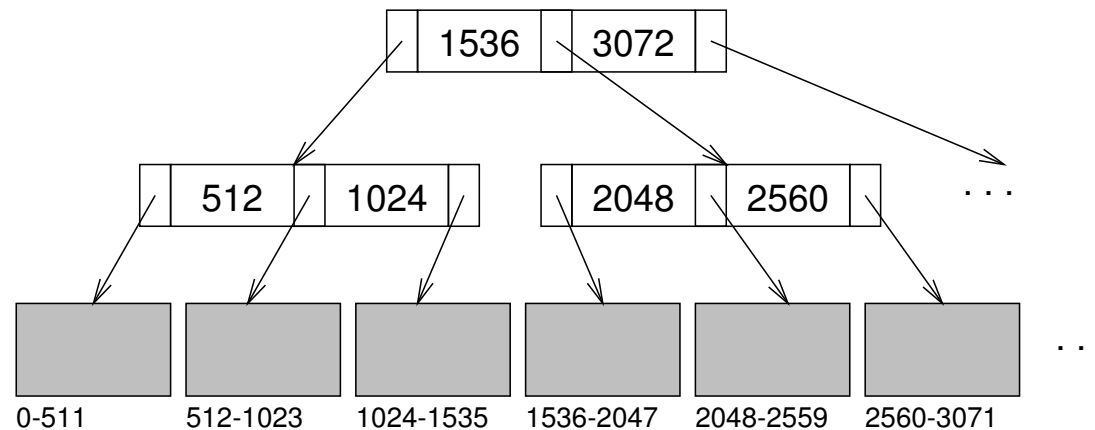
# B\*- und B<sup>#</sup>-Baum

---

- Problem beim B-Baum: häufiges Aufspalten von Seiten und geringe Speicherplatzausnutzung von nahe 50%
- B\*-Baum, B<sup>#</sup>-Baum:
  - ◆ statt Aufteilen von Seiten bei Überlauf zunächst Neuverteilen der Datensätze auf eventuell nicht voll ausgelastete Nachbarseiten
  - ◆ falls nicht möglich: zwei Seiten in drei aufteilen (ermöglicht durchschnittliche Speicherplatzausnutzung von 66% statt 50%)

# B<sup>+</sup>-Baum für BLOBs

- Statt Zugriffsattributwerte in B<sup>+</sup>-Baum: Positionen oder Offsets im BLOB indexieren
- BLOB-B<sup>+</sup>-Baum: Positions-B<sup>+</sup>-Baum



- Auch für andere große Speicherobjekte (wie in objektorientierten Datenbanken üblich) geeignet

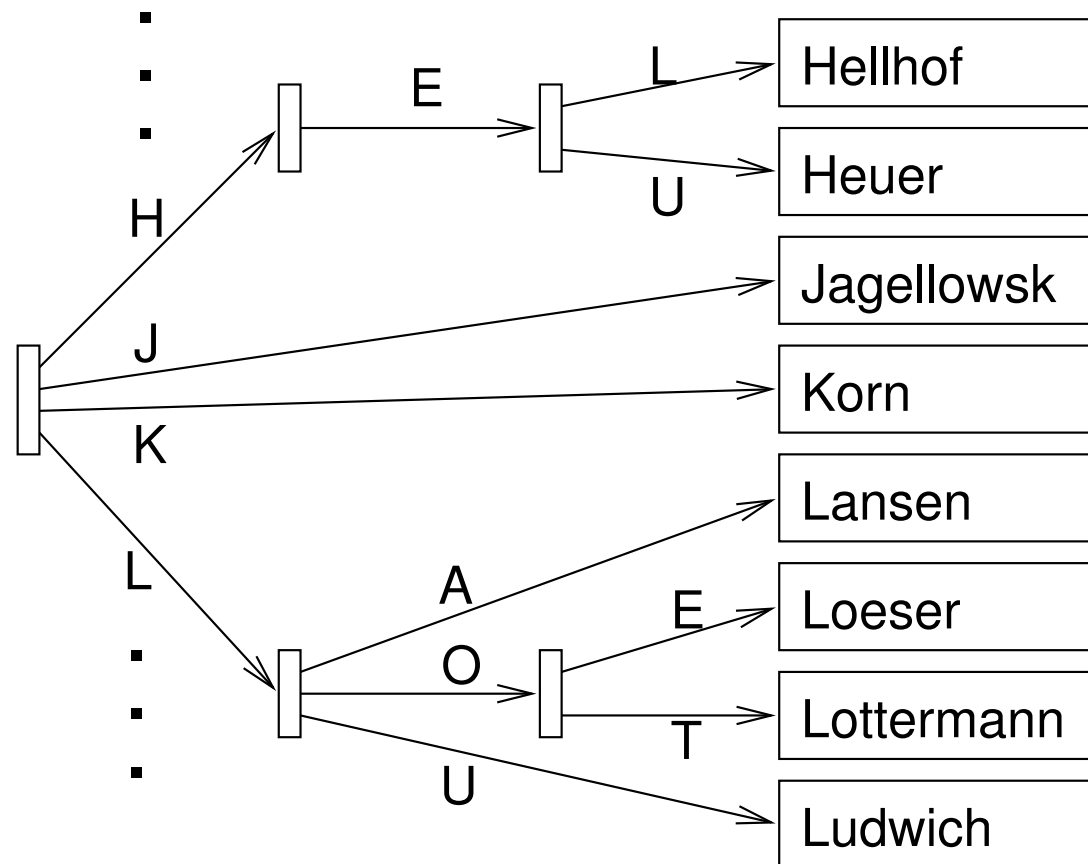
# Digital- und Präfixbäume

---

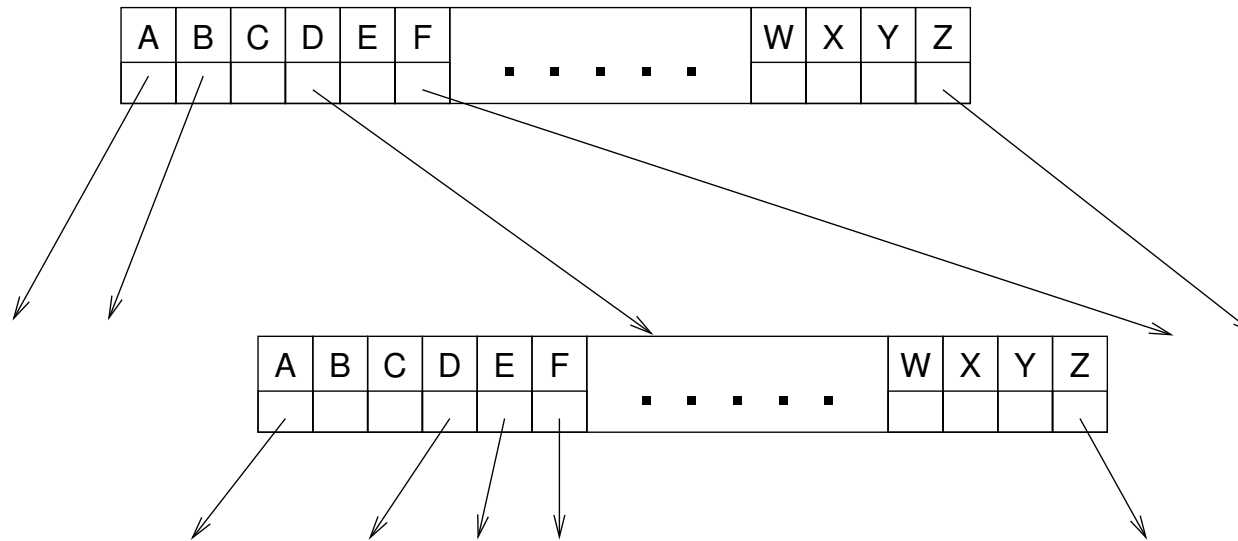
- B-Bäume: Problem bei zu indexierenden Zeichenketten
- Lösung: Digital- oder Präfixbäume
- Digitalbäume indexieren (fest) die Buchstaben des zugrundeliegenden Alphabets
- können dafür unausgeglichen werden
- Beispiele: Tries, Patricia-Bäume
- Präfixbäume indexieren Präfix der Zeichenketten

# Tries

- von „Information Retrieval“, aber wie *try* gesprochen



# Knoten eines Tries



- Probleme: lange gemeinsame Teilworte, nicht vorhandenen Buchstaben und Buchstabenkombinationen, möglicherweise leere Knoten, sehr unausgeglichene Bäume

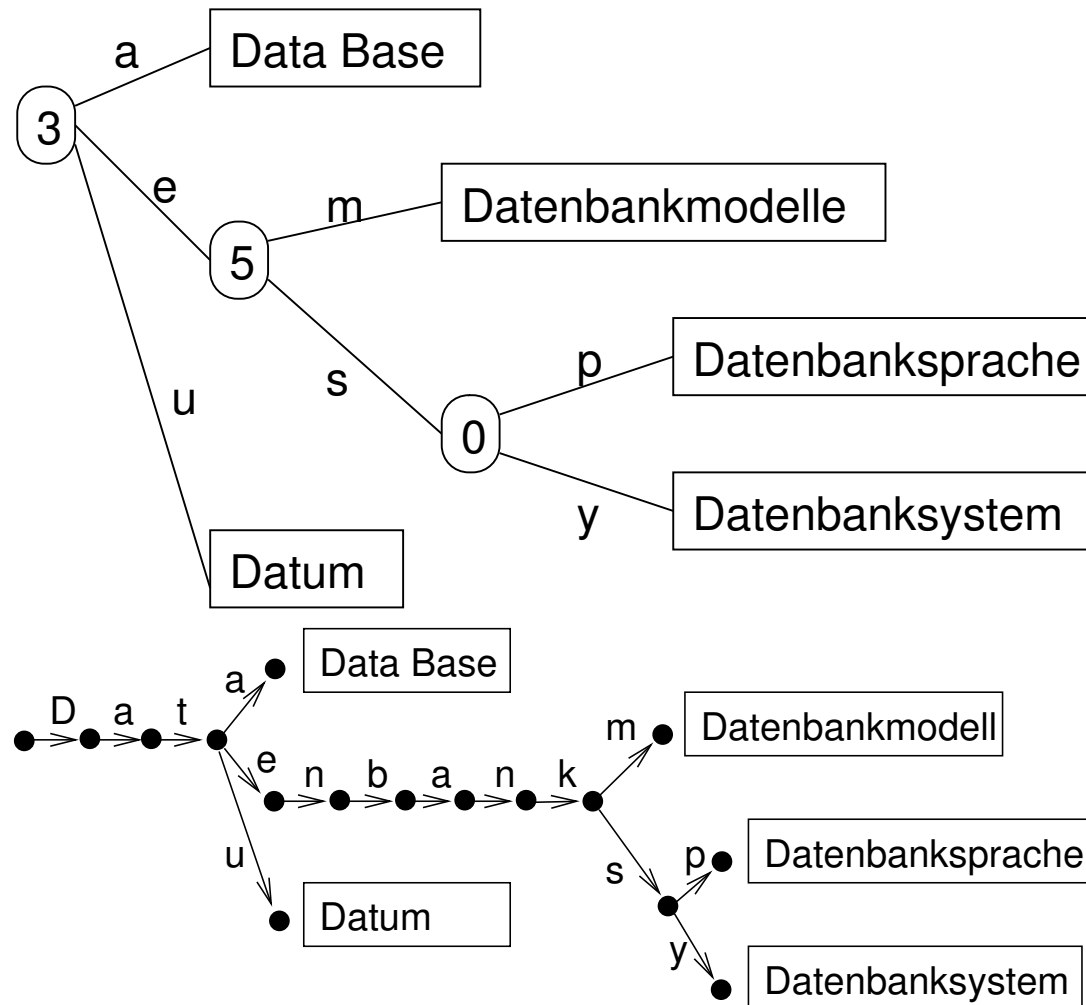


# Patricia-Bäume

---

- Tries: Probleme bei Teilekennzahlen, Pfadnamen, URLs (lange gemeinsame Teilworte)
- Lösung: *Practical Algorithm To Retrieve Information Coded In Alphanumeric* (Patricia)
- Prinzip: Überspringen von Teilworten
- Problem: Datenbanksprache bei Suchbegriff  
Triebwerksperre

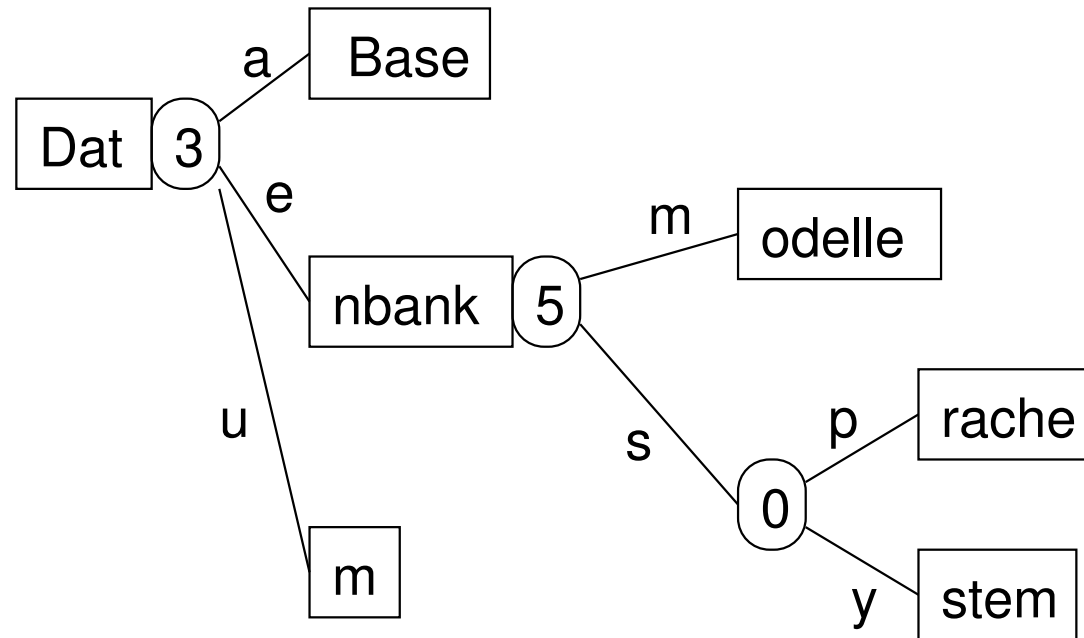
# Patricia-Baum und Trie im Vergleich



übersprungene Teilworte zusätzlich speichern:  
Präfix-Bäume

# Präfix-Bäume

---



# Hash-Verfahren

---

- Schlüsseltransformation und Überlaufbehandlung
- DB-Technik: Bildbereich entspricht Seiten-Adreßraum
- Dynamik: dynamische Hash-Funktionen oder Re-Hashen

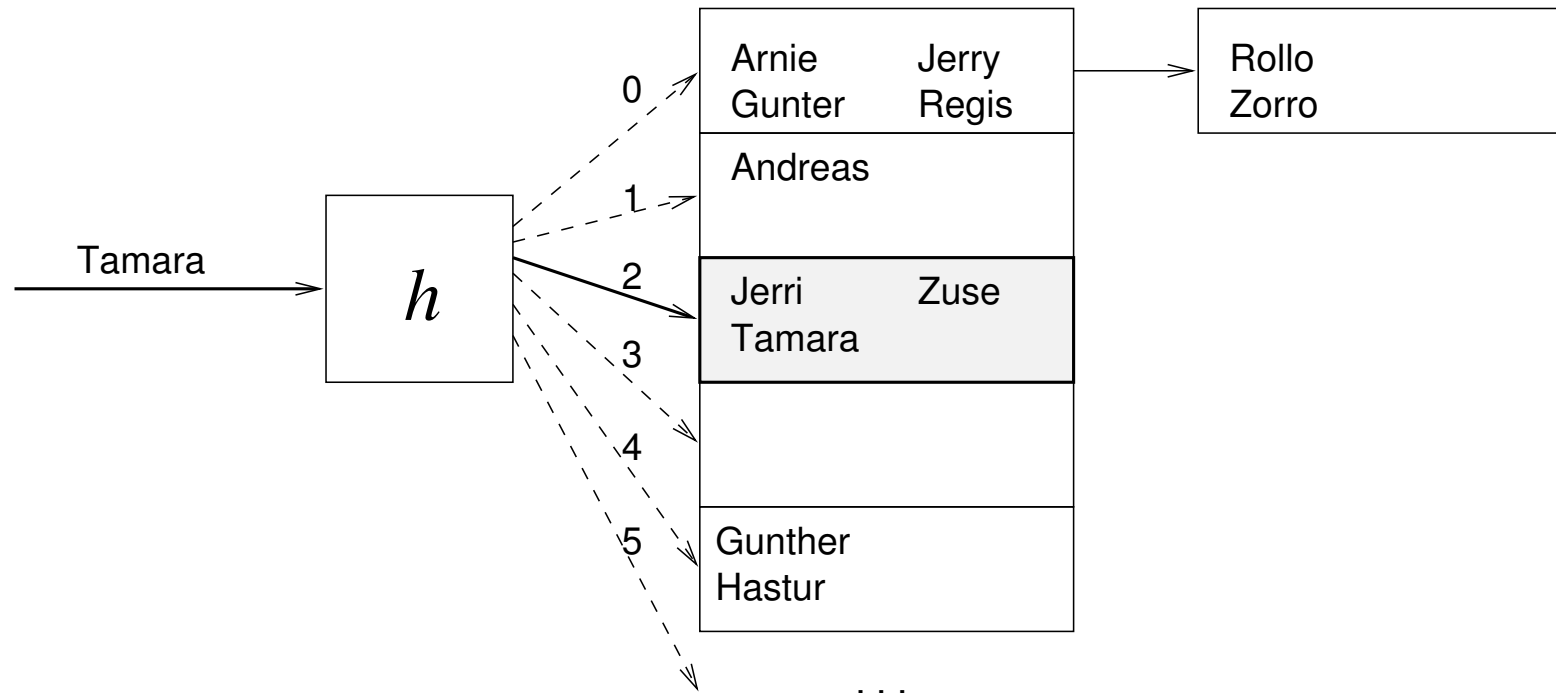
# Grundprinzipien

---

- Basis-Hash-Funktion:  $h(k) = k \bmod m$
- $m$  möglichst Primzahl
- Überlauf-Behandlung
  - ◆ Überlaufseiten als verkettete Liste
  - ◆ lineares Sondieren
  - ◆ quadratisches Sondieren
  - ◆ doppeltes Hashen

# Hash-Verfahren für Datenbanken

---



# Operationen und Zeitkomplexität

---

- **lookup, modify, insert, delete**
- **lookup** benötigt maximal  $1 + \#B(h(w))$  Seitenzugriffe
- $\#B(h(w))$  Anzahl der Seiten (inklusive der Überlaufseiten) des Buckets für Hash-Wert  $h(w)$
- Untere Schranke 2 (Zugriff auf Hash-Verzeichnis plus Zugriff auf erste Seite)

# Statisches Hashen: Probleme

---

- mangelnde Dynamik
- Vergrößerung des Bildbereichs erfordert komplettes Neu-Hashen
- Wahl der Hash-Funktion entscheidend; Bsp.: Hash-Index aus 100 Buckets, Studenten über 6-stellige MATNR (wird fortlaufend vergeben) hashen
  - ◆ ersten beiden Stellen: Datensätze auf wenigen Seiten quasi sequentiell abgespeichert
  - ◆ letzten beiden Stellen: verteilen die Datensätze gleichmäßig auf alle Seiten
- Sortiertes Ausgeben einer Relation schlecht



# Dynamisches Hashen

---

- statische Hash-Verfahren
  - ◆ Neu-Hashen bei steigender Auslastung
- dynamische Hash-Verfahren
  - ◆ Hash-Verfahren mit dynamisch vergrößerbarem Bildbereich
    - meist Verdopplung des Bildbereichs...
  - ◆ oft in Verbindung mit zusätzlichem Index zur Adressierung von Blöcken
- lineares Hashen als Beispiel

# Lineares Hashen

---

Folge von Hash-Funktionen, die wie folgt charakterisiert sind:

- $h_i : \text{dom}(\text{Primärschlüssel}) \rightarrow \{0, \dots, 2^i \times N\}$  ist eine Folge von Hash-Funktionen mit  $i \in \{0, 1, 2, \dots\}$  und  $N$  als Anfangsgröße des Hash-Verzeichnisses
- Wert von  $i$  wird auch als *Level* der Hash-Funktion bezeichnet  
 $\text{dom}(\text{Primärschlüssel})$  wird im folgenden als  $\text{dom}(Prim)$  abgekürzt

# Lineares Hashen (II)

---

- Für diese Hash-Funktionen gelten die folgenden Bedingungen:

- ◆  $h_{i+1}(w) = h_i(w)$  für etwa die Hälfte aller  $w \in \text{dom}(\text{Prim})$

- ◆  $h_{i+1}(w) = h_i(w) + 2^i \times N$  für die andere Hälfte

Bedingungen sind zum Beispiel erfüllt, wenn  $h_i(w)$  als  $w \bmod (2^i \times N)$  gewählt wird

- ◆ Darstellung durch Bit-Strings, Hinzunahme eines Bits verdoppelt Bildbereich

# Prinzip lineares Hashen

---

für ein  $w$  höchstens zwei Hash-Funktionen zuständig, deren Level nur um 1 differiert, Entscheidung zwischen diesen beiden durch *Split-Zeiger*

- $sp$  Split-Zeiger (gibt an, welche Seite als nächstes geteilt wird)
- $lv$  Level (gibt an, welche Hash-Funktionen benutzt werden)

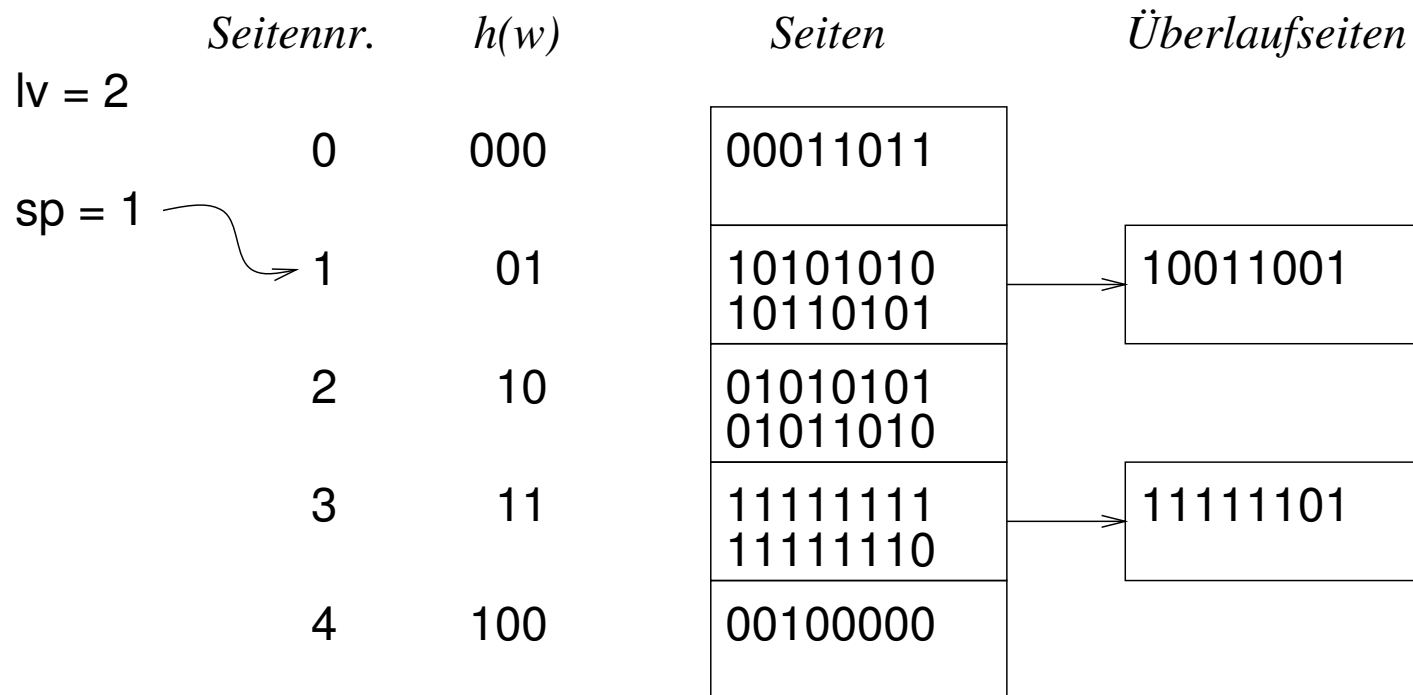
Aus Split-Zeiger und Level läßt sich die Gesamtanzahl  $Anz$  der belegten Seiten wie folgt berechnen:

$$Anz = 2^{lv} + sp$$

Beide Werte werden am Anfang mit 0 initialisiert.

# Prinzip lineares Hashen (II)

---



# Lookup

---

```
 $s := h_{lv}(w);$   
if  $s < sp$   
  then  $s := h_{lv+1}(w);$ 
```

- zuerst Hash-Wert mit der „kleineren“ Hash-Funktion bestimmen
- liegt dieser unter dem Wert des Split-Zeigers  $\Rightarrow$  größere Hash-Funktion verwenden

# Splitten einer Seite

---

1. Die Sätze der Seite (Bucket), auf die  $sp$  zeigt, werden mittels  $h_{lv+1}$  neu verteilt (ca. die Hälfte der Sätze wird auf Seite (Bucket) unter Hash-Nummer  $2^{lv} * N + sp$  verschoben)
2. Der Split-Zeiger wird weitergesetzt:  $sp := sp + 1$ ;
3. Nach Abarbeiten eines Levels wird wieder bei Seite 0 begonnen; der Level wird um 1 erhöht:

```
if  $sp = 2^{lv} * N$  then  
    begin  
         $lv := lv + 1$ ;  
         $sp := 0$   
    end;
```

# Beispiel nach Split

	<i>Seitennr.</i>	<i>h(w)</i>	<i>Seiten</i>	<i>Überlaufseiten</i>
lv = 2	0	000	00011011	
sp = 2	1	001	10011001	
	2	10	01010101 01011010	
	3	11	11111111 11111110	11111101
	4	100	00100000	
	5	101	10101010 10110101	



# Problem lineares Hashen

lv = 3  
sp = 0

Seitennr.	$h(w)$	Seiten	Überlaufseiten
0	000	00011011	
1	001	10011001	
2	010	01010101 01011010	
3	011		
4	100	00100000	
5	101	10101010 10110101	
6	110		
7	111	11111111 11111110	11111101

# Verbesserungen

---

- Erweiterbares Hashen
- Spiral-Hashen
- Kombinationen dieser Techniken

# Erweiterbares Hashen

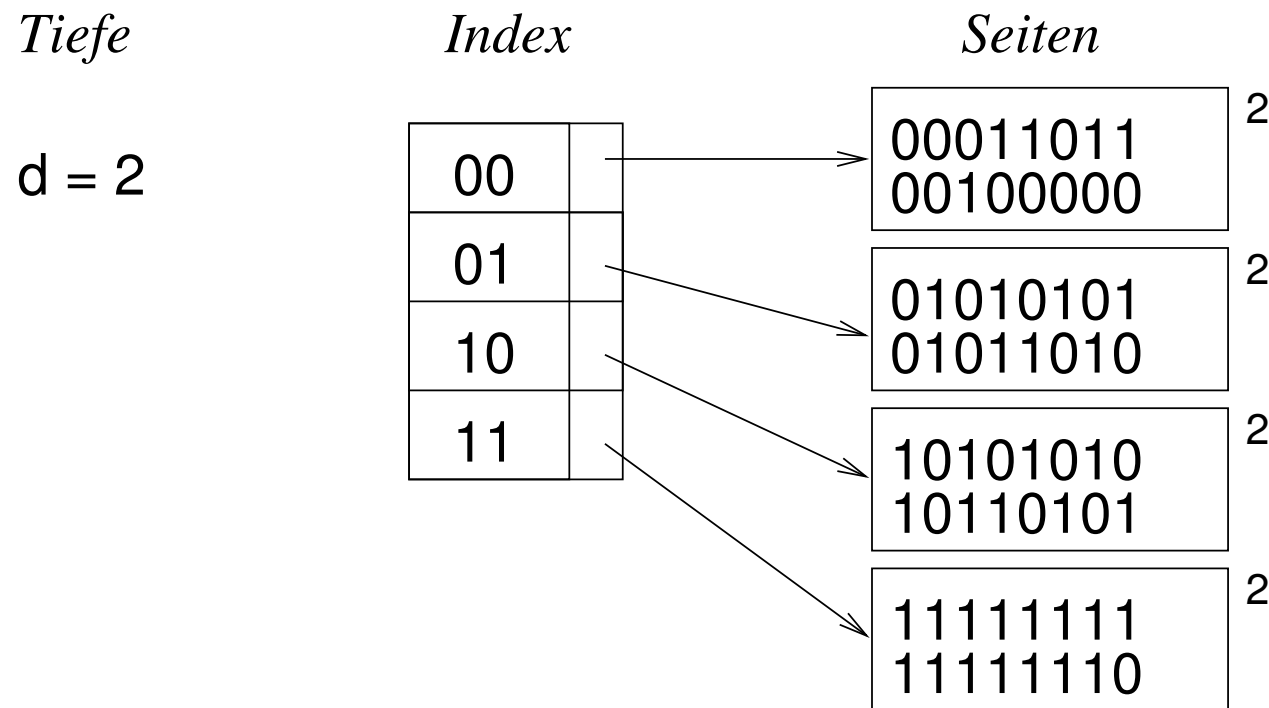
---

- Problem: Split erfolgt an fester Position, nicht dort wo Seiten überlaufen
- Idee: binärer Trie zum Zugriff auf Indexseiten
- Blätter unterschiedlicher Tiefe
  - ◆ Indexseiten haben Tiefenwert
  - ◆ Split erfolgt bei Überlauf
- aber: Speicherung nicht als Trie, sondern als Array
  - ◆ entspricht vollständigem Trie mit maximaler Tiefe
    - “shared” Seiten als Blätter
  - ◆ Array der Grösse  $2^d$  für maximale Tiefe  $d$ 
    - erfordert nun nur einen Speicherzugriff!
  - ◆ bei Überlauf: Indexgrösse muss möglicherweise verdoppelt werden!

# Erweiterbares Hashen II

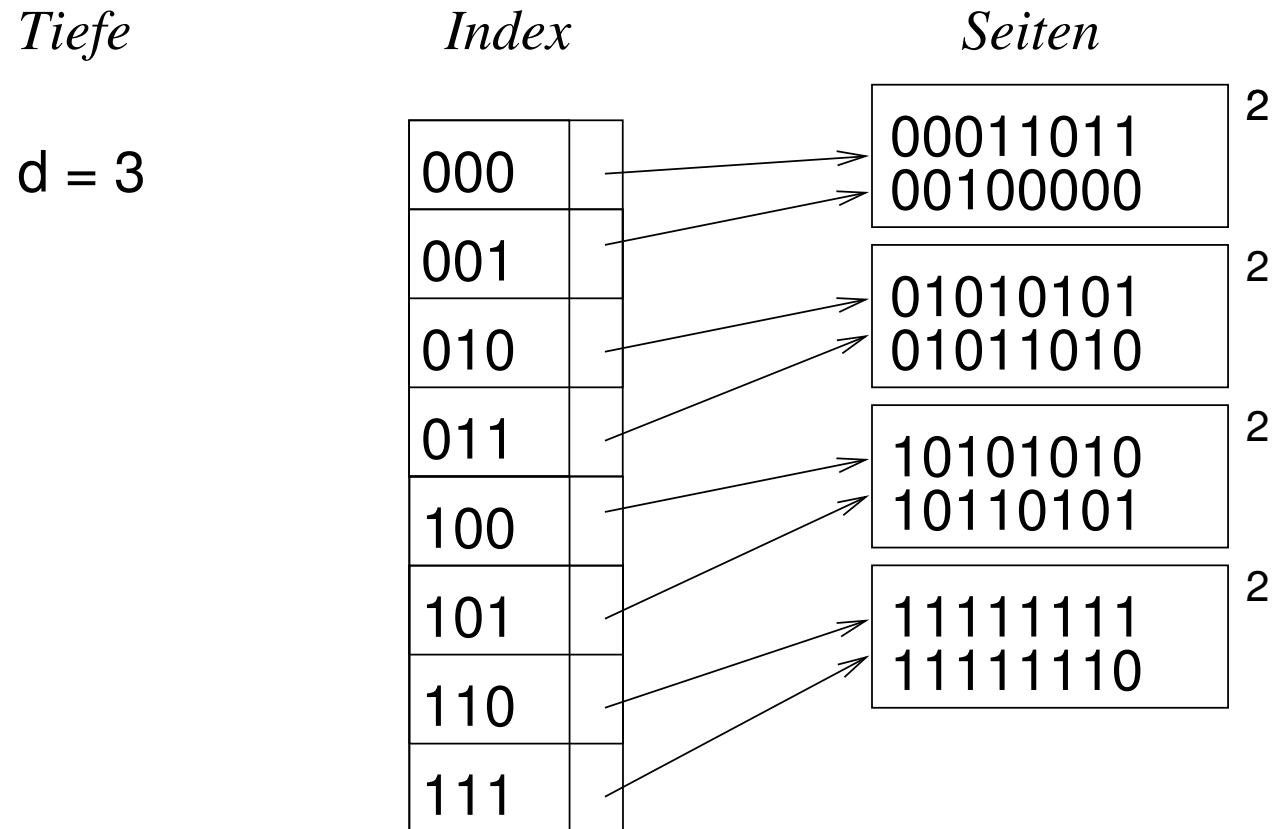
## ■ Ausgangslage:

- ◆ Einfügen von 00111111 würde Überlauf bei erreichter maximaler Tiefe erzeugen



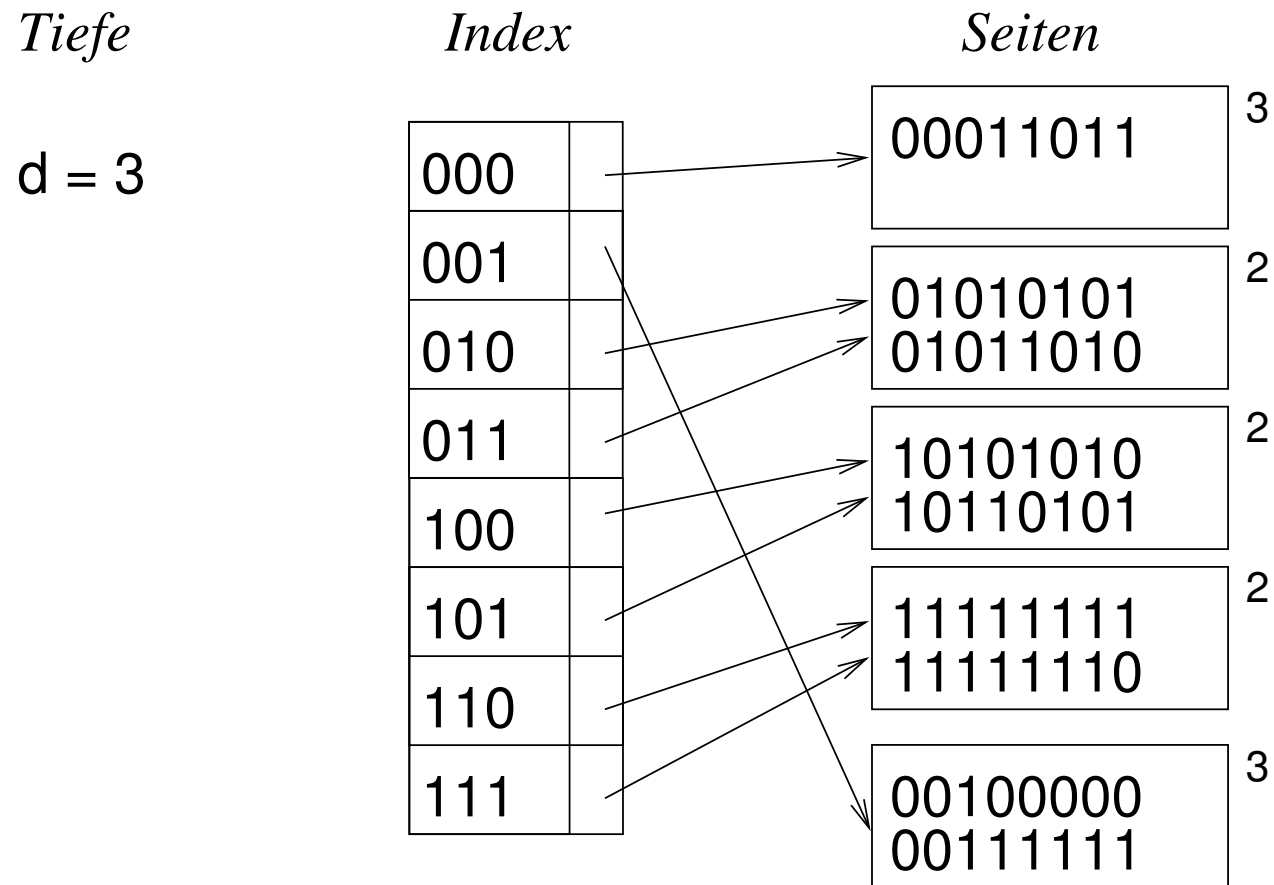
# Erweiterbares Hashen III

## ■ Verdopplung der Indexgrösse



# Erweiterbares Hashen IV

- nun möglich: Split der Seite



# Spiral-Hashen I

---

- Problem: zyklisch erhöhte Wahrscheinlichkeit des Splittens
- Lösung: unterschiedliche Dichte der Hashwerte
  - ◆ Interpretation der Bit-Strings als binäre Nachkommadarstellung einer Zahl zwischen 0.0 und 1.0
  - ◆ Funktion von  $[0.0, 1.0] \rightarrow [0.0, 1.0]$  so dass Dichte gleichmässig verteilter Werte nahe 1.0 doppelt so gross ist wie nahe 0.0

# Spiral-Hashen II

---

- Umverteilung mittels Exponentialfunktion
- Funktion  $exp(n)$

$$exp(n) = 2^n - 1$$

erfüllt die Bedingungen

◆ insbesondere gilt  $2^0 - 1 = 0$  und  $2^1 - 1 = 1$

- Hashfunktion **exhash**

$$\mathbf{exhash}(k) = exp(h(k)) = 2^{h(k)} - 1$$



# Spiral-Hashen III

---

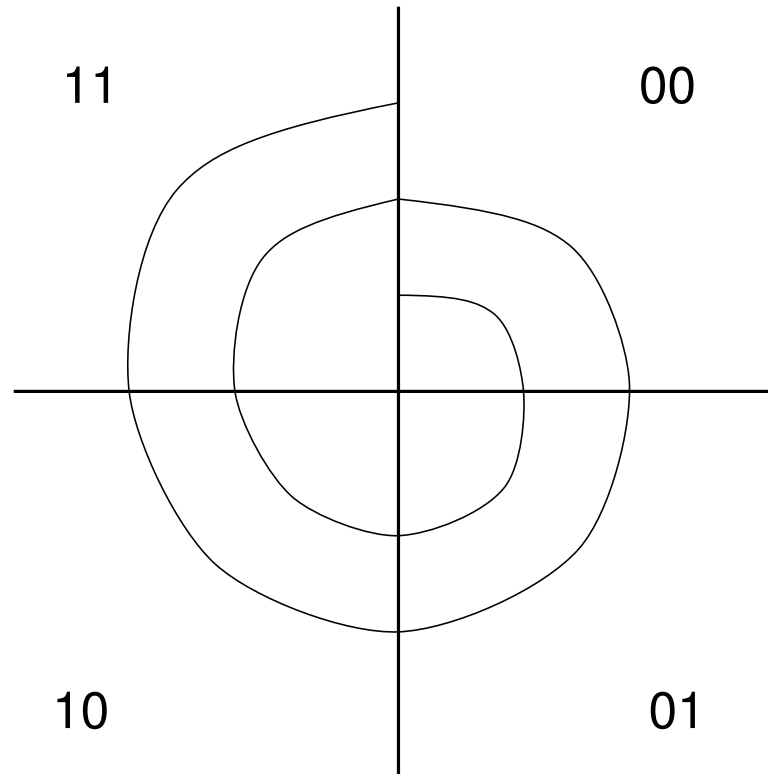
- Wirkung der verwendeten Hashfunktion im Intervall 0.0 bis 1.0

$n$	$2^n - 1$
0.0	0.0
0.1	0.0717735
0.2	0.1486984
0.3	0.2311444
0.4	0.3195079
0.5	0.4142136
0.6	0.5157166
0.7	0.6245048
0.8	0.7411011
0.9	0.866066
1.0	1.0

# Spiral-Hashen IV

---

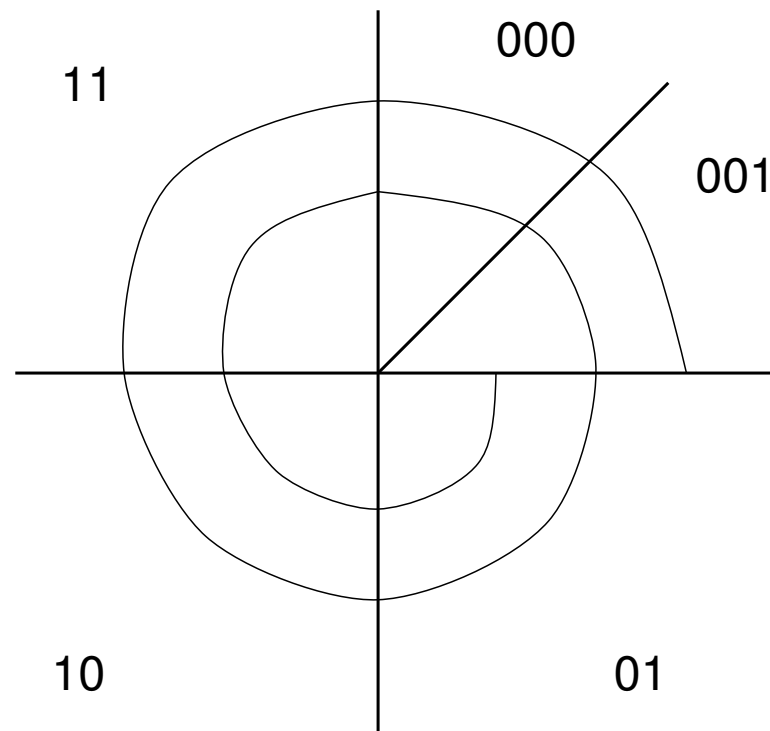
- Spiralförmiges Ausbreiten
  - ◆ Ausgangslage: 4 Seiten der Tiefe 2



# Spiral-Hashen V

---

- Spiralförmiges Ausbreiten
  - ◆ Split der Seite mit der höchsten Dichte
  - ◆ Ergebnis: 5 Seiten, davon 3 der Tiefe 2 und 2 der Tiefe 3



# Bitmap-Indexe

---

- Idee: *Bit-Array* zur Kodierung der Tupel-Attributwert-Zuordnung
- Vergleich mit baumbasierten Indexstrukturen:
  - ◆ vermeidet degenerierte B-Bäume
  - ◆ unempfindlicher gegenüber höherer Zahl von Attributen
  - ◆ einfachere Unterstützung von Anfragen, in denen nur einige (der indexierten) Attribute beschränkt werden
  - ◆ dafür aber i.allg. höhere Aktualisierungskosten
    - beispielsweise in Data Warehouses wegen des überwiegend lesenden Zugriffs unproblematisch

# Bitmap-Index: Realisierung

---

- Prinzip: Ersetzung der TIDs (rowid) für einen Schlüsselwert im  $B^+$ -Baum durch Bitliste
- Knotenaufbau:

Bestellstatus		
$F : 010010 \dots 01$	$O : 0111010 \dots 00$	$P : 100000 \dots 10$

- Vorteil: geringerer Speicherbedarf
  - ◆ Beispiel: 150.000 Tupel, 3 verschiedene Schlüsselwerte, 4 Byte für TID
    - $B^+$ -Baum: 600 KB
    - Bitmap:  $3 \cdot 18750 \text{ Byte} = 56\text{KB}$
- Nachteil: Aktualisierungsaufwand

# Bitmap-Index: Realisierung /2

---

- Definition in Oracle

```
CREATE BITMAP INDEX bestellstatus_idx  
ON bestellung(status);
```

- Speicherung in komprimierter Form

# Standard-Bitmap-Index

---

- jedes Attribut wird getrennt abgespeichert
- für jede Ausprägung eines Attributs wird ein Bitmap-Vektor angelegt:
  - ◆ für jedes Tupel steht ein Bit, dieses wird auf 1 gesetzt, wenn das indexierte Attribut in dem Tupel den Referenzwert dieses Bitmap-Vektors enthält
  - ◆ die Anzahl der entstehenden Bitmap-Vektoren pro Dimension entspricht der Anzahl der unterschiedlichen Werte, die für das Attribut vorkommen

# Standard-Bitmap-Index /2

---

- Beispiel: Attribut Geschlecht
  - ◆ 2 Wertausprägungen (m/w)
  - ◆ 2 Bitmap-Vektoren

PersId	Name	Geschlecht	Bitmap-w	Bitmap-m
007	James Bond	M	0	1
008	Amelie Lux	W	1	0
010	Harald Schmidt	M	0	1
011	Heike Drechsler	W	1	0



# Standard-Bitmap-Index /3

---

- Selektion von Tupeln kann nun durch entsprechende Verknüpfung von Bitmap-Vektoren erfolgen
- Beispiel: Bitmap-Index über Attribute Geschlecht und Geburtsmonat
  - ◆ (d.h. 2 Bitmap-Vektoren B-w und B-m für Geschlecht und 12 Bitmap-Vektoren B-1, ..., B-12 für die Monate, wenn alle Monate vorkommen)
- Anfrage: „alle im März geborenen Frauen“
  - ◆ Berechnung:  $B-w \wedge B-3$  (bitweise konjunktiv verknüpft)
  - ◆ Ergebnis: alle Tupel, an deren Position im Bitmap-Vektor des Ergebnis eine 1 steht

# Mehrkomponenten-Bitmap-Index

---

- bei Standard-Bitmap-Indexten entstehen für Attribute mit vielen Ausprägungen sehr viele Bitmap-Vektoren
- $\langle n, m \rangle$ -Mehrkomponenten-Bitmap-Indexte erlauben es  $n \cdot m$  mögliche Ausprägungen durch  $n + m$  Bitmap-Vektoren zu indexieren
- jeder Wert  $x (0 \leq x \leq n \cdot m - 1)$  kann durch zwei Werte  $y$  und  $z$  repräsentiert werden:

$$x = n \cdot y + z \text{ mit } 0 \leq y \leq m - 1 \text{ und } 0 \leq z \leq n - 1$$

- ◆ dann nur noch maximal  $m$  Bitmap-Vektoren für  $y$  und  $n$  Bitmap-Vektoren für  $z$
- ◆ Speicheraufwand reduziert sich von  $n \cdot m$  auf  $n + m$
- ◆ dafür müssen für eine Punktanfrage aber 2 Bitmap-Vektoren gelesen werden

# Mehrkomponenten-Bitmap-Index

---

- Beispiel: Zweikomponenten-Bitmap-Index
- Für  $M = 0..11$  etwa  $x = 4 \cdot y + z$
- y-Werte: B-2-1, B-1-1, B-0-1
- z-Werte: B-3-0, B-2-0, B-1-0, B-0-0

x	y			z			
M	B-2-1	B-1-1	B-0-1	B-3-0	B-2-0	B-1-0	B-0-0
5	0	1	0	0	0	1	0
3	0	0	1	1	0	0	0
0	0	0	1	0	0	0	1
11	1	0	0	1	0	0	0

# Beispiel: Postleitzahlen

---

- Kodierung von Postleitzahlen
- Werte 00000 bis 99999
- direkte Umsetzung: 100.000 Spalten
- Zweikomponenten-Bitmap-Index (erste 2 Ziffern + 3 letzte Ziffern): 1.100 Spalten
- Fünf Komponenten: *50 Spalten*
  - ◆ geeignet für Bereichsanfragen „PLZ 39\*\*\*“
- Binärkodiert (bis  $2^{17}$ ): 34 Spalten
  - ◆ nur für Punktanfragen!
- *Bemerkung: Kodierung zur Basis 3 ergibt sogar nur 33 Spalten....*

# Mehrdimensionale Speichertechniken

---

- bisher: eindimensional (keine partial-match-Anfragen, nur lineare Ordnung)
- jetzt: mehrdimensional (auch partial-match-Anfragen, Positionierung im mehrdimensionalen Datenraum)
- $k$  Dimensionen =  $k$  Attribute können gleichberechtigt unterstützt werden
- dieser Abschnitt
  - ◆ mehrdimensionaler B-Baum
  - ◆ mehrdimensionales Hashverfahren
  - ◆ Grid-Files
- weitere mehrdimensionale Verfahren für Multimedia- und Geo-Daten im nächsten Kapitel

# Mehrdimensionale Baumverfahren

---

*KdB-Baum* ist  $B^+$ -Baum, bei dem Indexseiten als binäre Bäume mit Zugriffsattributen, Zugriffsattributwerten und Zeigern realisiert werden

Varianten von  $k$ -dimensionalen Indexbäumen:

- *kd-Baum* von Bentley und Friedman: für Hauptspeicheralgorithmen entwickelte, mehrdimensionale Grundstruktur (binärer Baum)
- *KDB-Baum* von Robinson: Kombination *kd*-Baum und B-Baum ( $k$ -dimensionaler Indexbaum bekommt höheren Verzweigungsgrad)
- *KdB-Baum* von Kuchen: Verbesserung des Robinson-Vorschlags, wird hier behandelt

# Mehrdimensionale Baumverfahren (II)

---

- KdB-Baum kann Primär- und mehrere Sekundärschlüssel gleichzeitig unterstützen
- macht als Dateiorganisationsform zusätzliche Sekundärindexe überflüssig

# Definition KdB-Baum

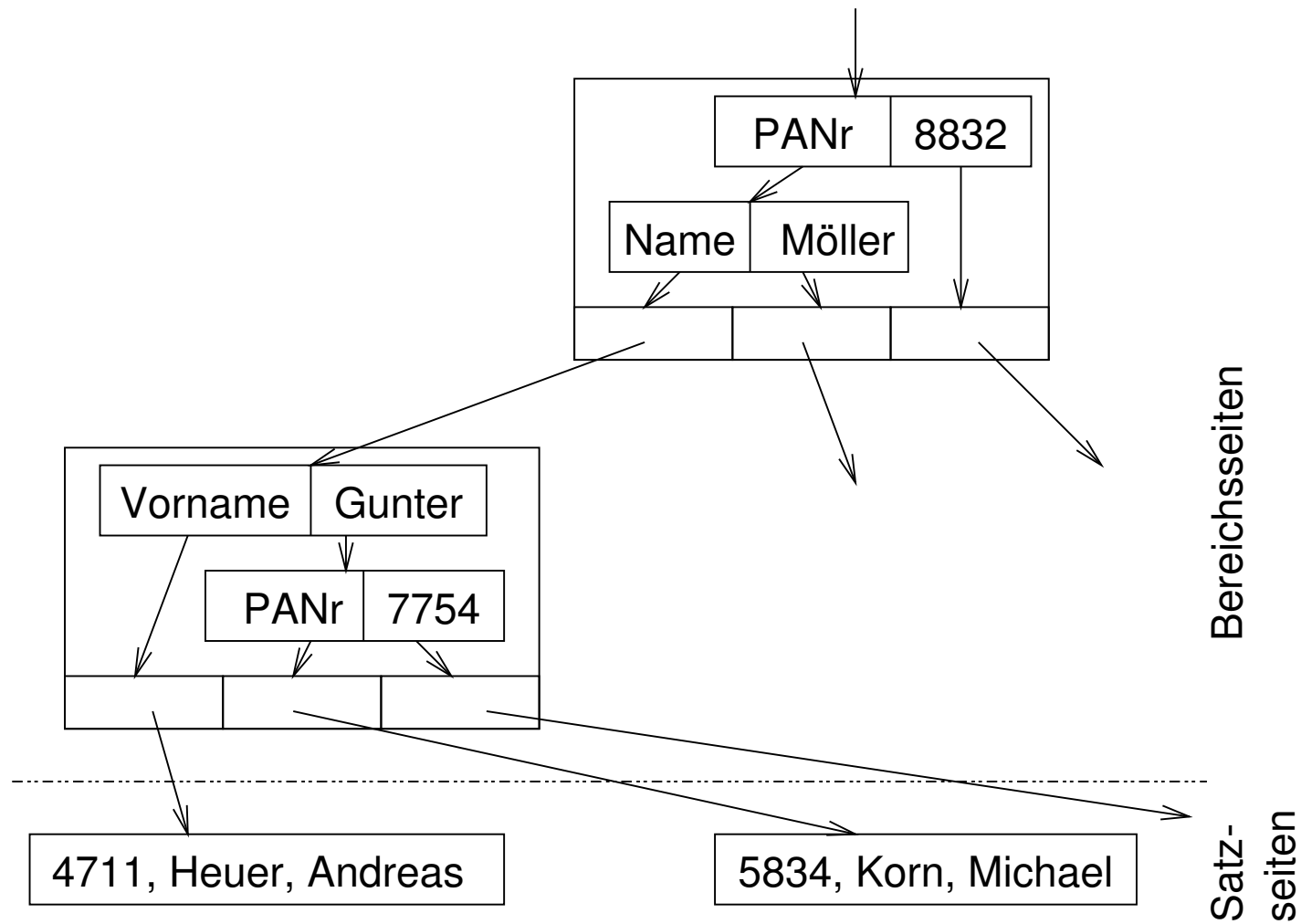
---

Idee: auf jeder Indexseite einen Teilbaum darstellen, der nach mehreren Attributen hintereinander verzweigt

- *KdB-Baum vom Typ  $(b, t)$  besteht aus*
  - ◆ *inneren Knoten (Bereichsseiten) die einen kd-Baum mit maximal  $b$  internen Knoten enthalten*
  - ◆ *Blättern (Satzseiten) die bis zu  $t$  Tupel der gespeicherten Relation speichern können*
- *Bereichsseiten: kd-Baum enthalten mit Schnittelementen und zwei Zeigern*
  - ◆ *Schnittelement enthält Zugriffsattribut und Zugriffsattributwert; linker Zeiger: kleinere Zugriffsattributwerte; rechter Zeiger: größere Zugriffsattributwerte*



# Beispiel



# KdB-Baum: Struktur

---

- Bereichsseiten
  - ◆ Anzahl der Schnitt- und Adressenelemente der Seite
  - ◆ Zeiger auf Wurzel des enthaltenen kd-Baumes
  - ◆ *Schnitt- und Adressenelemente.*
- Schnittelement
  - ◆ Zugriffsattribut
  - ◆ Zugriffsattributwert
  - ◆ zwei Zeiger auf Nachfolgerknoten des kd-Baumes dieser Seite (können Schnitt- oder Adressenelemente sein)
- Adressenelemente: Adresse eines Nachfolgers der Bereichsseite im KdB-Baum (Bereichs- oder Satzseite)

# KdB-Baum: Operationen

---

- Komplexität **lookup**, **insert** und **delete** bei *exact-match*  $O(\log n)$
- bei *partial-match* besser als  $O(n)$
- bei  $t$  von  $k$  Attributen in der Anfrage spezifiziert: Zugriffskomplexität von  $O(n^{1-t/k})$

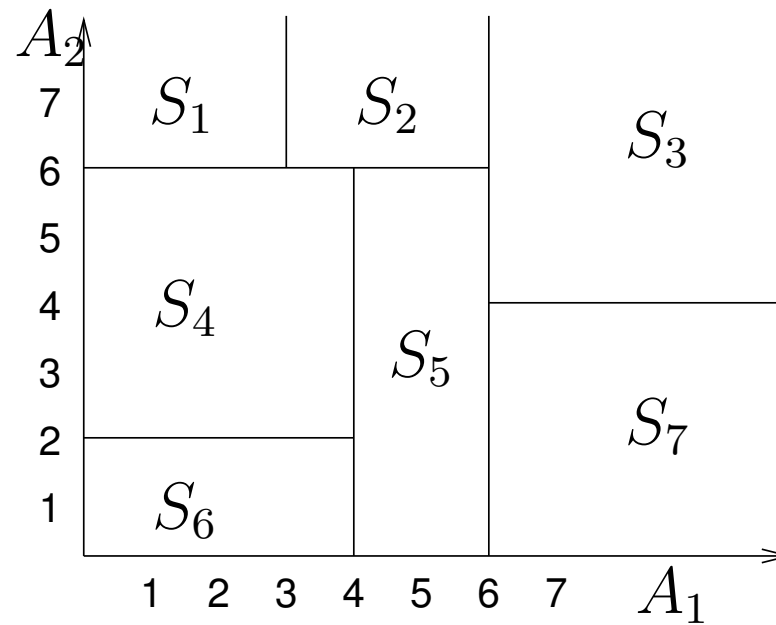
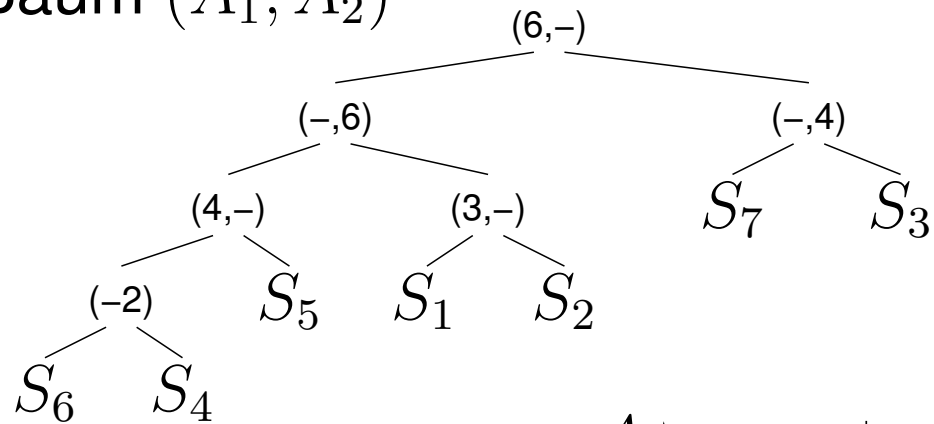
# KdB-Baum: Trennattribute

---

- Reihenfolge der Trennattribute
  - ◆ entweder zyklisch festgelegt
  - ◆ oder Selektivitäten einbeziehen: Zugriffsattribut mit hoher Selektivität sollte früher und häufiger als Schnittelement eingesetzt werden
- Trennattributwert: aufgrund von Informationen über Verteilung von Attributwerten eine geeignete „Mitte“ eines aufzutrennenden Attributwertebereichs ermitteln

# KdB-Baum: Brickwall

2d-Baum ( $A_1, A_2$ )



# Mehrdimensionales Hashen

---

- Idee: Bit Interleaving
- abwechselnd von verschiedenen Zugriffsattributwerten die Bits der Adresse berechnen
- Beispiel: zwei Dimensionen

	*0*0	*0*1	*1*0	*1*1
0*0*	0000	0001	0100	0101
0*1*	0010	0011	0110	0111
1*0*	1000	1001	1100	1101
1*1*	1010	1011	1110	1111

# MDH von Kuchen

---

## Idee

- MDH baut auf linearem Hashen auf
- Hash-Werte sind Bit-Folgen, von denen jeweils ein Anfangsstück als aktueller Hash-Wert dient
- je ein Bit-String pro beteiligtem Attribut berechnen
- Anfangsstücke nun nach dem Prinzip des Bit-Interleaving zyklisch abarbeiten
- Hash-Wert reihum aus den Bits der Einzelwerte zusammensetzen

# MDH formal (I)

---

- mehrdimensionaler Wert  $x$

$$x := (x_1, \dots, x_k) \in D = D_1 \times \dots \times D_k$$

- Folge von mit  $i$  indizierten Hash-Funktionen konstruieren
- $i$ -te Hash-Funktion  $h_i(x)$  wird mittels Kompositionsfunktion  $\bar{h}_i$  aus den jeweiligen  $i$ -ten Anfangsstücken der lokalen Hash-Werte  $h_{i_j}(x_j)$  zusammengesetzt:  $h_i(x) = \bar{h}_i(h_{i_1}(x_1), \dots, h_{i_k}(x_k))$
- lokale Hash-Funktionen  $h_{i_j}$  ergeben jeweils Bit-Vektor der Länge  $z_{i_j}$ :

$$h_{i_j} : D_j \rightarrow \{0, \dots, z_{i_j}\}, j \in \{1, \dots, k\}$$



# MDH formal (II)

---

- $z_{i_j}$  sollten möglichst gleich groß sein, um die Dimensionen gleichmäßig zu berücksichtigen
- Kompositionsfunktion  $\bar{h}_i$  setzt lokale Bitvektoren zu einem Bitvektor der Länge  $i$  zusammen:

$$\bar{h}_i : \{0, \dots, z_{i_1}\} \times \dots \times \{0, \dots, z_{i_k}\} \rightarrow \{0, \dots, 2^{i+1} - 1\}$$

# MDH formal (III)

---

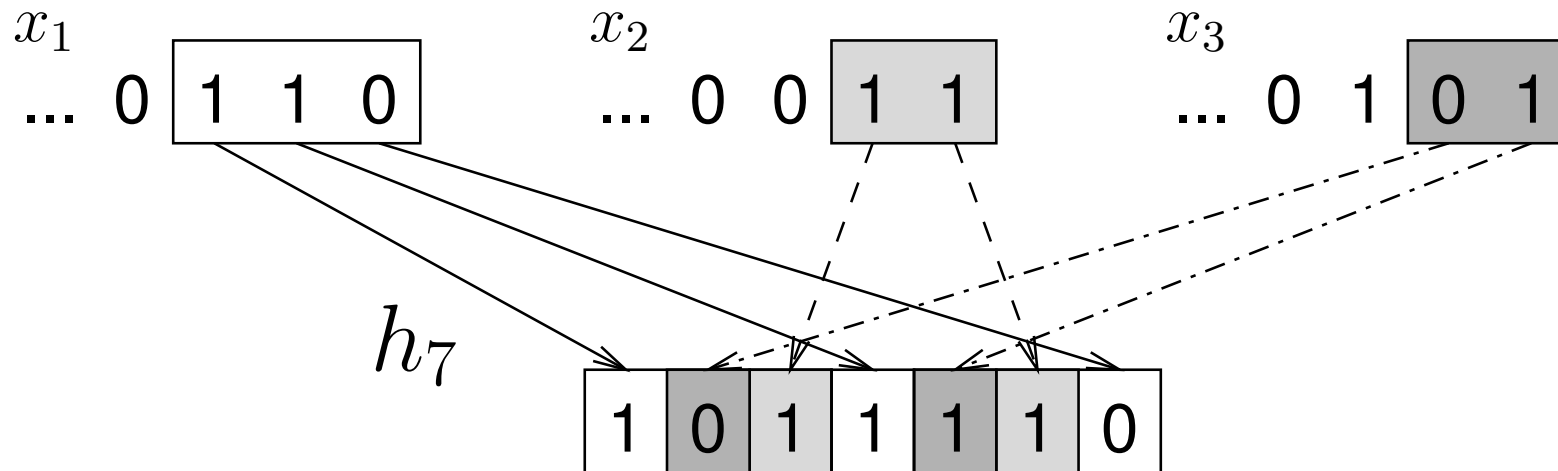
- ausgeglichene Länge der  $z_{i_j}$  wird durch folgende Festlegung bestimmt, die Längen zyklisch bei jedem Erweiterungsschritt an einer Stelle erhöht:

$$z_{i_j} = \begin{cases} 2^{\lfloor \frac{i}{k} \rfloor + 1} - 1 & \text{für } j - 1 \leq (i \bmod k) \\ 2^{\lfloor \frac{i}{k} \rfloor} - 1 & \text{für } j - 1 > (i \bmod k) \end{cases}$$

- Kompositionsfunktion:

$$\bar{h}_i(x) = \sum_{r=0}^i \left( \frac{(x_{(r \bmod k)+1} \bmod 2^{\lfloor \frac{r}{k} \rfloor + 1}) - (x_{(r \bmod k)+1} \bmod 2^{\lfloor \frac{r}{k} \rfloor})}{2^{\lfloor \frac{r}{k} \rfloor}} \right) 2^r$$

# MDH Veranschaulichung



- verdeutlicht Komposition der Hash-Funktion  $h_i$  für drei Dimensionen und den Wert  $i = 7$
- graphisch unterlegte Teile der Bit-Strings entsprechen den Werten  $h_{7_1}(x_1)$ ,  $h_{7_2}(x_2)$  und  $h_{7_3}(x_3)$
- beim Schritt auf  $i = 8$  würde ein weiteres Bit von  $x_2$  (genauer: von  $h_{8_2}(x_2)$ )) verwendet

# MDH Komplexität

---

- Exact-Match-Anfragen:  $O(1)$
- Partial-Match-Anfragen, bei  $t$  von  $k$  Attributen festgelegt, Aufwand  $O(n^{1-\frac{t}{k}})$
- ergibt sich aus der Zahl der Seiten, wenn bestimmte Bits „unknown“
- Spezialfälle:  $O(1)$  für  $t = k$ ,  $O(n)$  für  $t = 0$

# Grid-Files

---

- bekannteste und von der Technik her attraktivste mehrdimensionale Dateiorganisationsform
- eigene Kategorie: Elemente der Schlüsseltransformation wie bei Hash-Verfahren und Indexdateien wie bei Baumverfahren kombiniert
- mehrdimensionaler Raum sehr „gleichmäßig“ aufgeteilt (im Gegensatz zu Brickwall)

# Grid-File: Zielsetzungen

---

## Hinrichs und Nievergelt

- Prinzip der 2 Plattenzugriffe: Jeder Datensatz soll bei einer *exact-match*-Anfrage in 2 Zugriffen erreichbar sein
- Zerlegung des Datenraums in Quader:  $n$ -dimensionale Quader bilden die Suchregionen im Grid-File
- Prinzip der Nachbarschaftserhaltung: Ähnliche Objekte sollten auf der gleichen Seite gespeichert werden
- Symmetrische Behandlung aller Raum-Dimensionen: *partial-match*-Anfragen ermöglicht
- Dynamische Anpassung der Grid-Struktur beim Löschen und Einfügen

# Prinzip der zwei Plattenzugriffe

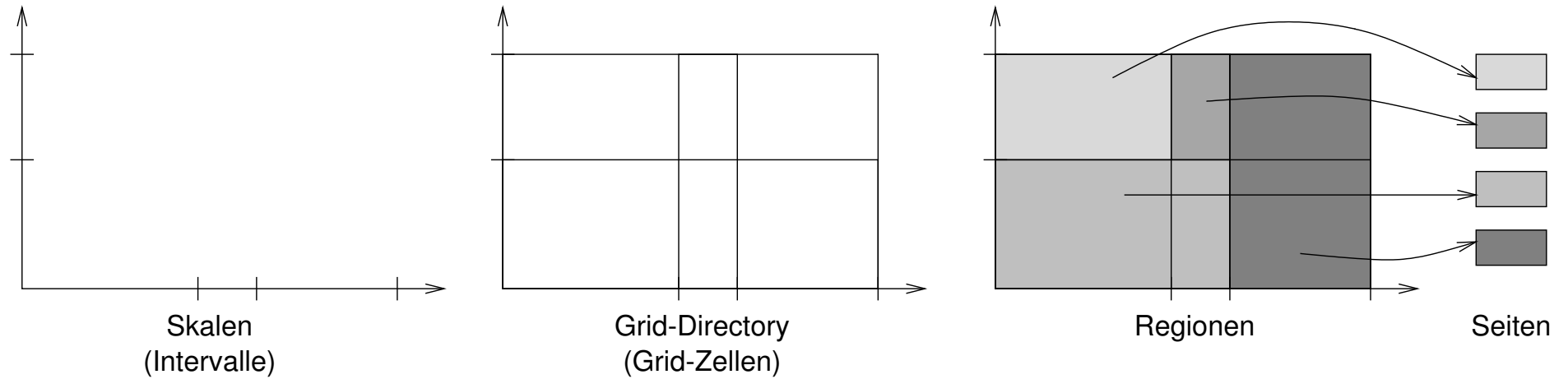
---

Bei exact-match

1. gesuchtes  $k$ -Tupel auf Intervalle der *Skalen* abbilden; als Kombination der ermittelten Intervalle werden Indexwerte errechnet; Skalen im Hauptspeicher  $\Rightarrow$  noch kein Plattenzugriff
2. über errechnete Indexwerte Zugriff auf das *Grid-Directory*; dort Adressen der Datensatz-Seiten gespeichert; erster *Plattenzugriff*.
3. Der Datensatz-Zugriff: zweiter *Plattenzugriff*.

# Aufbau eines Grid-Files (I)

---





# Aufbau eines Grid-Files (II)

---

- *Grid*:  $k$  eindimensionale Felder (Skalen), jede Skala repräsentiert Attribut
- *Skalen* bestehen aus Partition der zugeordneten Wertebereiche in *Intervalle*
- *Grid-Directory* besteht aus Grid-Zellen, die den Datenraum in Quader zerlegen
- *Grid-Zellen* bilden eine Grid-Region, der genau eine Datensatz-Seite zugeordnet wird
- *Grid-Region*:  $k$ -dimensionales, konvexes (Regionen sind paarweise disjunkt)

# Operationen

---

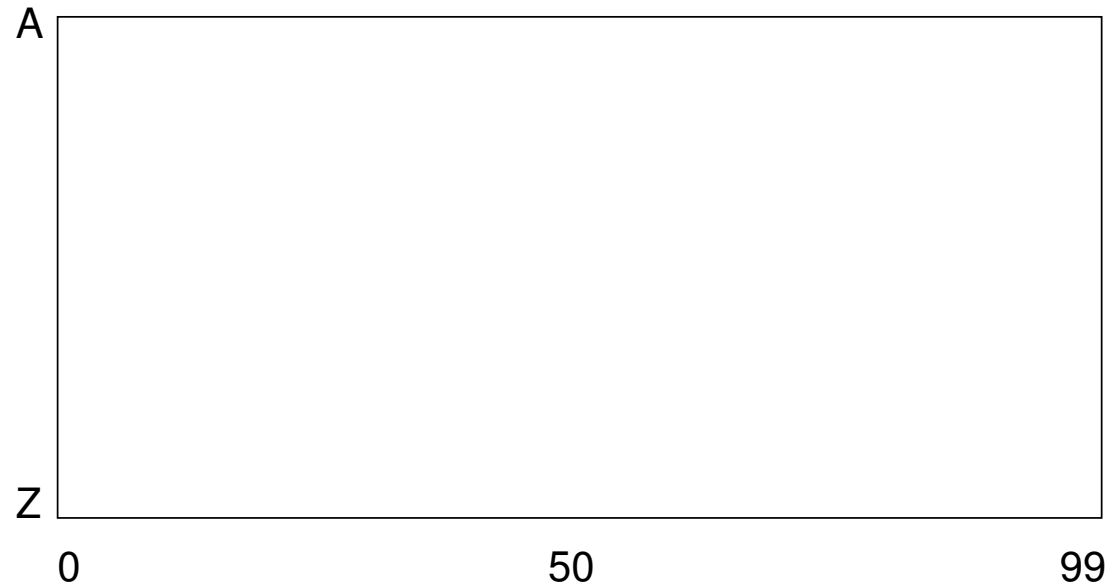
Zu Anfang: Zelle = Region = eine Datensatz-Seite

- *Seitenüberlauf*: Seite wird geteilt. Falls die zur Seite gehörende Grid-Region aus nur einer Grid-Zelle besteht, muß ein Intervall auf einer Skala in zwei Intervalle unterteilt werden. Besteht die Region aus mehreren Zellen, so werden diese Zellen in einzelne Regionen zerlegt.
- *Seitenunterlauf*: Zwei Regionen zu einer zusammenfassen, falls das Ergebnis eine neue, konvexe Region ergibt.

# Beispiel

---

- Start-Grid-File



- Datensätze einfügen: (45, D), (2, A), (87, S), (75, M), (55, K), (3, Z), (15, D), (25, K), (48, F)
- jede Seite des Grid-Files faßt bis zu drei Datensätze

# Buddy-System

---

- Beschriebenes Verfahren: *Buddy-System* (Zwillings-System)
- Die im gleichen Schritt entstandenen Zellen können zu Regionen zusammengefaßt werden; Keine andere Zusammenfassung von Zellen ist im Buddy-System erlaubt
- Unflexibel beim Löschen: nur Zusammenfassungen von Regionen erlaubt, die vorher als Zwillinge entstanden waren
- Beispiel: (15,D) löschen: Seiten 1 und 4 zusammenfassen; (87,S) löschen, Seite 2 zwar unterbelegt, kann aber mit keiner anderen Seite zusammengefasst werden

# Cluster-Bildung

---

- gemeinsame Speicherung von Datensätzen auf Seiten
- wichtige Spezialfälle:
  - ◆ *Ballung nach Schlüsselattributen.*  
Bereichsanfragen und Gruppierungen unterstützen:  
Datensätze in der Sortierreihenfolge  
zusammenhängend auf Seiten speichern  $\Rightarrow$   
*index-organisierte Tabellen* oder geclusterten,  
dichtbesetzte Primärindexe
  - ◆ *Ballung basierend auf Fremdschlüsselattributen.*  
Gruppen von Datensätzen, die einen Attributwert  
gemeinsam haben, werden auf Seiten geballt  
(Verbundanfragen)
- Komponentenbeziehungen statt Verbundattribute in  
OODBS

# Indexorganisierte Tabellen

---

- Tupel direkt im Index aufnehmen
- allerdings dann durch häufigen Split TID unsinnig
- weiterer Sekundärindex kann durch fehlenden TID dann aber nicht angelegt werden
- etwa kein **unique** möglich

# Cluster für Verbundanfragen

## Verbundattribut: Cluster-Schlüssel

Auftragsnr

100	Auftragsdatum	Kunde	Lieferdatum	
	15.04.98	Orion Enterprises	01.01.2001	
	Position	Teil	Anzahl	Preis
	1	Aluminiumtorso	2	3145,67
	2	Antenne	2	32,50
	3	Overkill	1	1313,45
	4	Nieten	1000	-.50

Auftragsnr

123	Auftragsdatum	Kunde	Lieferdatum	
	05.10.98	Kirk Enterpr.	31.12.1999	
	Position	Teil	Anzahl	Preis
	1	Beamer	1	13145,67
	2	Energiekristall	2	32,99
	3	Phaser	5	1313,45
	4	Nieten	2000	-.50

# Definition von Clustern

---

```
create cluster AuftragCluster
  (Auftragsnr number(3))
  pctused 80 pctfree 5;

create table T_Auftrag (
  Auftragsnr number(3) primary key, ...)
  cluster AuftragCluster (Auftragsnr);

create table T_Auftragspositionen (
  Position number(3),
  Auftragsnr number(3) references T_Auftrag,
  ...
  constraint AuftragPosKey
    primary key (Position, Auftragsnr)
  )
  cluster AuftragCluster (Auftragsnr);
```



# Organisation von Clustern

---

- *Indexierte Cluster* nutzen einen in Sortierreihenfolge aufgebauten Index (z.B. B<sup>+</sup>-Baum) über den Cluster-Schlüssel zum Zugriff auf die Cluster
- *Hash-Cluster* bestimmen den passenden Cluster mit Hilfe einer Hash-Funktion
- Indexe für Cluster entsprechen normalen Indexen für den Cluster-Schlüssel
- statt Tupelidentifikatoren Einsatz von Cluster-Identifikatoren oder direkte Speicheradressen (bei Hash-Verfahren)

# Indexierte Cluster

---

```
create index AuftragClusterIndex  
on cluster AuftragCluster
```

# Hash-Cluster

---

```
create cluster AuftragCluster (  
    Auftragsnr number(5,0))  
    pctused 80  
    pctfree 5  
    size 2k  
    hash is Auftragsnr  
    hashkeys 100000;
```

# Physische Datendefinition in SQL

---

	Ingres	Oracle	DB2	Informix
sequentiell, Heap	+	+	+	+
index-seq. (dünn)	+	-	-	-
indexiert-nichtseq. (dicht)	+	-	-	-
mehrstufig	+	-	-	-
B-Baum	-	-	-	-
B <sup>+</sup> -Baum (dicht)	+	+	+	+
KdB-Baum	-	-	-	-
Hash	+	(+)	-	-
MDH	-	-	-	-
Cluster	-	+	+	-

# Ingres

---

```
create [ unique ] index indexname  
      on relname (  
attrname [ asc | desc ],  
      ...  
) [ with-klausel ];
```

Angaben zur physischen Speicherung in **with**-Klausel

```
structure = cbtree | btree | cisam | isam  
          | chash | hash,  
key = ( attrname, ... ),  
minpages = n, maxpages = n, fillfactor = n,  
leaffill = n, noleaffill = n,  
location (location, ...)
```

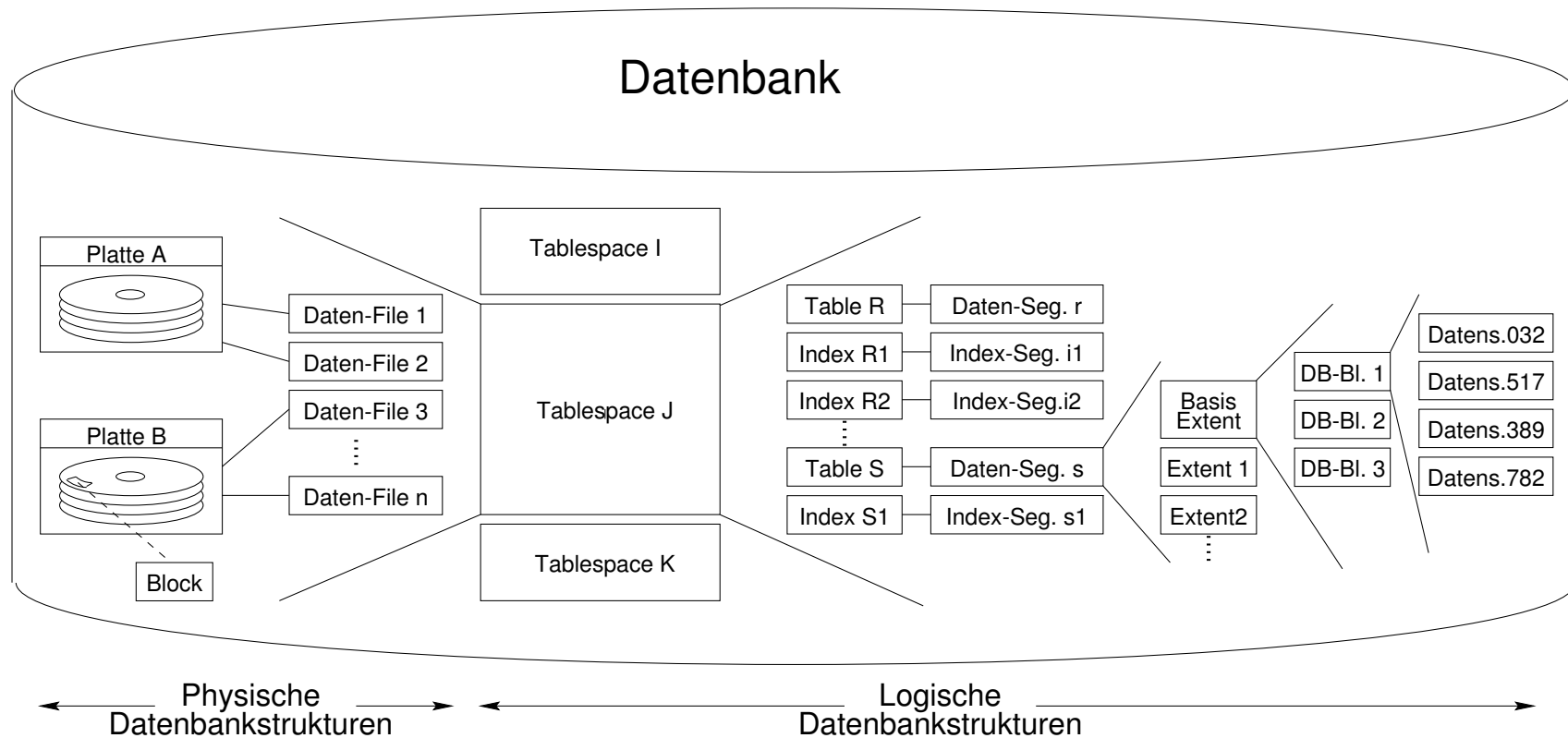
# Ingres: modify

---

```
modify rel-or-index-name  
to storage-structure  
[ on attr-name [asc|desc]  
{, attr-name [asc|desc] } ]  
with-clause
```

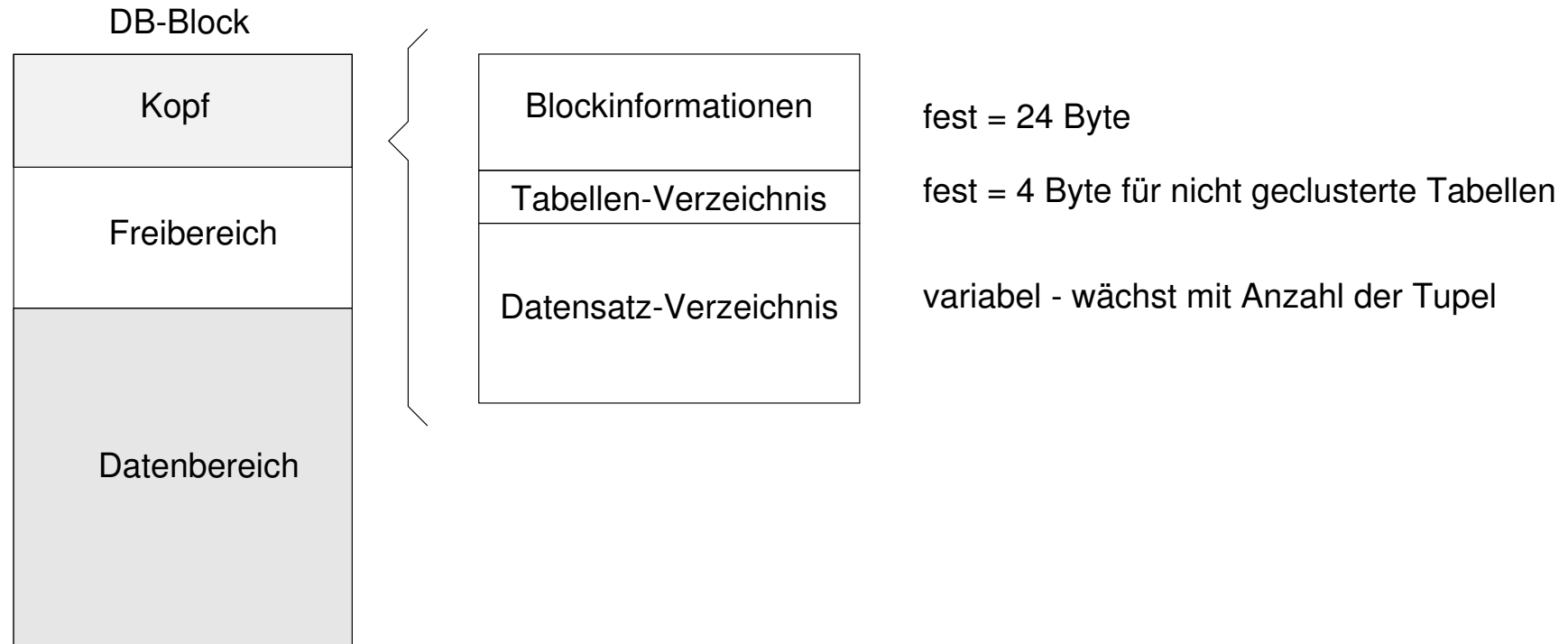
- daneben komprimierte Versionen zu allen Organisationsformen
- insbesondere bei zu indexierenden Zeichenketten sinnvoll: (CHeap, CHash, CBtree, ...)
- Adressierung der Sätze erfolgt über das TID-Konzept

# Oracle



# Oracle: Blöcke

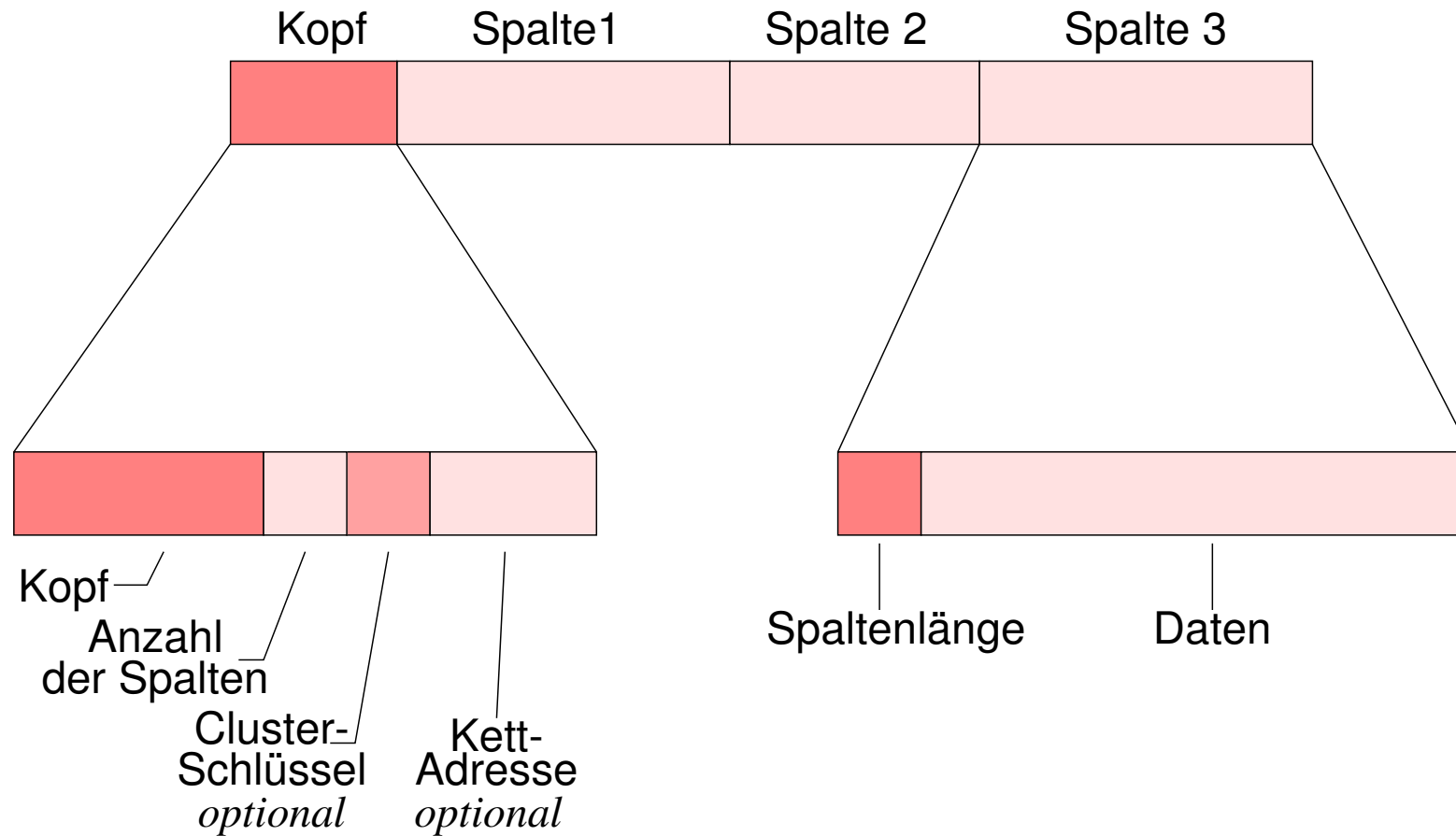
---





# Oracle: Datensätze

---



# Oracle: Datenorganisation

---

- Der Standard-Index ist als  $B^+$ -Baum aufgebaut
- Indexorganisierte Tabellen speichern Tupel direkt in den Blättern eines  $B^+$ -Baums
- Clusterung mehrerer Relationen möglich; Cluster-Indexe können als  $B^+$ -Baum oder als Hash-Index organisiert werden.
- Bitmap-Indexe speichern Bit-Matrizen für Aufzählungsattribute (siehe Data Warehouses)
- Reverse Indexe interpretieren die Bytes des Primärschlüssels in umgekehrter Reihenfolge und brechen damit die Speicherung in Sortierreihenfolge auf

# Informix

---

- **dbspace** (mehrere Chunks – Rohdatei) und **blobspace**
- Chunks - Extents - Seite (Timestamp, TID-Feld)
- Seiten innerhalb eine Extents (garantiert geclustert)
  - ◆ *Bitmap Page*: Verzeichnis aller Seiten im Extent
  - ◆ *Data Page*: eigentliche Daten aus den Tabellen
  - ◆ *Remainder Page*: Spannsätze mit Überlaufseiten
  - ◆ *Index Page*: Indexdaten
  - ◆ *Blob Page*: BLOBs mit Puffer- und Logging-Mechanismen (im Originalsatz Verweis auf Startseite einer Liste von BLOB-Seiten)
  - ◆ *Free Page*: freie Seiten in diesem Extent
- **tablespace**: mehrere Extents

# Informix: Datendefinition

---

- Indexstruktur: B<sup>+</sup>-Baum

```
create [ unique | distinct ] [ cluster ]  
      index indexname on relname (  
    attrname [asc | desc ],  
    attrname [asc | desc ], ...)  
      [ using indexart, ]  
      [ fillfactor = prozent ];
```

- **cluster**-Option: geclusterter Index (nur bei Neuanlage wird Eigenschaft zugesichert)
- **using**: „B-tree“ für B<sup>+</sup>-Baum, „R-tree“
- **fillfactor**: initial 90%
- statt Attribute auch Methoden indexierbar

Sekundärindex: B<sup>+</sup>-Baum

- geclustert und dichtbesetzt **clustered**
- nicht-geclustert und dichtbesetzt **non-clustered**

# DB2 (II)

---

## tablespace

- Menge von *Containern*
- ein Container: Betriebssystem-Verzeichnis, Betriebssystem-Datei, Sekundärspeicher-Medium
- **tablespace**: für mehrere Tabellen (physisches Clustern)
- mehrere **tablespaces** pro Tabelle: (Indexdaten, BLOB-Daten, ...)
- Allokationsgranulat: *Extents*
- Betriebssystem-verwaltet (SMS, System Managed Space)
- DB2-verwaltet (DMS, Database Managed Space)

# DB2: bufferpool

---

**tablespace** im Hauptspeicher per Default ein **bufferpool**

- Veränderungen der Größe des **bufferpools**
- Anlegen mehrerer **bufferpools** für einen **tablespace**
- **prefetchsize**: Anzahl der Seiten, die bei einem *page fault* zu holen sind

## DB2: reorg

---

- Daten einer Tabelle wieder in zusammenhängenden Speicherbereichen speichern
- Sortierung einer Datei neu festlegen
- Überlaufseiten nach Vergrößerungen von Datensätzen eliminieren (zweistufiges TID-Konzept)



# 5. Spezielle Zugriffsstrukturen

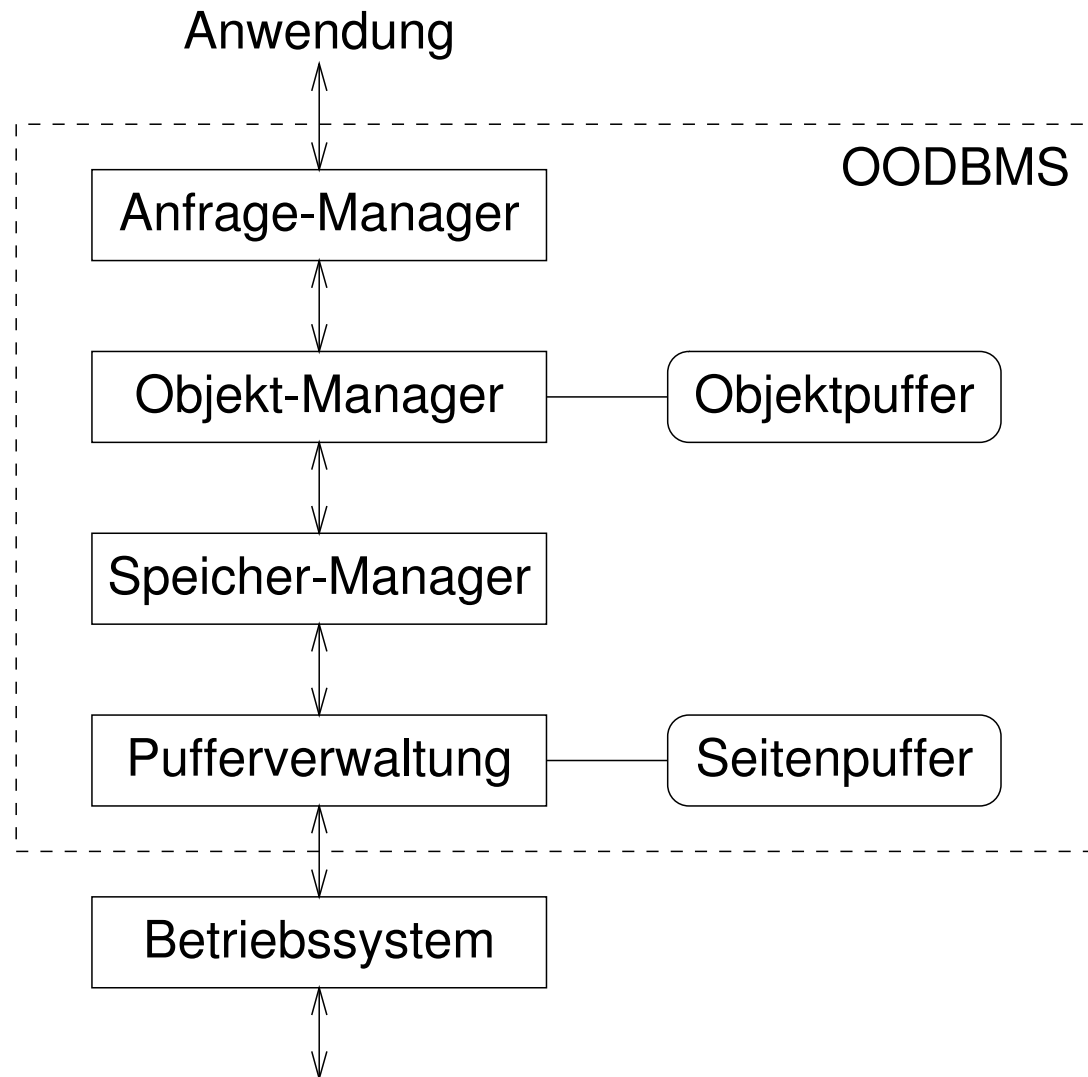
---

Zugriffsstrukturen für spezielle Anwendungen

- Objektorientierte Datenbanken
- Geometrische Zugriffsstrukturen
- Zugriffsverfahren für Multimedia- und Text-Daten

# Objektorientierte Datenbanken

## ■ Aufbau eines OODBMS



# Objektidentität

---

- Varianten
  - ◆ *Darstellung der Objektidentität durch Surrogate und ihre Implementierung durch indirekte Referenzen: (GemStone, ORION/ITASCA, Postgres)*
  - ◆ *Darstellung und Implementierung der Objektidentität durch direkte Referenzen: (ONTOS, ObjectStore)*
- erste Technik logisch sauberer, aber langsamer

# Objektidentität (II)

---

- Surrogate:
  - ◆ immer *eindeutig*, auch nach der Löschung des Objektes
  - ◆ in *verteilter Umgebung* eindeutig (Codierung der Rechner-ID im Surrogat)
  - ◆ *abstrakte Klasse*, in der das Objekt erzeugt wird, im Surrogat kenntlich machen
- Länge: 32 oder 64 Bit

# Klassen

---

(ohne komplexe Attributwerte, Komponentenobjekte)

- *Binäre Speicherung*  
Objekte zusammen mit jeweils einem Attribut als binäre Relation
- *Objektstruktur mit integriertem Schema*  
Schemainformationen in die Speicherstruktur jedes Objekts (etwa in ORION/ITASCA)
- *Objektstruktur mit externem Schema*

# Klassen (II)

---

## ■ Objektstruktur mit integriertem Schema

Surrogat		Bytes	Anz_Attribute	Attribute	
011001010		38	3	Arbeitsstelle	...
Attribute			Werte-Offsets		
...	Gehalt	Vorgesetzter	0	10	...
Attributwerte					
...	14	Informatik	3050	1001101	

## ■ Objektstruktur mit externem Schema

Surrogat	Bytes	Attributwerte		
011001010	10	Informatik	3050	10001101

# Komplexe Attribute

---

- Prinzip
  - ◆ Objekt größer als Seite  $\Rightarrow$  mehrere Seiten in Form eines B-Baums (etwa EXODUS)
  - ◆ Menge von Objekten einer Klasse geclustert
- *komplexe Attribute*
  - ◆ *zerlegte Speicherung* (wie in RDBS normalisiert)  
Iris, OSCAR
  - ◆ gesamte Objektstruktur in einem *Cluster*  
DASDBS
- *private Komponentenobjekte*: Cluster möglich
- *gemeinsame Komponentenobjekte*: Referenzen auf Komponentenobjekt

# Cluster

---

- Cluster-Definition zur Zeit der
  - ◆ *Klassendefinition (im Schema):*  $O_2$
  - ◆ *Objektinstanziierung (pro Objekt):* ObjectStore, ObServer, ONTOS und GemStone
- Cluster-Strukturen für
  - ◆ alle Objekte einer Klasse
  - ◆ bestimmte Teile von Klassen, etwa Partition der Klasse nach bestimmten Attributwerten
  - ◆ alle Instanzen von Klassen, die zu einem spezifizierten Teil der Klassenhierarchie gehören,
  - ◆ zusammengesetztes Objekt (Objekt mit Komponentenobjekten)
  - ◆ komplexe Attributwerte



# Klassenhierarchien

---

- Objekt nur in genau einer Klasse:
  - ◆ Zustand dieses Objektes in dieser Klasse gespeichert (*Home Class Model*) (bei OODBPLs und einigen Neuentwicklungen wie ORION)

# Klassenhierarchien (II)

---

- Objekt in mehreren Klassen
  - ◆ Objekt in kleinster Klasse, zusammen mit vererbten Attributwerten (*Leaf Overlap Model*) (bei ORDBMS wie Illustra)
  - ◆ Objekt in jeder Klasse, zusammen mit lokalen Attributwerten (*Split Instance Model*) (OpenODB)
  - ◆ Objekt in jeder Klasse, zusammen mit dort definierten und allen vererbten Attributwerten (*Repeat Class Model*) (tiefe Extension direkt, etwa UniSQL)
  - ◆ Alle Objekte in einer Datei, nicht anwendbare Attribute auf **null** (*Universal Class Model*)
  - ◆ Alle Objekte in einer ternären Datei mit Surrogat, Attribut und Attributwert (*Value Triple Model*)

# Zugriffspfade für Klassen

---

- grundlegende Dateiorganisationsform durch Speicherstruktur der Klasse bereits festgelegt
- durch Hash-Funktionen oder B-Bäume zusätzlich unterstützen
- Zugriff auf Objekte in Klassenhierarchien und Komponentenhierarchien unterstützen
- RDBMS: Zugriffspfad nur eine Relation
- OODBS: Menge von Klassen durch einen Zugriffspfad unterstützen

# Zugriffspfade für Klassenhierarchien

---

- Index für Hierarchie von Klassen über einem Attribut einer (Ober-)Klasse  $K$ ; Verweis auf
  - ◆ alle Vorkommen der passenden Objekte in der Klassenhierarchie mit Wurzel  $K$ , wenn die Objekte nach der Split-Instance-Methode gespeichert sind
  - ◆ Vorkommen des Objektes in der Klassenhierarchie in den anderen Fällen, wobei das Objekt auch in einer der Unterklassen von  $K$  gespeichert sein kann
- *Klassenhierarchie-Index* (ORION/ITASCA,  $O_2$ )
- Beispiel: Personen, Angestellten und Studenten: Index Namen des Studenten

# Komponentenhierarchien

---

- Pfadausdrücke unterstützen; Attributwerte einer (auch indirekten) Komponentenkategorie gegeben

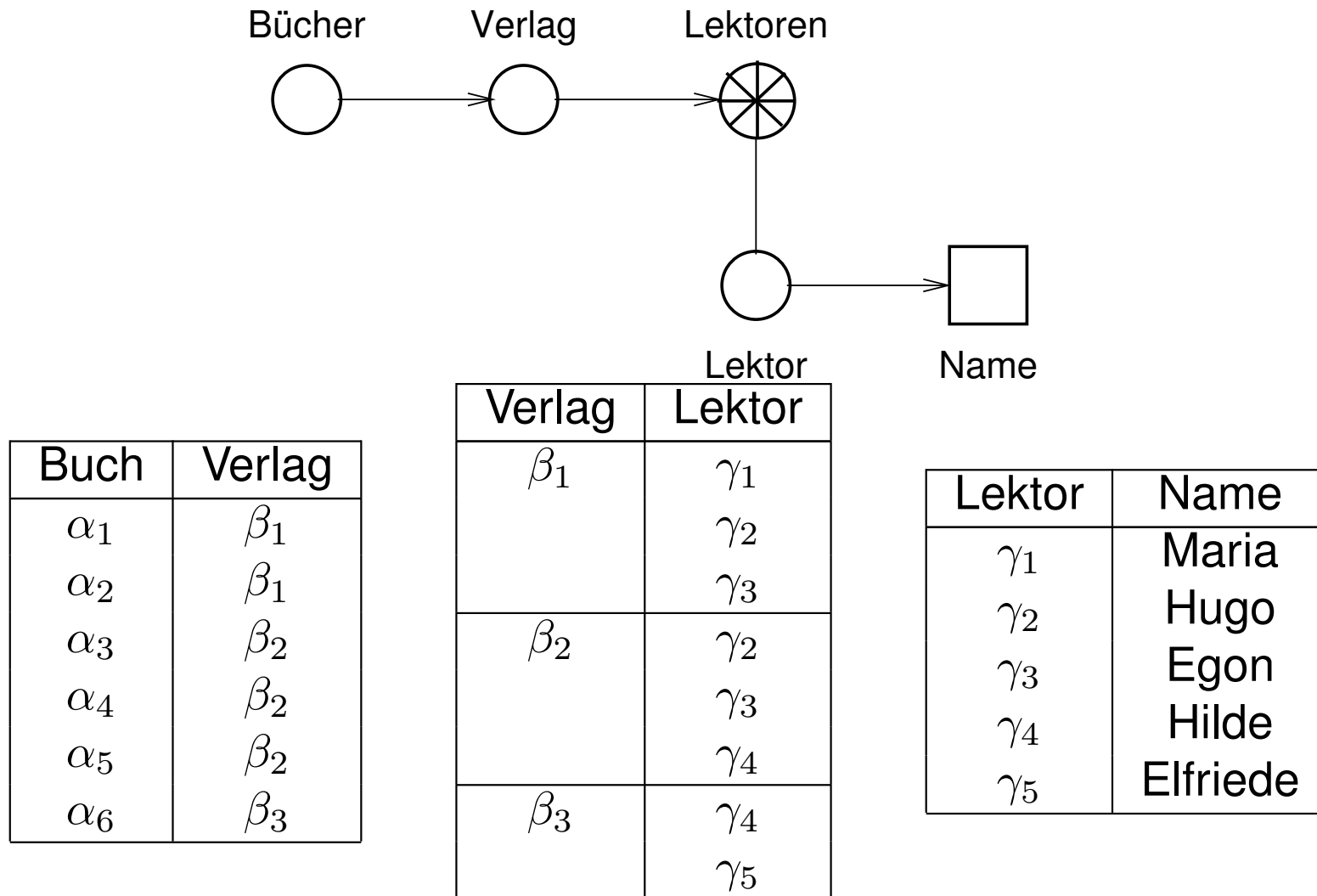
`Buch.Verlag.Verlagsort`

(Zugriff auf ein Buch über den Sitz des Verlages)

`Buch.Verlag.Lektor.Name`

(Zugriff auf ein Buch über den Namen des Lektors des Verlages)

# Komponentenhierarchien (II)



# equality und identity index

---

- *Unterstützung von Attributen, deren Typ ein Standard-Datentyp ist, analog zu den klassischen Zugriffspfaden in relationalen Systemen*

GemStone: *equality index* für Teilpfade

`Verlag.Verlagsort` **oder** `Lektor.Name`

- *Unterstützung von Komponentenobjekten (d.h. objektwertigen Attributen)*

GemStone: *identity index* für Teilpfade `Buch.Verlag`  
**oder** `Verlag.Lektor`

# Realisierungsformen: Übersicht

---

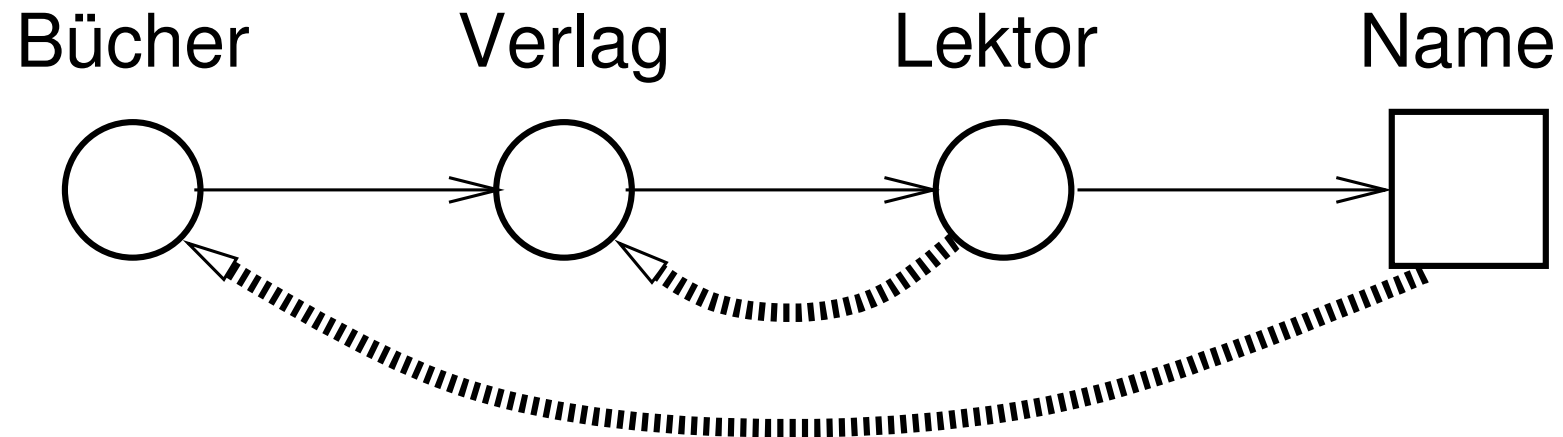
- *Pfadindex*
- *Multiindex*
- *Verbundindex*
- *geschachtelter Index*
- *Zugriffsunterstützungsrelation*

(Formen 1 und 2 in ORION und GemStone)

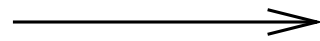


# Veranschaulichung: Indexgraph

---



Legende:



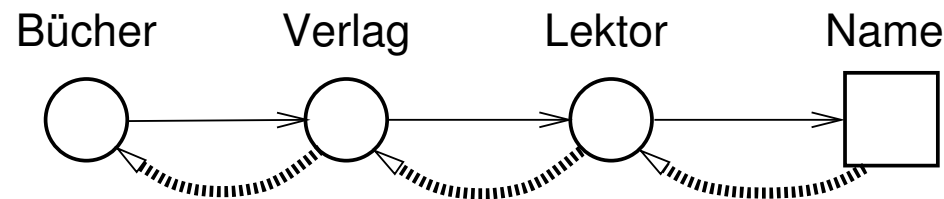
Komponentenbeziehung



Indexunterstützung

# Multiindex

- *Multiindex*: binäre Indexdateien von  $n$ -ter Komponente des Pfadausdrucks auf  $n - 1$ -te Komponente



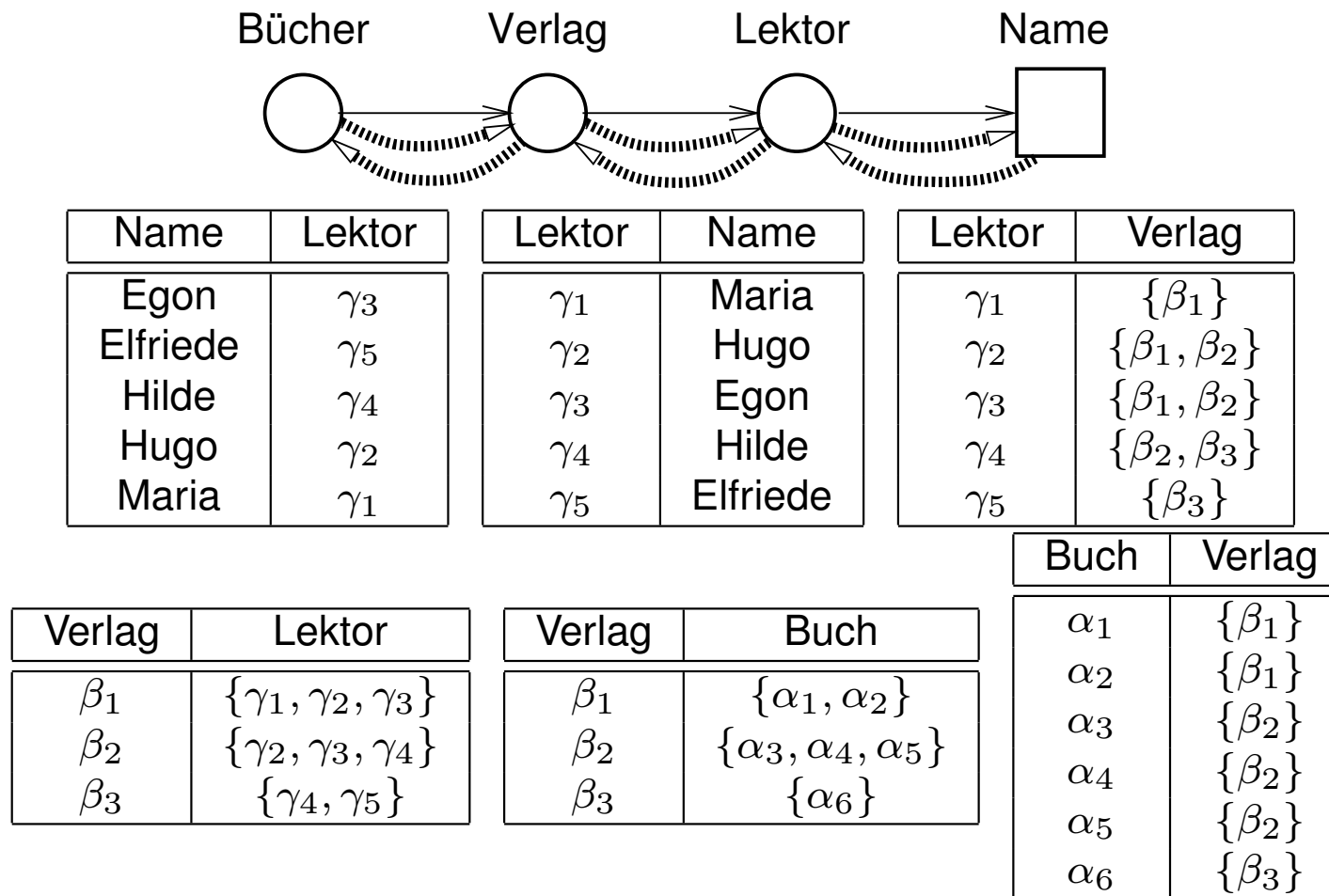
Name	Lektor
Egon	$\{\gamma_3\}$
Elfriede	$\{\gamma_5\}$
Hilde	$\{\gamma_4\}$
Hugo	$\{\gamma_2\}$
Maria	$\{\gamma_1\}$

Lektor	Verlag
$\gamma_1$	$\{\beta_1\}$
$\gamma_2$	$\{\beta_1, \beta_2\}$
$\gamma_3$	$\{\beta_1, \beta_2\}$
$\gamma_4$	$\{\beta_2, \beta_3\}$
$\gamma_5$	$\{\beta_3\}$

Verlag	Buch
$\beta_1$	$\{\alpha_1, \alpha_2\}$
$\beta_2$	$\{\alpha_3, \alpha_4, \alpha_5\}$
$\beta_3$	$\{\alpha_6\}$

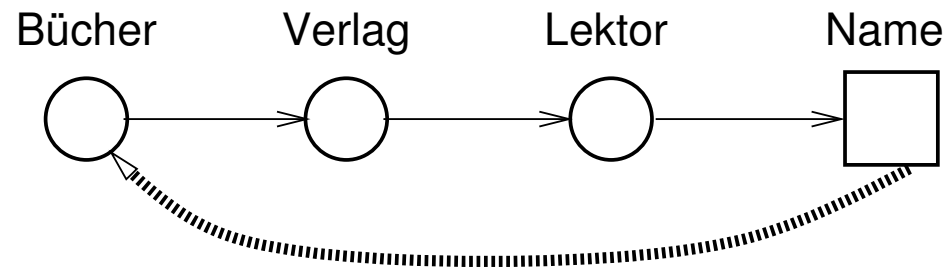
# Verbundindex

- *Verbundindex (join index)*: symmetrischer Multiindex



# Geschachtelter Index

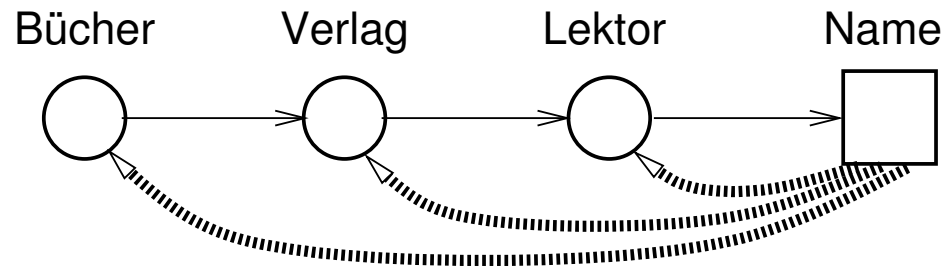
- *geschachtelter Index (nested index)*: eine einzige Indexdatei für  $n$ -te und erste Komponente des Pfadausdrucks



Name	Buch
Egon	$\{\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5\}$
Elfriede	$\{\alpha_6\}$
Hilde	$\{\alpha_3, \alpha_4, \alpha_5, \alpha_6\}$
Hugo	$\{\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5\}$
Maria	$\{\alpha_1, \alpha_2\}$

# Pfadindex

- *Pfadindex*: verallgemeinerter geschachtelter Index



Name	Lektor
Egon	$\{\gamma_3\}$
Elfriede	$\{\gamma_5\}$
Hilde	$\{\gamma_4\}$
Hugo	$\{\gamma_2\}$
Maria	$\{\gamma_1\}$

Name	Verlag
Egon	$\{\beta_1, \beta_2\}$
Elfriede	$\{\beta_3\}$
Hilde	$\{\beta_2, \beta_3\}$
Hugo	$\{\beta_1, \beta_2\}$
Maria	$\{\beta_1\}$

Name	Buch
Egon	$\{\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5\}$
Elfriede	$\{\alpha_6\}$
Hilde	$\{\alpha_3, \alpha_4, \alpha_5, \alpha_6\}$
Hugo	$\{\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5\}$
Maria	$\{\alpha_1, \alpha_2\}$

# Zugriffsunterstützungsrelation (ASR)

---

- *Zugriffsunterstützungsrelation (Access Support Relation, ASR)*
- Verallgemeinerung aller bisherigen Zugriffspfade, etwa verallgemeinerter (kompakter) Pfadindex

Name	Lektor	Verlag	Buch
Egon	$\{\gamma_3\}$	$\{\beta_1, \beta_2\}$	$\{\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5\}$
Elfriede	$\{\gamma_5\}$	$\{\beta_3\}$	$\{\alpha_6\}$
Hilde	$\{\gamma_4\}$	$\{\beta_2, \beta_3\}$	$\{\alpha_3, \alpha_4, \alpha_5, \alpha_6\}$
Hugo	$\{\gamma_2\}$	$\{\beta_1, \beta_2\}$	$\{\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5\}$
Maria	$\{\gamma_1\}$	$\{\beta_1\}$	$\{\alpha_1, \alpha_2\}$

# Zugriffspfade für Methoden

---

- Ergebnisse der Methodenausführung im Index gespeichert
  - ◆ parameterlose Methode: ein Methodenergebnis pro Objekt im Index
  - ◆ parameterbehaftete Methode: Methodenergebnis pro Objekt und pro möglicher Parameterbelegung im Index speichern;  
nicht effizient, daher nur bestimmte Bereiche aus der möglichen Wertemenge in den Index  
oder nur Parameter in den Index, die schon einmal bei einer Anfrage benutzt wurden (*adaptive*, *lernender Index*)
- Materialisierung von Methodenergebnissen:  
*Function-Materialization-Technik*

# Objektpuffer

---

- bisher:
  - ◆ Anwendungsdaten vom Seitenpuffer in die Hauptspeicherbereiche laden, die dem Anwendungsprogramm zur Verfügung stehen
  - ◆ „kostet“ eine Transformation der internen Darstellung in die vom Anwendungsprogramm gewünschte
- in einigen Systemen Objekte von der Platte direkt in den Anwendungsspeicher: (O<sub>2</sub>, Objectivity, ONTOS), eventuell mit gewissen Adresstransformationen (ObjectStore, siehe „Pointer Swizzling“)
- andere OODBs wie GemStone, ORION/ITASCA, DASDBS und OSCAR haben zweiten Puffer:  
*Objektpuffer*



# Pointer Swizzling

---

- Aufgabe: ein im Hauptspeicher befindliches Objekt beim Zugriff aus dem Anwendungsprogramm heraus schnell finden
- mit Objektpuffer und logischen Objektidentitäten:
  1. Objekt  $\alpha$  im Hauptspeicher besitzt ein Komponentenobjekt  $\beta$ , das im Objektpuffer nicht gefunden wird
  2. Objekt  $\beta$  wird im Seitenpuffer gesucht
    - ◆ Durchsuchen einer *Seitenzuordnungstabelle* (Resident Object Table, ROT)
    - ◆ Annahme:  $\beta$  ist nicht im Seitenpuffer

# Pointer Swizzling (II)

---

- mit Objektpuffer und logischen Objektidentitäten (fortg.):
    3.  $\beta$  vom Sekundärspeicher nachladen
      - ◆ *Blockzuordnungstabelle* (Persistent Object Table, POT) durchsuchen, um zu der gegebenen Objektidentität die Sekundärspeicheradresse zu ermitteln
    4. nach Laden des Objektes in den Seiten- bzw. Objektpuffer: bei jedem Zugriff im Anwendungsprogramm auf das Objekt wiederum die zugehörige Hauptspeicheradresse über die logische Objektidentität finden werden.
- ↪ zu umständlich und indirekt

# Pointer Swizzling (III)

---

- falls direkte Referenzen zur Implementierung der Objektidentität benutzt werden
  - ◆ Wegfall der Indirektion
  - ◆ trotzdem: direkte Sekundärspeicheradresse nützt im Hauptspeicher nichts  $\rightsquigarrow$  muss bei jedem Zugriff umgewandelt werden.
- falls Objektpuffer wegfällt:
  - ◆ keine Transformation aus dem Seitenpuffer notwendig
  - ◆ im Seitenpuffer enthaltenen Objekte müssen aber ebenfalls über deren aktuelle Hauptspeicheradressen gefunden werden

# Pointer Swizzling (IV)

---

- daher: Transformation von indirekten oder direkten (Sekundärspeicher-)Referenzen in Hauptspeicheradressen  $\rightsquigarrow$  *Pointer Swizzling*
- Varianten:
  - ◆ Original oder Kopie:
    - Zeigertransformation auf der Originalseite (im Seitenpuffer) oder auf der Kopie (im Objektpuffer)
  - ◆ Sofort oder verzögert:
    - Zeiger aller Objekte bereits beim Laden transformieren oder verzögert beim ersten Zugriff auf das Objekt im Hauptspeicher

# Pointer Swizzling (V)

---

- Varianten (fortg.):
  - ◆ Direkt oder indirekt:
    - Transformation in die direkte Hauptspeicheradresse oder „nur“ in einen Deskriptor (indirekter Zeiger), der die Hauptspeicheradresse enthält
- Systeme
  - ◆ ORION: sofortiges, indirektes Swizzling
  - ◆ O<sub>2</sub>: kein Pointer Swizzling
  - ◆ Exodus: verzögerte Strategie
  - ◆ ObjectStore: sofortiges, direktes Swizzling (VMMA)

# Geometrische Zugriffsstrukturen

---

- große Mengen geometrischer Objekte ( $> 10^6$ )
- Eigenschaften geometrischer Objekte
  - ◆ Geometrie (etwa Polygonzug)
  - ◆ zur Unterstützung bei Anfragen: zusätzlich  $d$ -dimensionales umschreibendes Rechteck (*bounding box*)
  - ◆ nichtgeometrische Attribute
- Anwendungsszenarien: Geoinformationssysteme (Katasterdaten, Karten), CAx-Anwendungen (etwa VLSI Entwurf), ...
- Zugriff primär über Geometriedaten: Suchfenster (Bildschirmausschnitt), Zugriff auf benachbarte Objekte

# Typische Operationen

---

- exakte Suche
  - ◆ Vorgabe: exakte geometrische Suchdaten
  - ◆ Ergebnis: maximal ein Objekt
- Bereichsanfrage für vorgegebenes  $n$ -dimensionales Fenster
  - ◆ Suchfenster:  $d$ -dimensionales Rechteck (entspricht mehrdimensionalem Intervall)
  - ◆ Ergebnis sind alle geometrischen Objekte, die das Suchfenster schneiden
  - ◆ Ergebnisgröße parameterabhängig

# Typische Operationen (II)

---

- Einfügen von geometrischen Objekten
  - ◆ *wünschenswert ohne globale Reorganisation!*
- Löschen von geometrischen Objekten
  - ◆ *wünschenswert ohne globale Reorganisation!*



# Nachbarschaftserhaltende Suchbäume

---

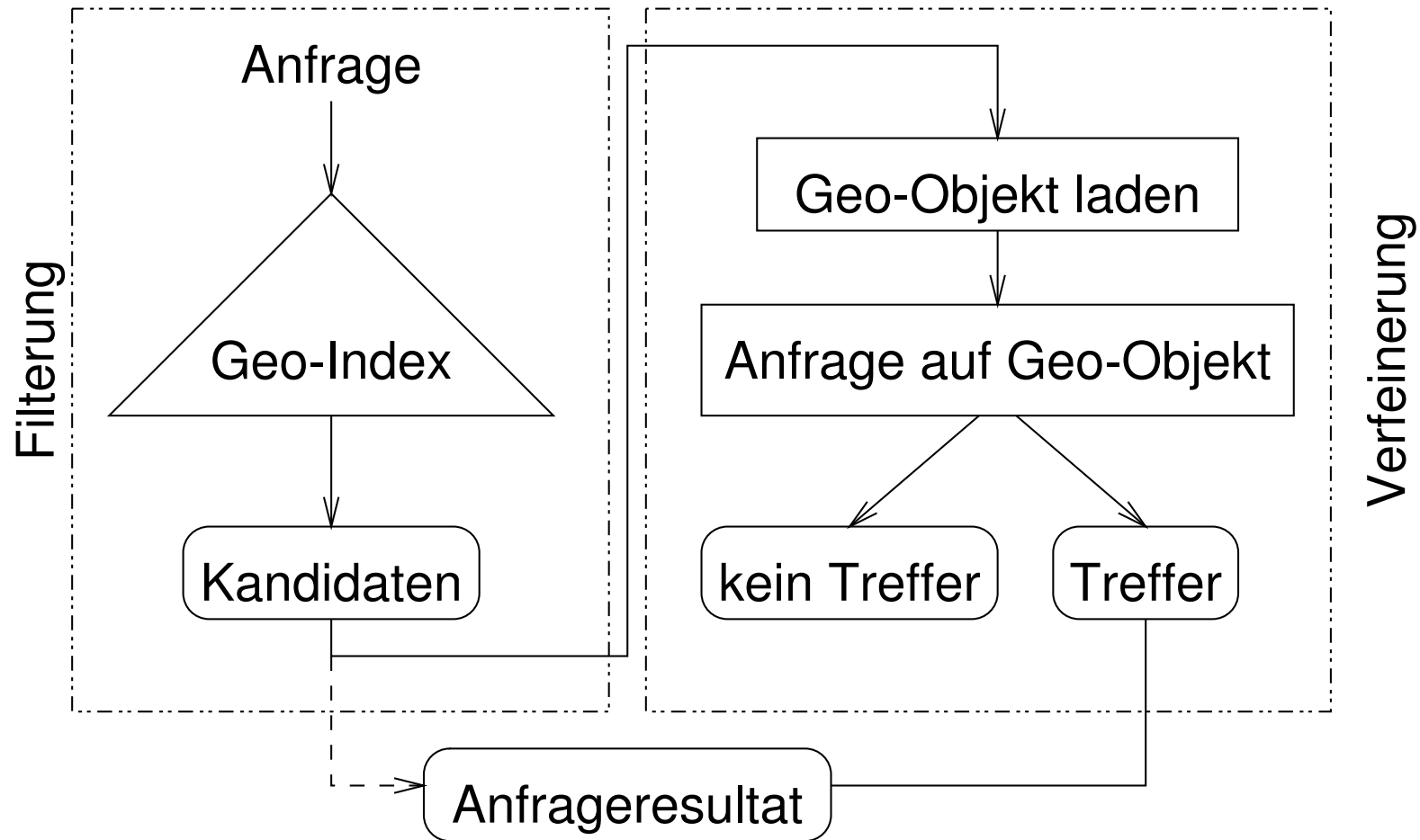
- Aufteilung des geometrischen Bereichs in *Regionen*
- benachbarte Objekte wenn möglich der *selben* Region zuordnen
- falls dieses nicht geht, diese auf *benachbarte* Regionen aufteilen
- Baumstruktur entsteht durch Verfeinerung von Regionen in benachbarte Teilregionen
- Speicherung von Objekten erfolgt in den Blattregionen

# Nachbarschaftserh. Suchbäume (II)

---

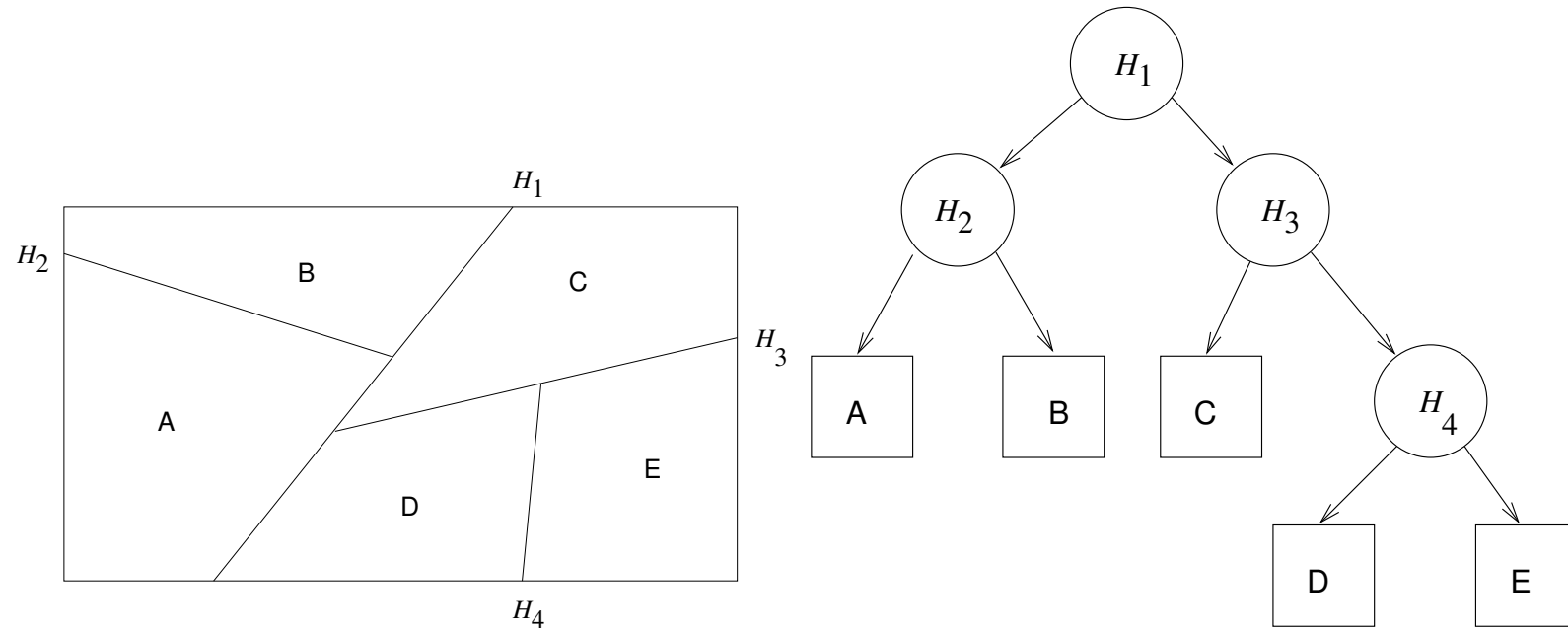
- Freiheitsgrade
  - ◆ Form der Regionen
  - ◆ vollständige Partitionierung oder Überlappung durch die Regionen
  - ◆ eindeutige Zuordnung von Objekten zu Regionen oder Mehrfachzuordnung
  - ◆ Speicherung und Zugriff über Originalgeometrie oder über abgeleitete Geometrie für Objekte
  - ◆ Grad des Baumes & Organisationsform

# Mehrstufige Bearbeitung geom. Anfragen



# Geom. Baumstruktur: BSP-Baum

---



# Realisierungsvarianten

---

Alternative	Baumstrukturen			
	BSP-Baum	R-Baum	R <sup>+</sup> -Baum	Zell-Baum
Regionenform	konvexe Polygone	Rechtecke	Rechtecke	konvexe Polygone
Teilregionen	vollständig	unvollständig	unvollständig	vollständig
Überlappung	nein	ja	nein	nein
ausgeglichen	nein	ja	ja	ja

# R-Bäume (I)

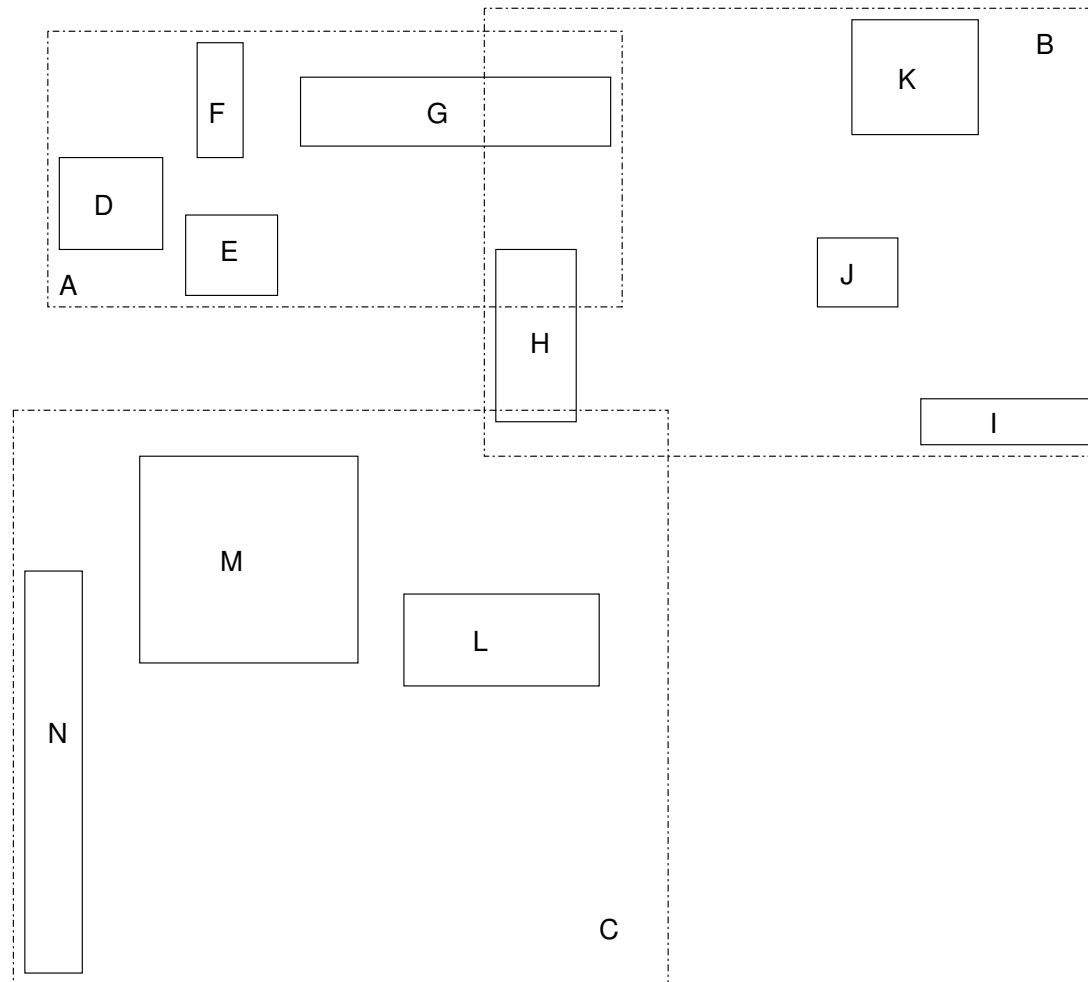
---

- R-Baum: Verallgemeinerung des B-Baum-Prinzips auf mehrere Dimensionen
  - ◆ Baumwurzel entspricht einem Rechteck, das alle geometrischen Objekte umfasst
  - ◆ Geo-Objekte werden durch ihre umschließenden Rechtecke repräsentiert
  - ◆ Aufteilung in Regionen erfolgt in nichtdisjunkte Rechtecke
  - ◆ Jedes Geo-Objekt ist eindeutig einem Blatt zugeordnet

# R-Bäume (II)

---

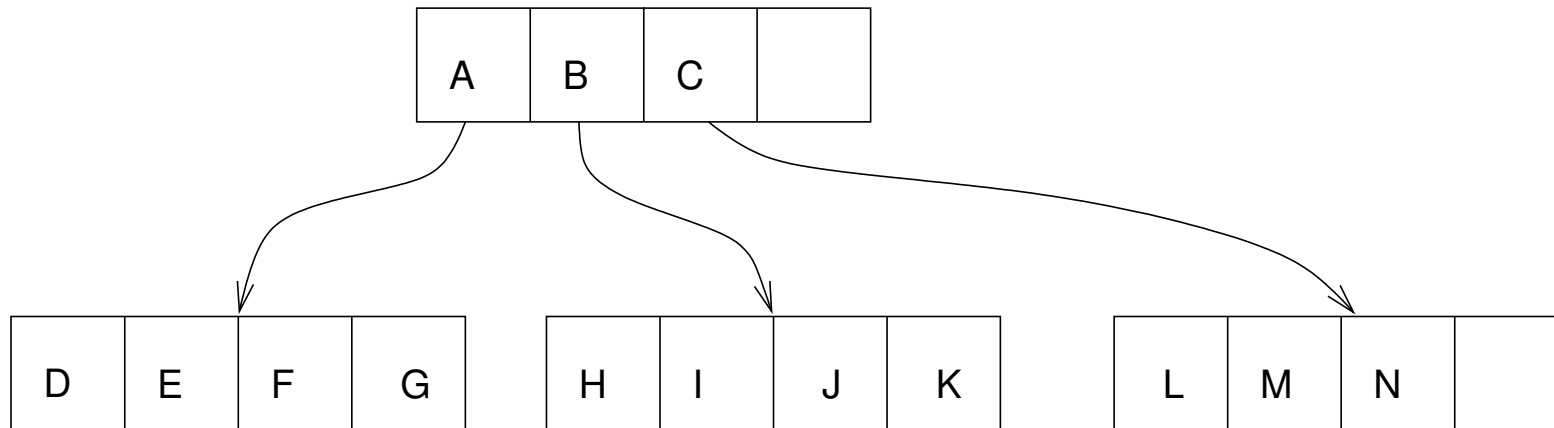
## ■ Regionenaufteilung durch Rechtecke im R-Baum



# R-Bäume (III)

---

## ■ Baumstruktur für R-Baum



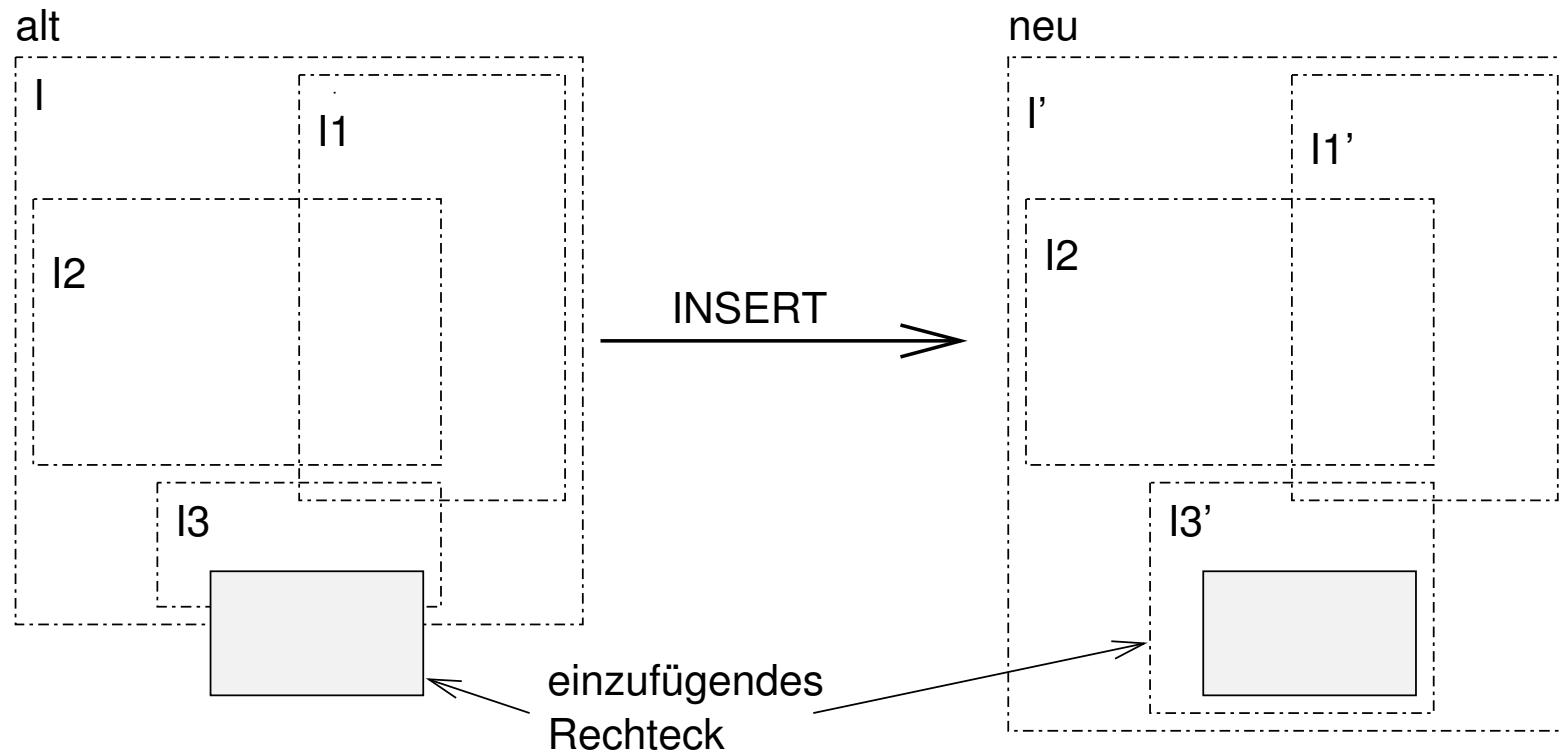


# Probleme mit R-Bäumen

---

- gegebenes Rechteck kann von vielen Regionen überlappt werden, es ist aber genau in einer Region gespeichert
- auch Punktanfragen können eine Suche in sehr vielen Rechteckregionen bedeuten
- Ineffizient bei exakter Suche (exakte Suche auch bei Einfügen und Löschen notwendig!)
- Probleme beim Einfügen
  - ◆ Einfügen erfordert oft Vergrößern von Regionen (aufwärts propagiert)

# Vergrößern beim Einfügen



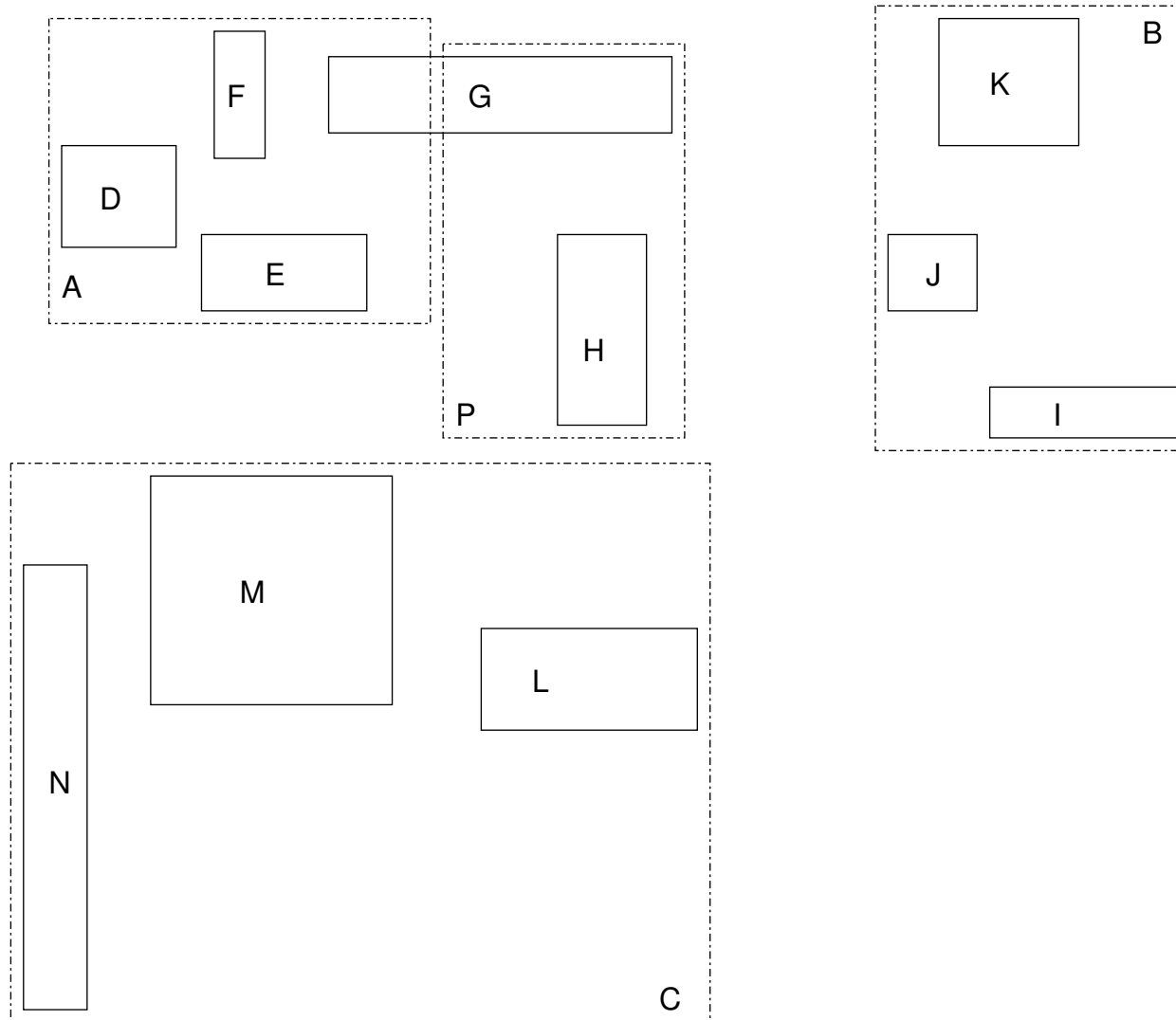
# $R^+$ -Bäume (I)

---

- $R^+$ -Bäume: Aufteilung in Teilregionen *disjunkt*
- Jedem gespeicherten Punkt des geometrischen Bereichs ist eindeutig ein Blatt zugeordnet
- In jeder Baumebene ist einem Punkt ebenfalls maximal ein Rechteck zugeordnet → eindeutiger Pfad von der Wurzel zum speichernden Blatt
- 'Clipping' von Geo-Objekten notwendig!

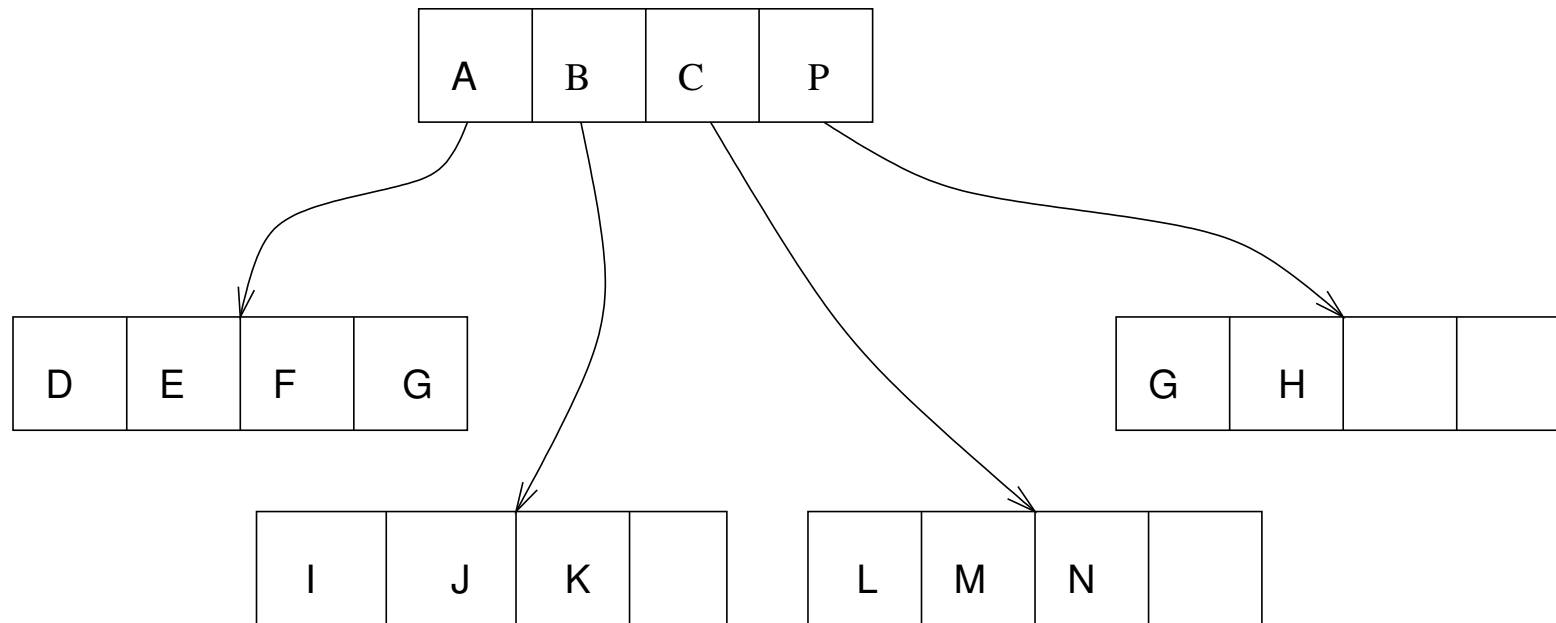
# R<sup>+</sup>-Bäume (II)

---



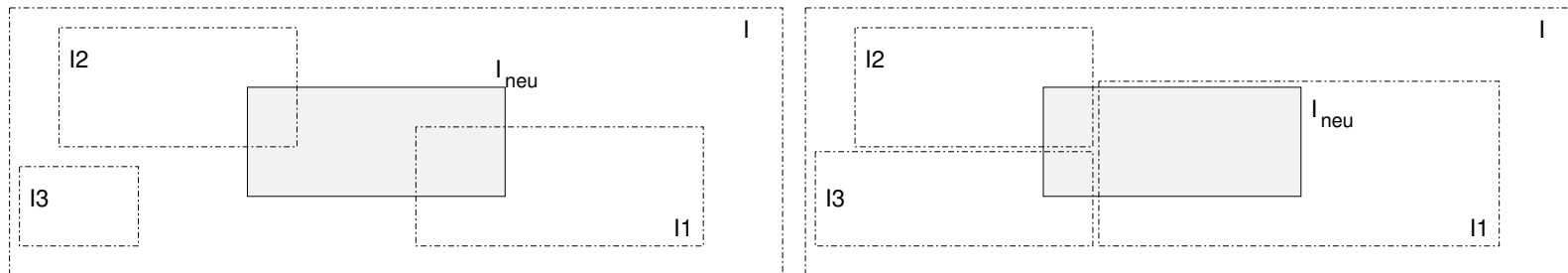
# R<sup>+</sup>-Bäume (III)

---



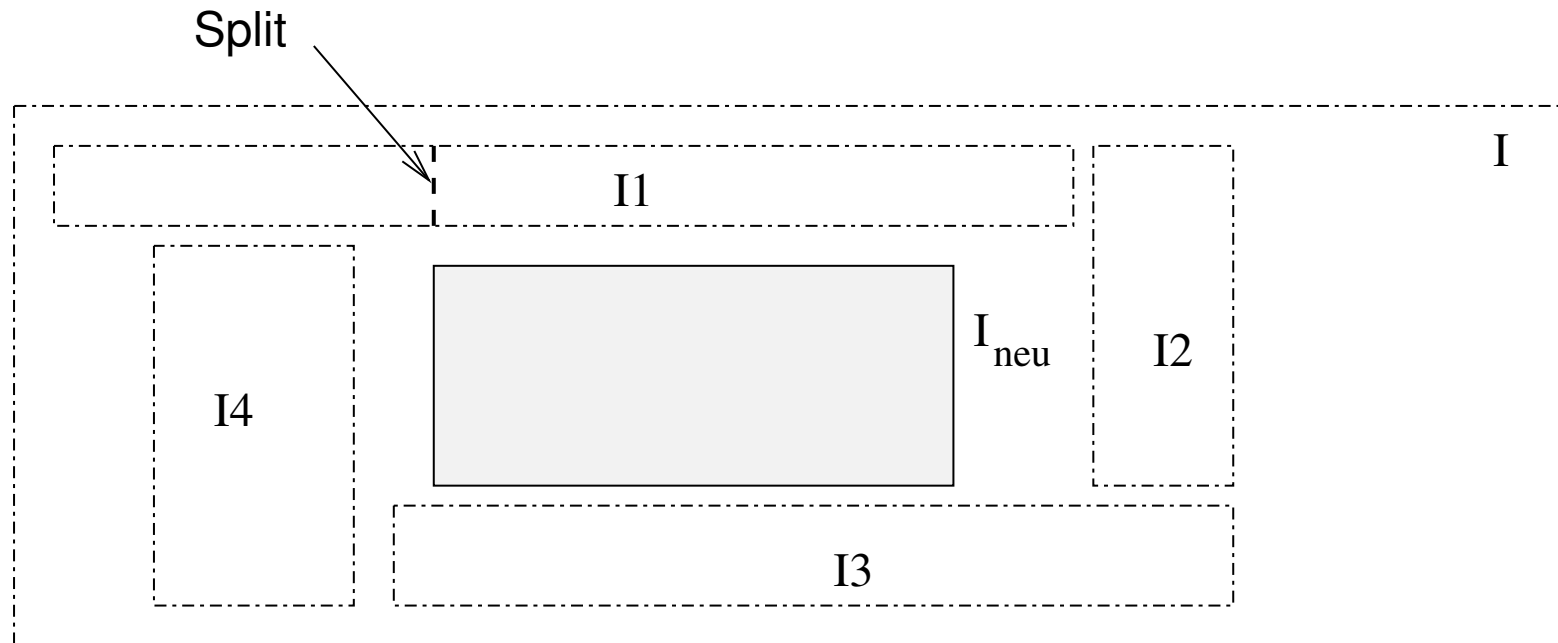
# Probleme mit $R^+$ -Bäumen

- Objekte müssen in mehreren Rechteckregionen gespeichert werden (*clipping*) — erhöhter Speicher- & Modifikationsaufwand
- Einfügen von Objekten erfordert möglicherweise Modifikation mehrerer Rechteckregionen



# Probleme mit $R^+$ -Bäumen (II)

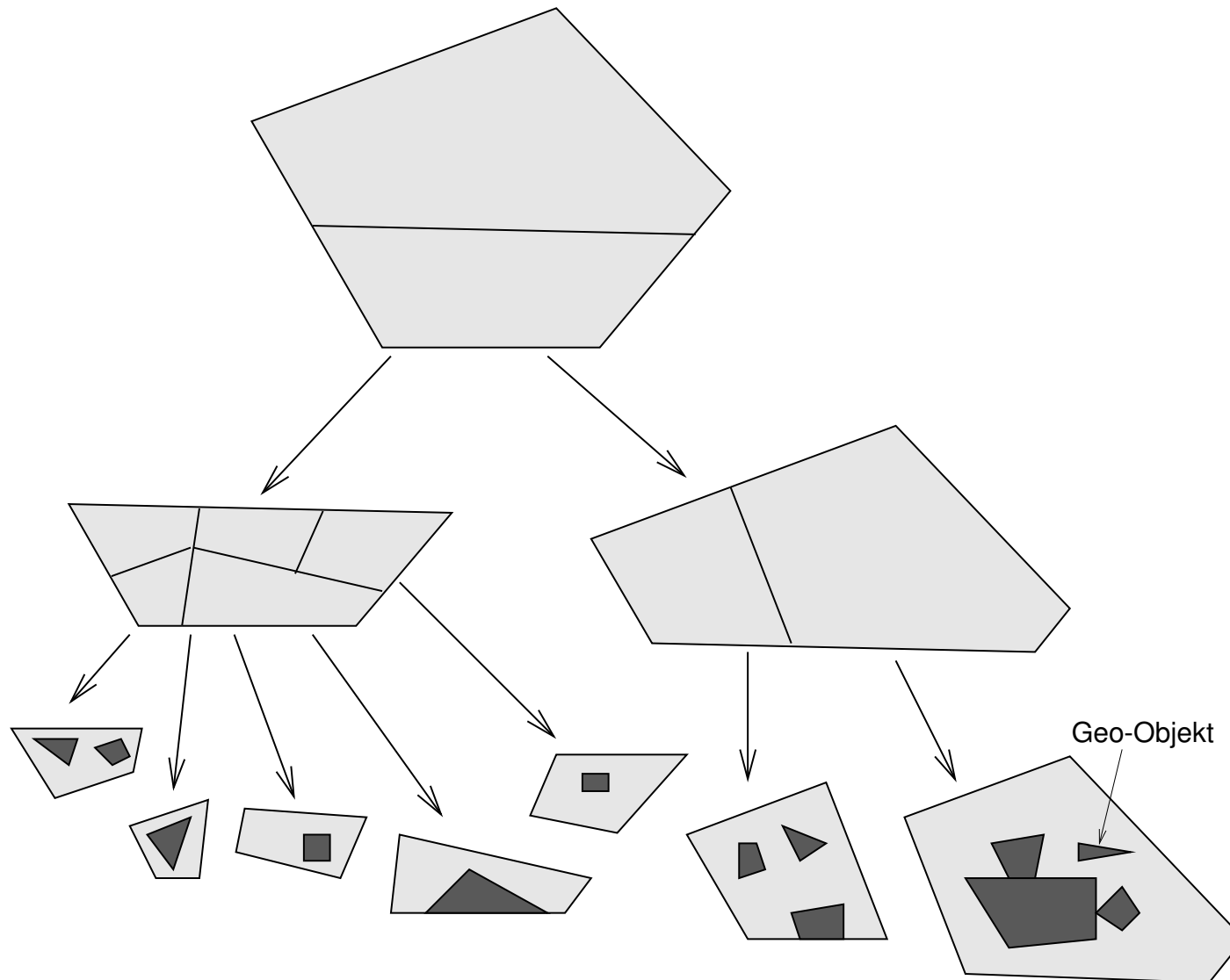
- Einfügen kann in bestimmten Situationen *unvermeidbar* zu Regionenaufteilungen führen



- Regionenmodifikationen haben Konsequenzen sowohl in Richtung Blätter als auch in Richtung Wurzel
- obere Grenze für Einträge in Blattknoten kann nicht mehr garantiert werden

# Zell-Bäume

---





# Punktdatenstrukturen

---

- Rechteckspeicherung durch Punktdatenstrukturen
  - ◆ Transformation von ausgedehnten Objekten (mehrdimensionale Rechtecke) in Punktdaten
  - ◆ Transformation bildet  $d$ -dimensionale Rechtecke auf Punkte im  $2d$ -dimensionalen Raum  $\mathcal{R}^{2d}$  ab
  - ◆  $d$ -dimensionale Rechteck :

$$r = [l_1, r_1] \times \cdots \times [l_d, r_d]$$

# Punktdatenstrukturen (II)

---

## ■ Eckentransformation

$$p_r = (l_1, r_1, \dots, l_d, r_d) \in \mathcal{R}^{2d}$$

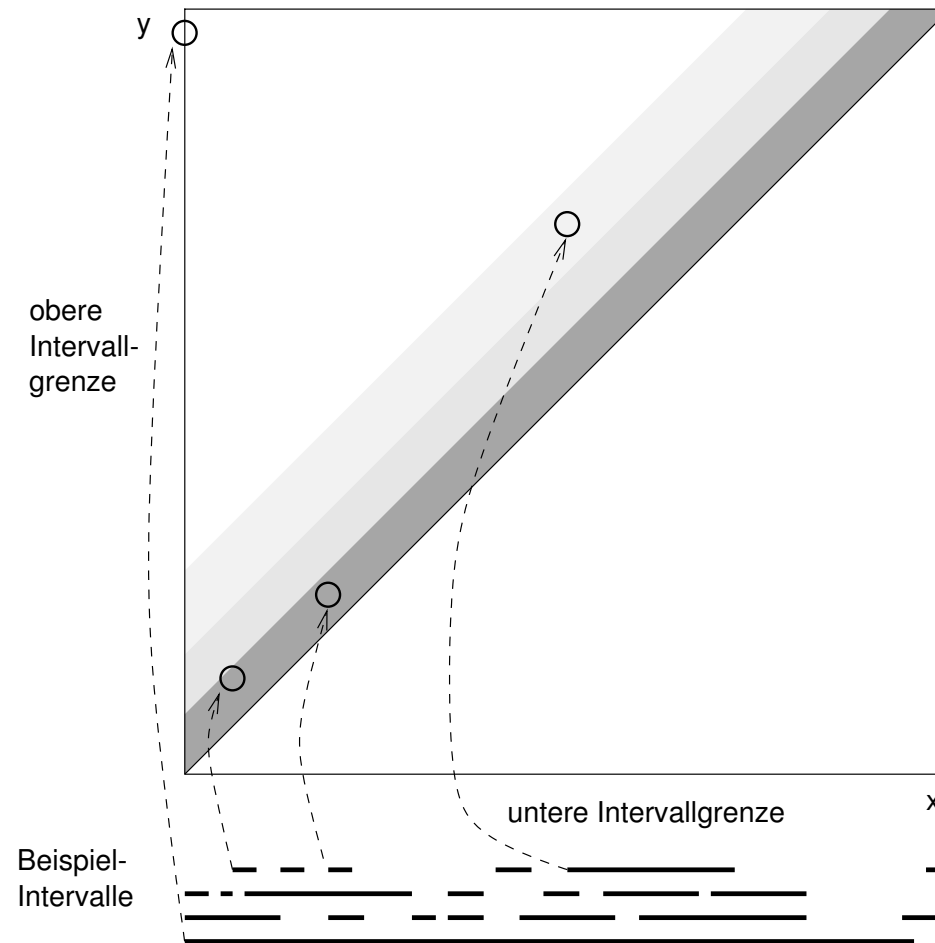
pro Intervall als Koordinaten: obere Schranke, untere Schranke

## ■ Mittentransformation

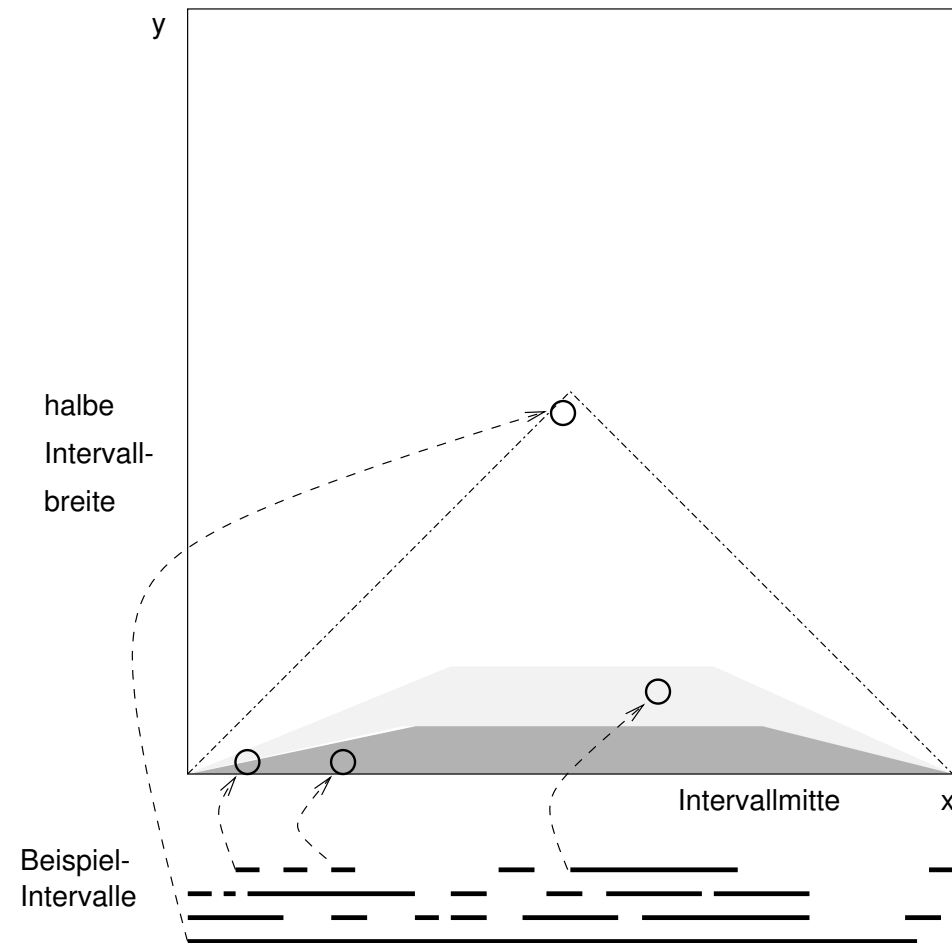
$$p_r = \left( \frac{l_1 + r_1}{2}, \frac{r_1 - l_1}{2}, \dots, \frac{l_d + r_d}{2}, \frac{r_d - l_d}{2} \right) \in \mathcal{R}^{2d}$$

pro Intervall als Koordinaten: Mittelpunkt, halbe Breite

# Eckentransformation

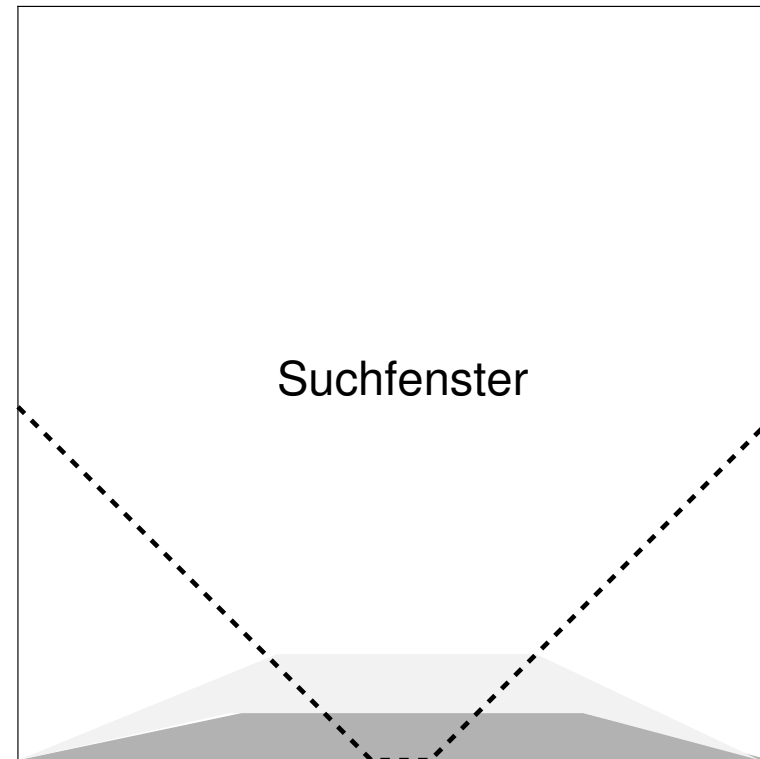
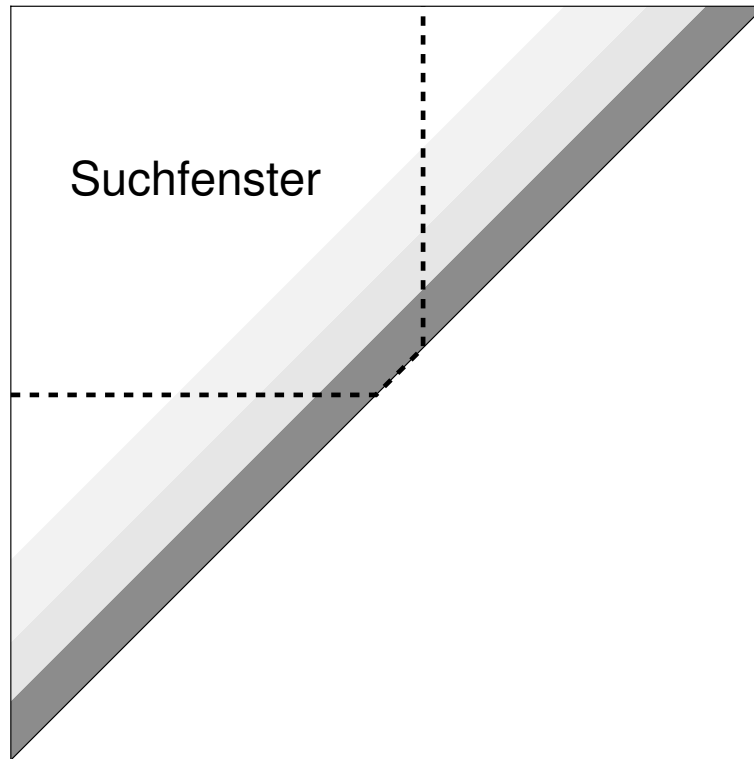


# Mittentransformation



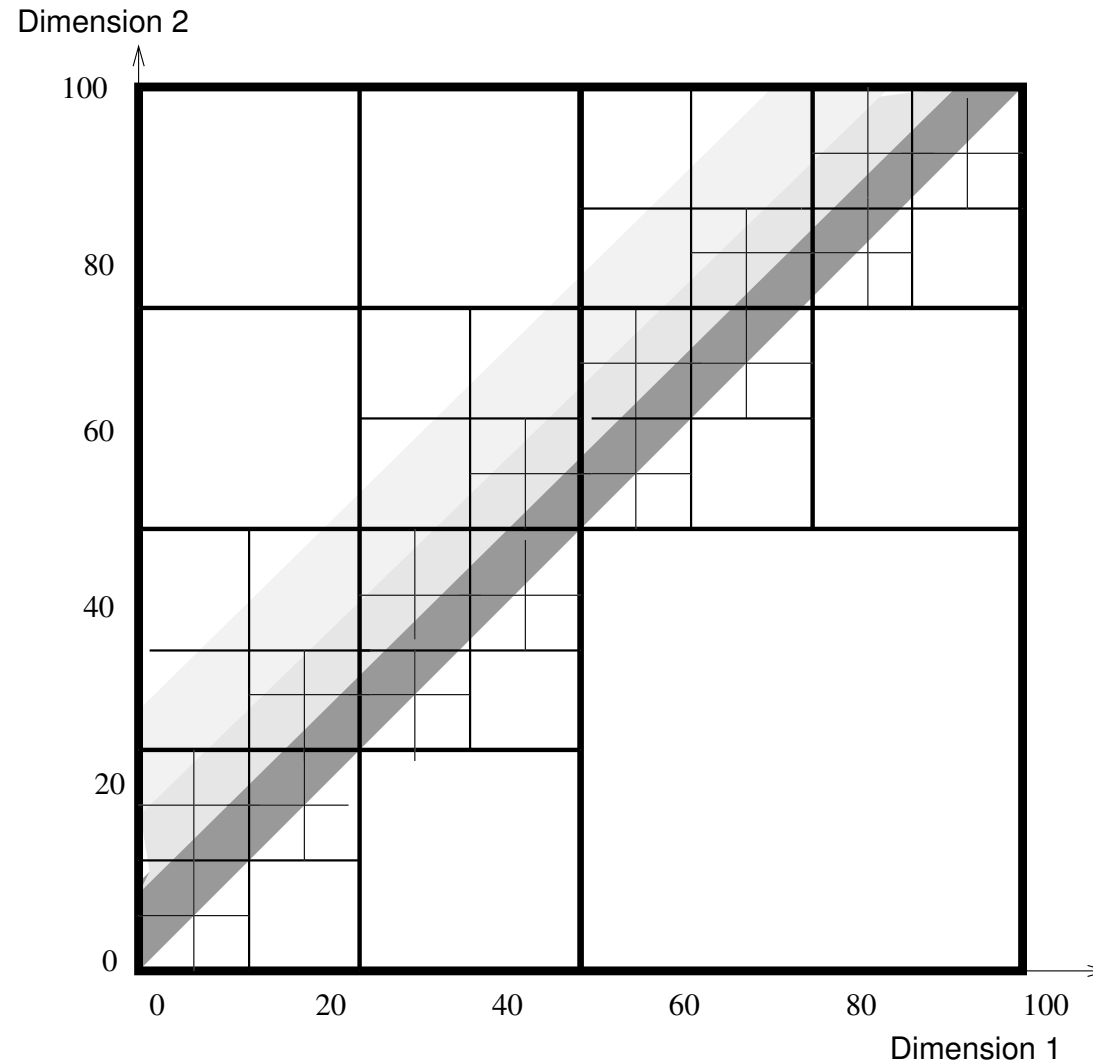
# Suchfenster

---



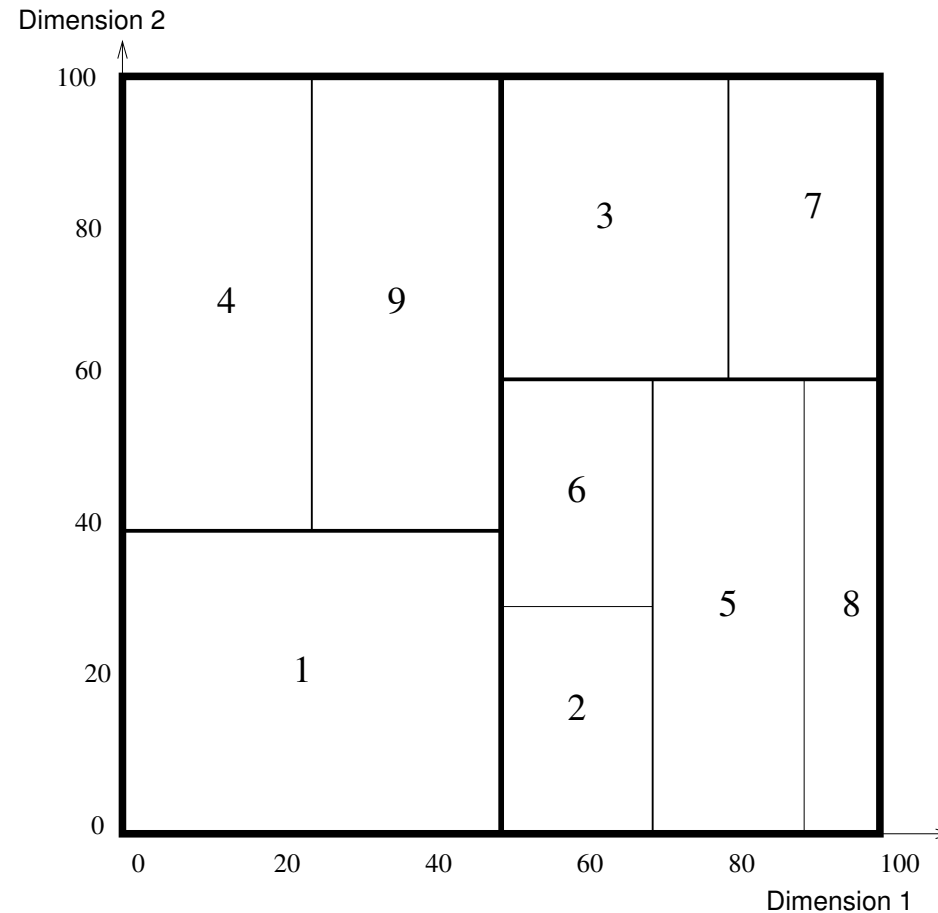
# Grid-File-Degeneration

---

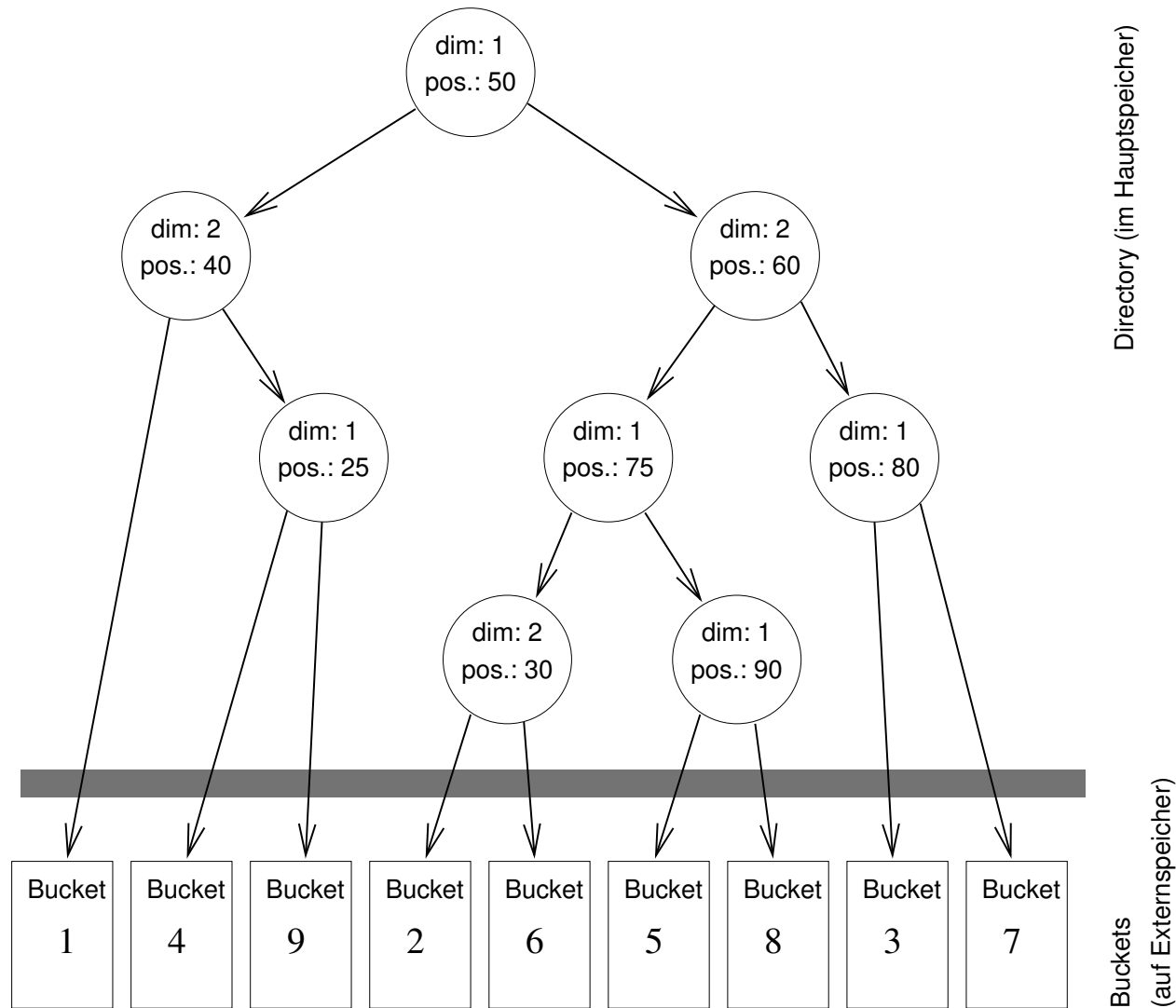


# Grid-File-Erweiterungen: LSD-Bäume

- Mögliche Aufteilung einer Ebene für einen LSD-Baum



# Beispiel für LSD-Bäume





# Zugriffsstrukturen für Multimedia-DB

---

- Medien-Objekte → große *Binär-Objekte*

Format	Megabyte	Komprimierung
JPEG	75	unkomprimiert
MPEG-1	12.5	komprimiert, Qualitätsverlust
MPEG-2	17	komprimiert, hohe Qualität

1 Minute kombinierte Audio/Video-Aufzeichnung

- Indexierung über abgeleitete Eigenschaften (sogenannte *Features*)  
→ Indexierung über hoch-dimensionale *Feature-Vektoren*
- *kontinuierliche Datentypen*

# Kontinuierliche Datentypen

---

- Datentypen zur Speicherung *kontinuierlicher Medien*
  - ◆ Daten müssen *schnell genug* zum Abspielen in Echtzeit in den Hauptspeicher gebracht werden.
  - ◆ da Bandbreite für den Transfer vom Hintergrundspeicher in den Hauptspeicher nicht konstant ist, wird in der Regel *auf Vorrat in einen Hauptspeicher-Cache (Prefetching-Strategie)* geladen
  - ◆ aber: *Cache-Überlauf* vermeiden
  - ◆ gerade bei kontinuierlichen Medien: *Speicherung auf Tertiär-Speichern* weit verbreitet  $\leadsto$  zweistufige Caching-Strategie notwendig

# Hochdimensionale Indexe

---

- Feature-Vektoren zur Charakterisierung von Medien-Objekten
- Feature-Vektor: Punkt im hochdimensionalen Raum
- Datenbank als Menge von Punkten des  $d$ -dimensionalen Raumes:

$$DB = \{P_0, \dots, P_{n-1}\}$$

- jeder Punkt dieser Datenbank ist ein  $d$ -dimensionaler Vektor:

$$P_i \in DB \subseteq \mathbb{R}^d$$

# Typische Operation auf Feature-Vektoren

---

- *Bereichsanfragen* (engl. *range queries*) berechnen für gegebenen Vektor alle benachbarten Punkte der Datenbank:

$$\mathbf{range}(DB, Q, r, M) = \{P \in DB \mid \delta_M(P, Q) \leq r\}$$

mit:

- ◆  $DB$ : die Datenbank als Menge von Feature-Vektoren, in der gesucht wird
- ◆  $r$ : Abstand, der den Bereich („range“) festlegt
- ◆  $Q$ : Suchvektor
- ◆  $M$ : Metrik

# Typische Operation auf Feature-Vektoren

---

- *Punktanfrage* (engl. *point query*) definiert die exakte Suche und entspricht somit einer Bereichsanfrage mit Abstand 0:

$$\mathbf{point}(DB, Q, M) = \{P \in DB \mid \delta_M(P, Q) = 0\}$$

- *Nächster-Nachbar-Anfragen* (engl. *nearest neighbor query*) bestimmen für einen Suchvektor den diesem am nächsten gelegenen Vektor der Datenbank:

$$\mathbf{nearest-neighbor}(DB, Q, M) =$$

$$\{P \in DB \mid \forall (P' \in DB) : \delta_M(P, Q) \leq \delta_M(P', Q)\}$$

# Distanzfunktion $\delta$

---

1.  $\delta(p, p) = 0$
2.  $\delta(p_1, p_2) = \delta(p_2, p_1)$
3.  $\delta(p_1, p_2) + \delta(p_2, p_3) \geq \delta(p_1, p_3)$  (Dreiecksungleichung)

# Metriken

---

## ■ *euklidische Distanz*

$$\delta_{Euclid}(P, Q) = \sqrt{\sum_{i=1}^d (Q_i - P_i)^2}$$

$Q_i$  ist Wert von  $Q$  für die  $i$ -te Dimension

## ■ *Manhattan-Distanz*

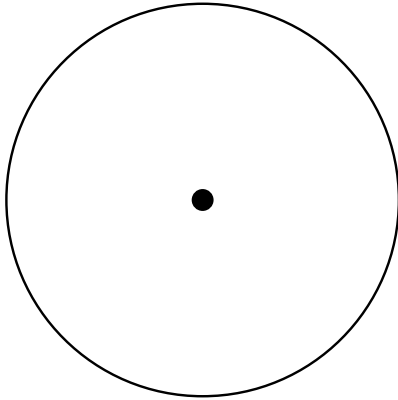
$$\delta_{Manhattan}(P, Q) = \sum_{i=1}^d |Q_i - P_i|$$

## ■ *Maximum-Distanz*

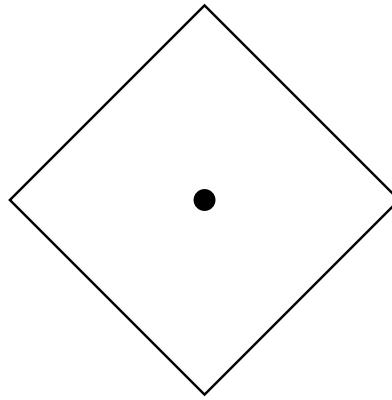
$$\delta_{Max}(P, Q) = \max\{|Q_i - P_i|\}$$

# Metriken im Vergleich

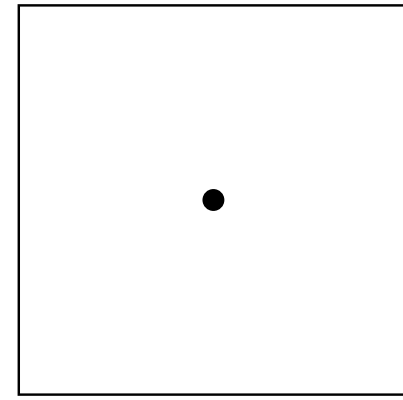
---



Euclid



Manhattan

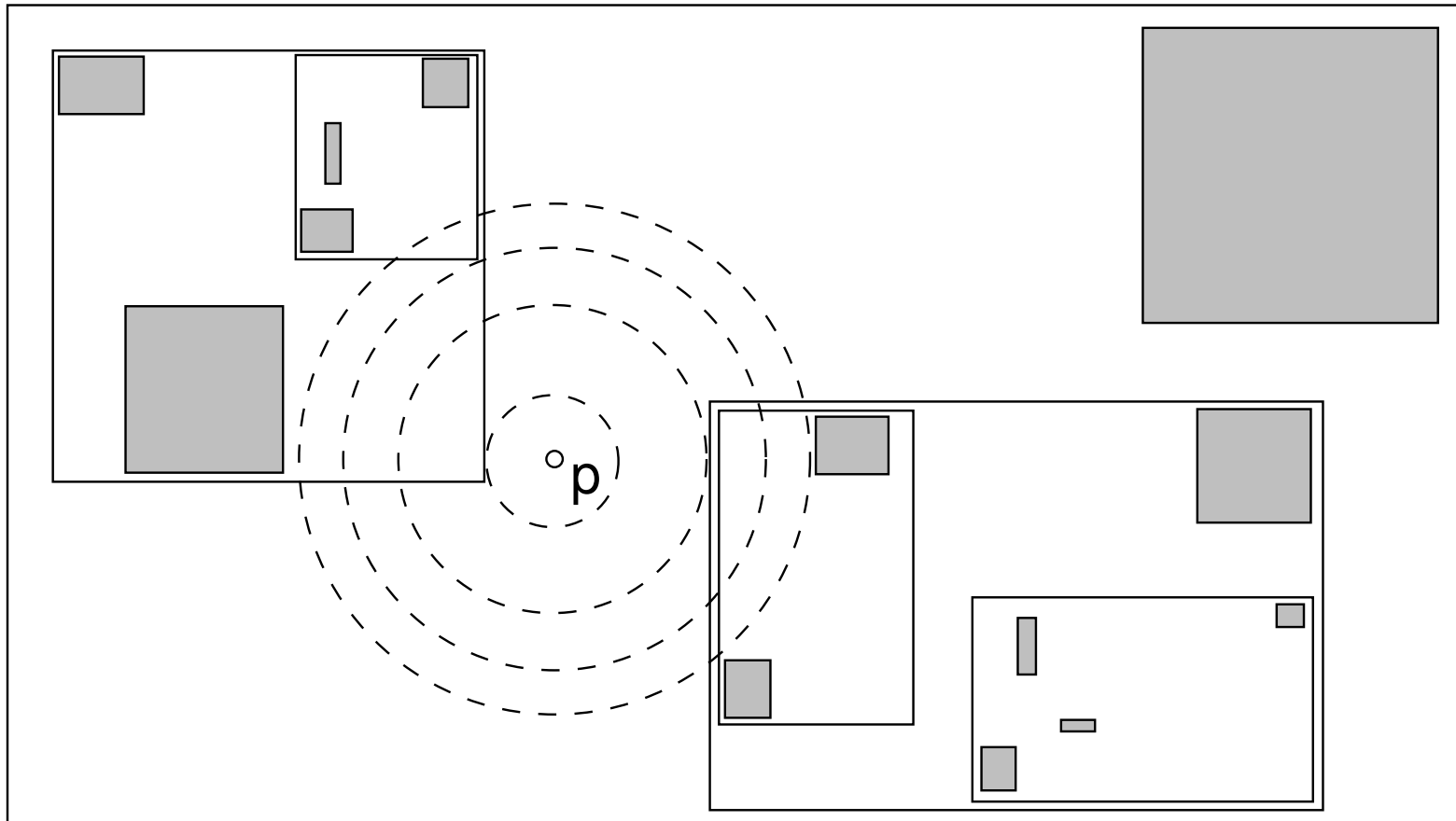


Max



# Nächster-Nachbar-Suche in R-Bäumen

---



# MinDist und MinMaxDist

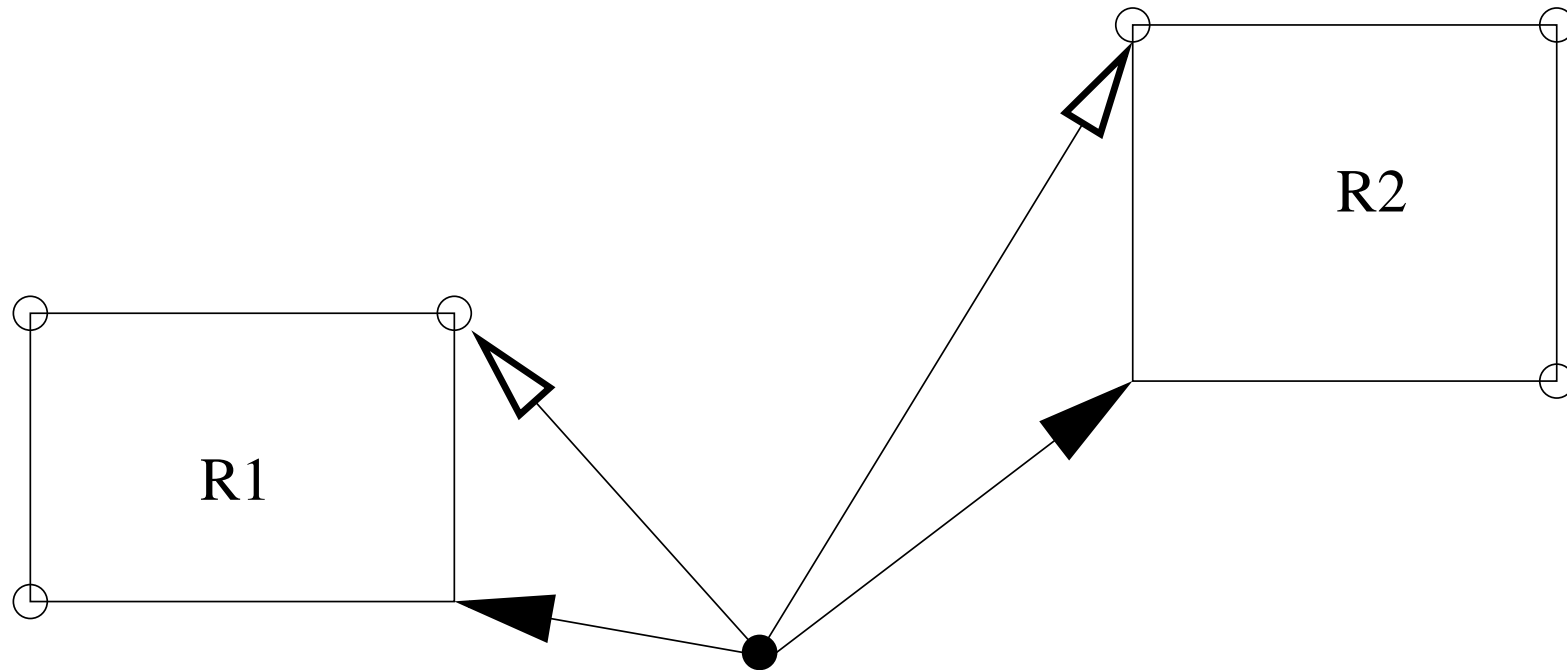
---

- **MinDist:** *Min-Distanz* ist minimale Distanz zwischen Anfragepunkt und dem MBR
- **MinMaxDist:** für jede Hyperfläche ( $n - 1$  Dimensionen) wird der vom Anfragepunkt entfernteste (ungünstigster Fall) Punkt ermittelt
  - ◆ Min-Max-Punkt ist dann der minimal entfernte Punkt dieser Max-Punkte und dessen Distanz zum Anfragepunkt die Min-Max-Distanz

$$\text{MinDist} \leq \text{MinMaxDist}$$

# MinDist und MinMaxDist (II)

---



—▶ MinDist  
—▷ MinMaxDist

# Nächster Nachbar

---

- aktuell angenommene minimale Distanz **Dist**
  - ◆ **Dist** < **MinDist** → Sohnknoten müssen nicht aufgesucht werden, da in ihnen kein Punkt enthalten sein kann, der als Treffer in Frage kommt
  - ◆ **MinDist** ≤ **Dist** ≤ **MinMaxDist** → Sohnknoten müssen aufgesucht werden, da in ihnen ein Kandidat enthalten sein kann
  - ◆ **Dist** > **MinMaxDist** → das betrachtete MBR oder ein Sohnknoten enthalten garantiert einen Kandidaten, der näher als der aktuell als nächster Punkt angenommene Punkt ist
  - ◆ Finden eines näheren Kandidaten → Aktualisierung von **Dist**

# X-Baum als Beispiel

---

- Probleme bei Leistung von R-Baum-Strukturen bei hohen Dimensionszahlen:
  - ◆ große Überlappung entsteht oft durch Splitten innerer Knoten bei Auswahl falscher Dimensionen
  - ◆ bei großer Überlappung ist lineare Suche schneller als der Block-orientierte Zugriff auf Baum-Knoten

# X-Baum als Beispiel (II)

---

- X-Baum: Modifikationen von R-Bäumen
  - ◆ konstanter Faktor **max\_overlap** legt einen maximalen Überlappungsgrad fest
  - ◆ für MBR wird eine *Split-Historie* über die Dimensionen der bisherigen Aufteilungen geführt
  - ◆ *Super-Knoten* haben die Größe mehrerer normaler innerer Baumknoten und können viele Einträge aufnehmen

# X-Baum als Beispiel (III)

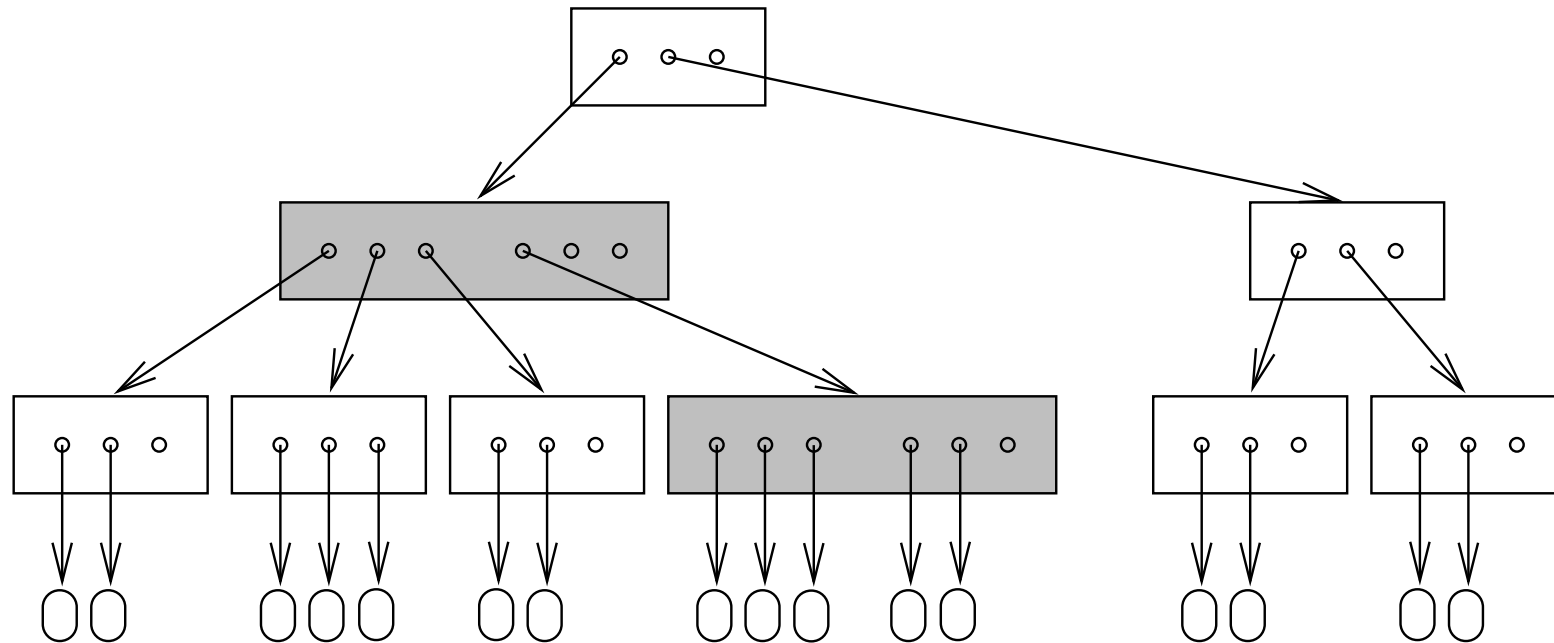
---

## ■ Knotenarten:

- ◆ einfache innere Knoten: gleicher Aufbau wie beim R-Baum, aber zusätzlich Abspeichern der Split-Historien
- ◆ innere *Super-Knoten* überspannen mehrere Datenblöcke und verweisen damit auf eine größere Zahl von Einträgen
- ◆ Blattknoten entsprechen den Blättern des R-Baums

# X-Baum graphisch

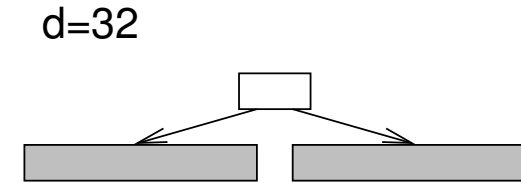
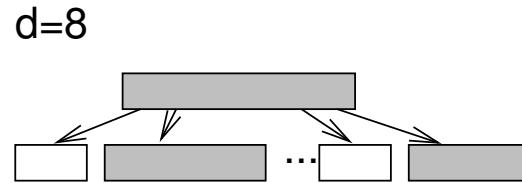
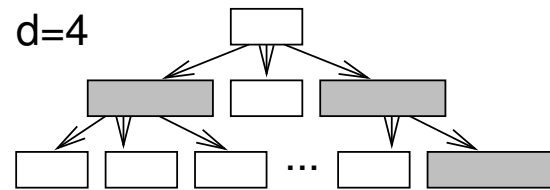
---





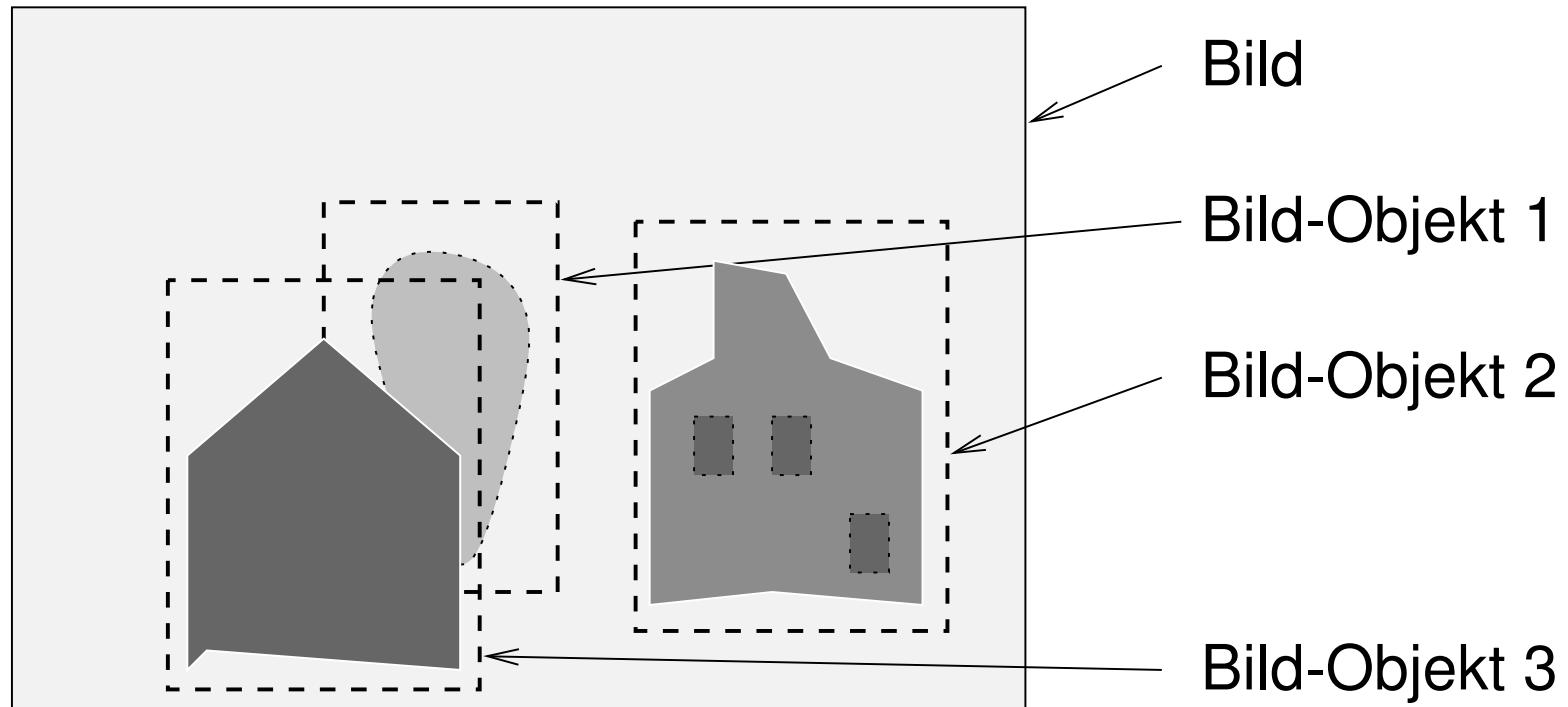
# X-Baum abhängig von Dimensionszahl

---



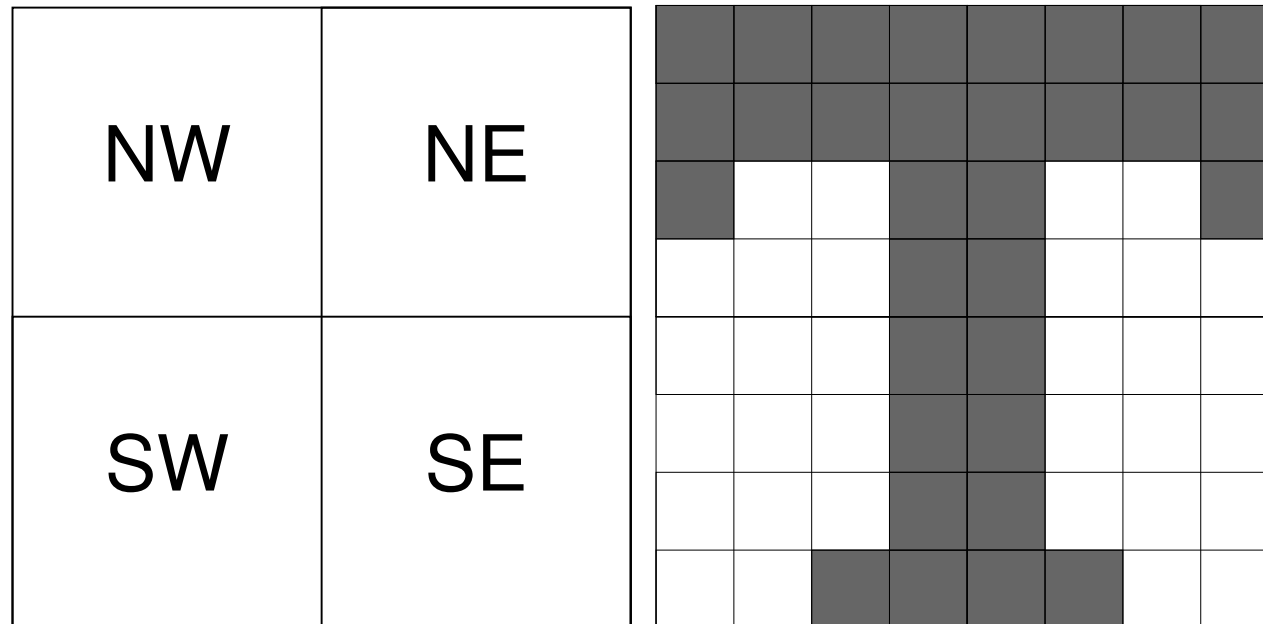
# Speicherung von Bildern

---



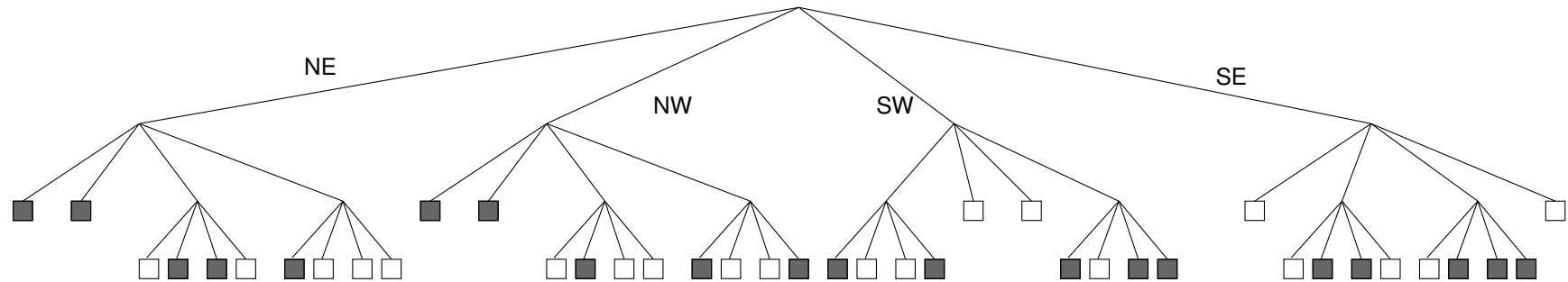
# Quadtrees

---



# Quadrees (II)

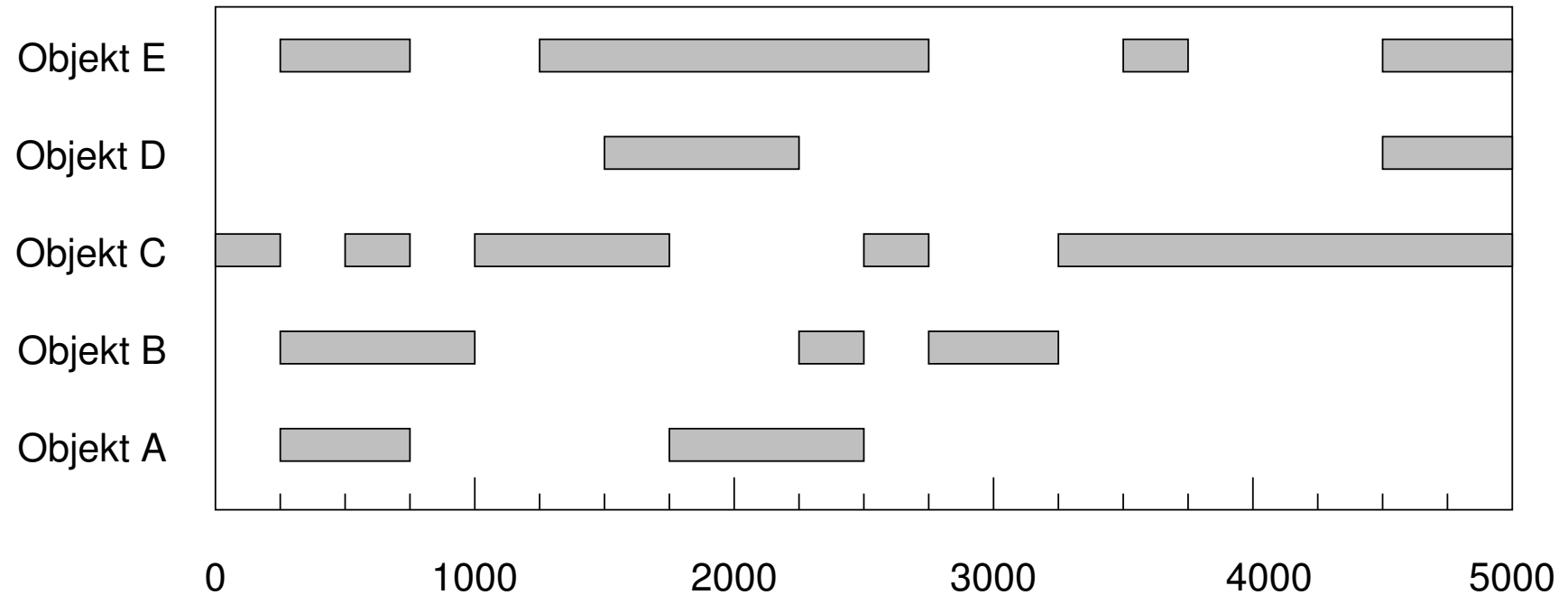
---



# Video-Daten

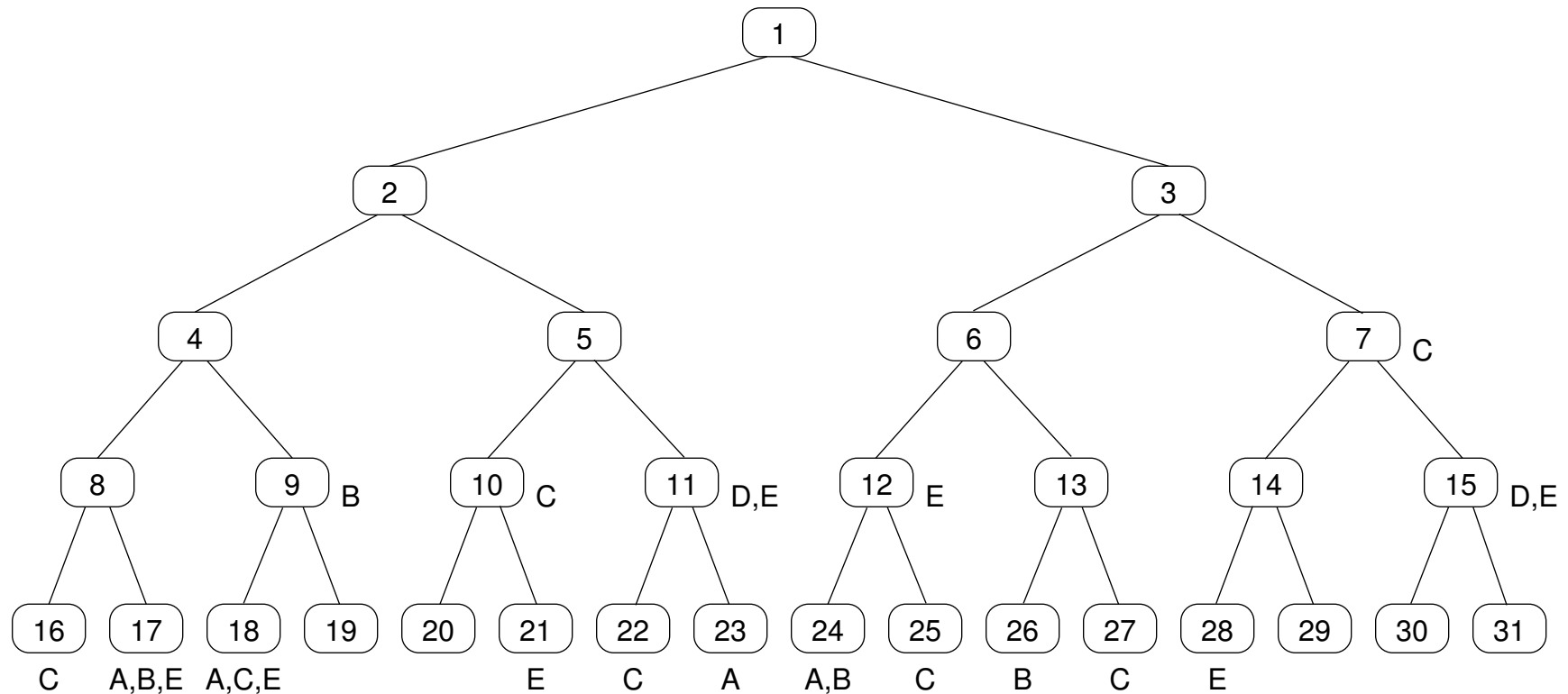
---

## Video-Segmente:



# Video-Daten: Segment-Baum

---



# Information Retrieval

---

- *Information Retrieval:*

Verfahren zur Speicherung und Wiedergewinnung  
(engl. *retrieval*) von Volltextdokumenten

- Beispiele:

```
[ 'Datenbanken' ] in Text
```

```
[ 'Datenbanken' and 'Multimedia' ] in Text
```

```
[ 'Objekt' 1 Wort vor 'Orientierung' ] in Text
```

```
[ 'Objekt' im gleichen Satz mit  
  'Orientierung' ] in Text
```

```
[ 'Objekt' innerhalb  
  2 Abschnitte mit 'Orientierung' ] in Text
```

# Qualitätsmaße im Information Retrieval

---

$$\text{Recall} = \frac{\text{Anzahl gefundener relevanter Dokumente}}{\text{Gesamtanzahl relevanter Dokumente}}$$

$$\text{Precision} = \frac{\text{Anzahl gefundener relevanter Dokumente}}{\text{Gesamtanzahl gefundener Dokumente}}$$

$$\text{Fallout} = \frac{\text{Anzahl gefundener irrelevanter Dokumente}}{\text{Gesamtanzahl irrelevanter Dokumente}}$$



# Konzept-Indizierung

---

- *Linguistische Analysen:*
  - ◆ *morphologische Analyse* erzeugt einen Wortstamm durch Elimination von Prä- und Suffixen und Pluralendungen → *Stemming*
  - ◆ Ableitung neuer Wortformen aus bereits bekannten → insbesondere für Deutsch!
- *Synonyme*: binäre Beziehung zwischen äquivalenten Worten
- *Begriffshierarchie* in (fachspezifischen) *Thesaurus*: Vererbung von Eigenschaften → Anfragen nach allgemeineren oder spezielleren Begriffen

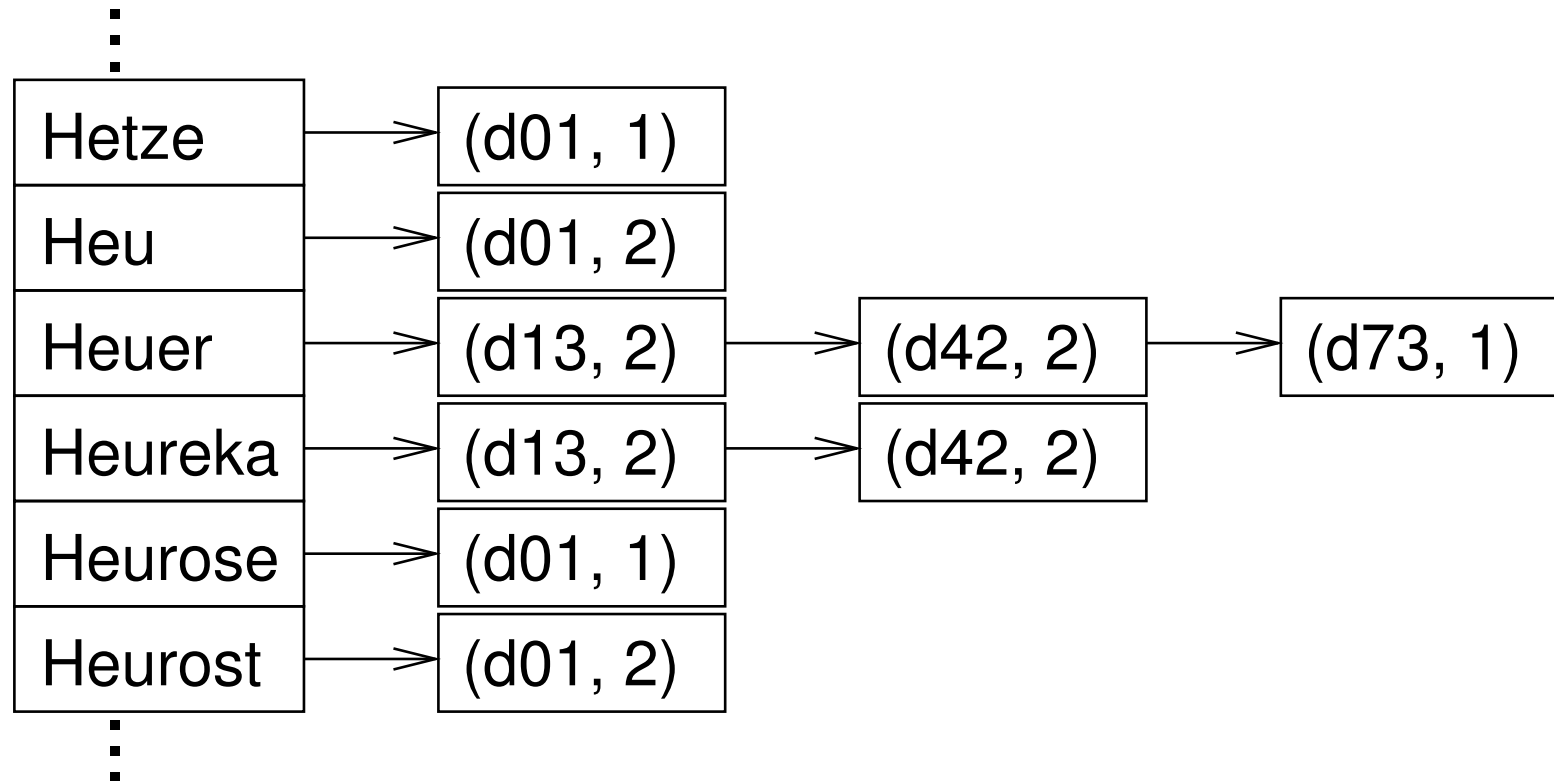
# Invertierte Listen

---

- indizierten Worte (Zeichenketten) bilden eine lexikographisch sortierte Liste
- einzelner Eintrag besteht aus einem *Wort* und einer Liste von Dokument-Identifikatoren derjenigen Dokumente, in denen das Wort vorkommt
- zusätzlich können weitere Informationen für die Wort-Dokument-Kombination abgespeichert werden:
  - ◆ Position des (ersten Auftretens des) Wortes im Text
  - ◆ Häufigkeit des Wortes im Text

# Invertierte Listen

---



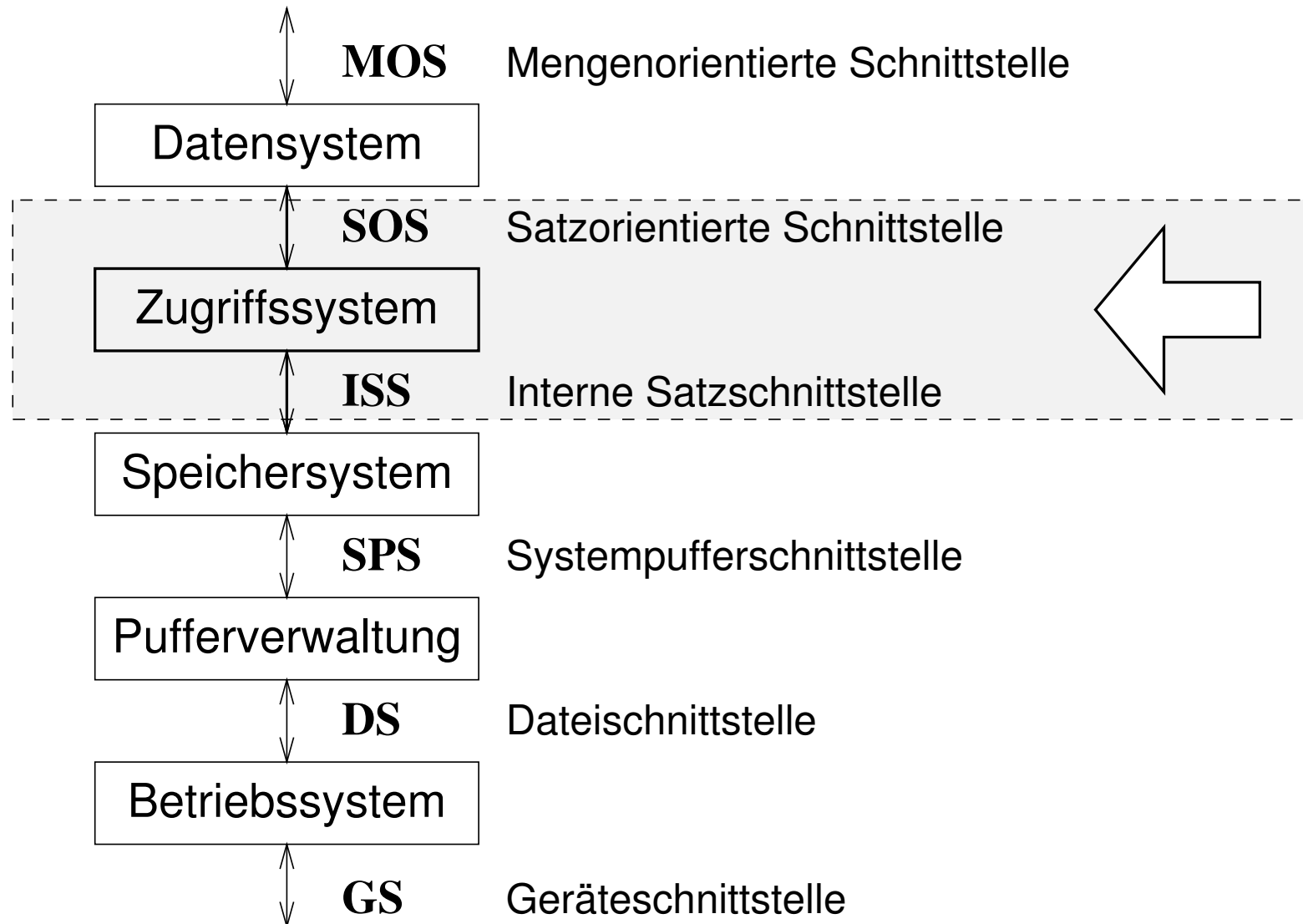
# 6. Basialgorithmen für DB-Operationen

---

- Datenbankparameter
- Komplexität von Grundalgorithmen
- Unäre Operationen (Scan, Selektion, Projektion)
- Binäre Operationen: Mengenoperationen
- Berechnung von Verbunden

# Einordnung

---



# Datenbankparameter

---

- Komplexitätsbetrachtungen ( $O(n^2)$ )
- Aufwandsabschätzungen (konkret)
- Datenbankparameter als Grundlage
- müssen im Katalog des Datenbanksystems gespeichert werden

# Datenbankparameter (II)

---

- $n_r$ : Anzahl Tupel in Relation  $r$
- $b_r$ : Anzahl von Blöcken (Seiten), die Tupel aus  $r$  beinhalten
- $s_r$ : (mittlere) Größe von Tupeln aus  $r$  ( $s$  für *size*)
- $f_r$ : *Blockungsfaktor* (Tupel aus  $r$  pro Block)

$$f_r = \frac{b_s}{s_r},$$

mit  $b_s$  Blockgröße

- Tupel einer Relation kompakt in Blöcken:

$$b_r = \left\lceil \frac{n_r}{f_r} \right\rceil$$

# Datenbankparameter (III)

---

- $V(A, r)$ : Anzahl verschiedener Werte für das Attribut  $A$  in der Relation  $r$  ( $V$  für *values*):  $V(A, r) = |\pi_A(r)|$
- $A$  Primärschlüssel:  $V(A, r) = n_r$
- $SC(A, r)$ : *Selektionskardinalität (selection cardinality)*; durchschnittliche Anzahl von Ergebnistupeln bei  $\sigma_{A=x}(r)$  für  $x \in \pi_A(r)$
- Schlüsselattribut  $A$ :  $SC(A, r) = 1$
- Allgemein:

$$SC(A, r) = \frac{n_r}{V(A, r)}$$

Weiterhin: Verzweigungsgrad bei B-Baum-Indexen, Höhe des Baums, Anzahl von Blätterknoten



# Komplexität von Grundalgorithmen

---

## Grundannahmen

- Indexe  $B^+$ -Bäume
- dominierender Kostenfaktor: Blockzugriff
- Zugriff auf Hintergrundspeicher auch für Zwischenrelationen
- Zwischenrelationen zunächst für jede Grundoperation
- Zwischenrelationen hoffentlich zum großen Teil im Puffer
- einige Operationen (Mengenoperationen) auf Adreßmengen (TID-Listen)

# Hauptspeicheralgorithmen

---

wichtig für den Durchsatz des Gesamtsystems, da sie sehr oft eingesetzt werden

- *Tupelvergleich*

(Duplikate erkennen, Sortierordnung angeben, ...)

iterativ durch Vergleich der Einzelattribute, Attribute mit großer Selektivität zuerst

- *TID-Zugriff*

TID innerhalb des Hauptspeichers: übliche Vorgehensweise bei der Auflösung indirekter Adressen

# Zugriffe auf Datensätze

---

- *Relationen*: interner Identifikator `RelID`
  - *Indexe*: interner Identifikator `IndexID`
    - ◆ *Primärindex*, etwa  $I(\text{Personen}(\text{PANr}))$   
bei  $A = a$  wird maximal ein Tupel pro Zugriff
    - ◆ *Sekundärindex*, etwa  $I(\text{Ausleihe}(\text{PANr}))$   
Bsp.:  $\text{PANr} = 4711$  liefert i.a. mehrere Tupel
- Indexzugriffe: Ergebnis TID-Listen

# Zugriffe auf Datensätze (II)

---

- **fetch-tupel** Direktzugriff auf Tupel mittels TID-Wertes holt Tupel in *Tupel-Puffer*

**fetch-tupel**(RelID, TID)  $\rightarrow$  *Tupel-Puffer*

- **fetch-TID**: TID zu (Primärschlüssel-)Attributwert bestimmen

**fetch-TID**(IndexID, Attributwert)  $\rightarrow$  TID

- weiterhin auf Relationen und Indexen: *Scans*

# Beispiel in SQL

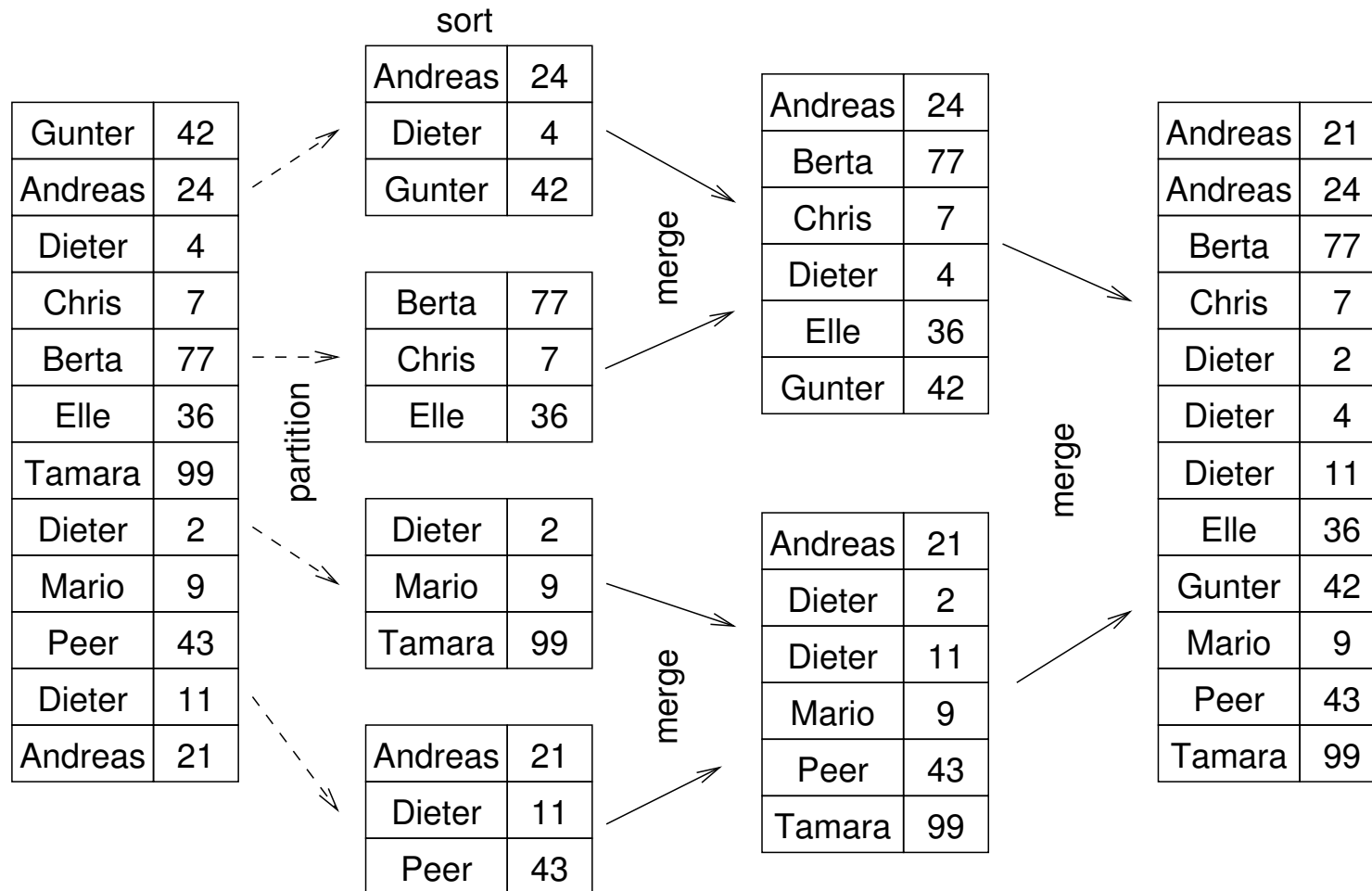
---

```
select *  
from KUNDE  
where KName ='Meier'
```

- Gleichheitsanfrage über einen Schlüssel
- **put**: hier Anzeige des Ergebnisses

```
aktuellerTID :=  
    fetch-TID (KUNDE-KName-Index, 'Meyer') ;  
aktuellerPuffer :=  
    fetch-tupel (KUNDE-RelationID, aktuellerTID) ;  
put (aktuellerPuffer) ;
```

# Externe Sortieralgorithmen



Externes Sortieren durch Mischen; Komplexität  $O(n \log n)$   
Vertauschoperationen

# Unäre Operationen

---

Scan durchläuft Tupel einer Relation

- *Relationen-Scan (full table scan)* durchläuft alle Tupel einer Relation in beliebiger Reihenfolge

Aufwand:  $b_r$

- *Index-Scan* nutzt Index zum Auslesen der Tupel in Sortierreihenfolge

Aufwand: Anzahl der Tupel plus Höhe des Indexes

Vergleich

- Relationen-Scan besser durch Ausnutzung der Blockung
- Index-Scan besser, falls wenige Daten benötigt, aber schlechter beim Auslesen vieler Tupel

# Operationen auf Scans

---

- Relationen-Scan öffnen  
**open-rel-scan**(RelationenID)  $\rightarrow$  ScanID  
liefert ScanID zurück, die bei folgenden Operationen zur Identifikation genutzt wird
- Index-Scan initialisieren  
**open-index-scan**(IndexID, Min, Max)  $\rightarrow$  ScanID  
liefert ScanID zurück; Min und Max bestimmen Bereich einer Bereichsanfrage
- **next-TID** liefert nächsten TID; Scan-Cursor weitersetzen
- **end-of-scan** liefert **true**, falls kein TID mehr im Scan abzuarbeiten
- **close-scan** schließt Scan



# Beispiel: Scan

---

```
select *  
from Personen  
where Nachname between 'Heuer' and  
        'Jagellowsk'
```

# Beispiel: Relationen-Scan

---

```
aktuellerScanID := open-rel-scan(Personen-RelationID);
aktuellerTID := next-TID(aktuellerScanID);
while not end-of-scan(aktuellerScanID) do
begin
    aktuellerPuffer :=
        fetch-tupel(Personen-RelationID, aktuellerTID);
    if aktuellerPuffer.Nachname >= 'Heuer'
        and aktuellerPuffer.Nachname <= 'Jagellowsk'
    then put (aktuellerPuffer);
    endif;
    aktuellerTID := next-TID(aktuellerScanID);
end;
close (aktuellerScanID);
```

# Beispiel: Index-Scan

---

```
aktuellerScanID :=  
    open-index-scan (Personen-Nachname-IndexID,  
        'Heuer', 'Jagellowsk') ;  
aktuellerTID := next-TID (aktuellerScanID) ;  
while not end-of-scan (aktuellerScanID) do  
begin  
    aktuellerPuffer :=  
        fetch-tupel (Personen-RelationID, aktuellerTID) ;  
    put (aktuellerPuffer) ;  
    aktuellerTID := next-TID (aktuellerScanID) ;  
end;  
close (aktuellerScanID) ;
```

# Selektion

---

- *exakte Suche, Bereichsselektionen*, komplex zusammengesetzte Selektionskriterien
- zusammengesetztes Prädikat  $\varphi$  aus atomaren Prädikaten (exakte Suche, Bereichsanfrage) mit **and**, **or**, **not**

## Tupelweises Vorgehen

- Gegeben  $\sigma_{\varphi}(r)$
- Relationen-Scan: für alle  $t \in r$  auswerten  $\varphi(t)$
- Aufwand  $O(n_r)$ , genauer  $b_r$

# Selektion: Konjunktive Normalform

---

- Zugriffspfade bei komplexen Prädikaten einsetzen  $\Rightarrow \varphi$  analysieren und geeignet umformen
- etwa  $\varphi$  in konjunktive Normalform KNF überführen; bestehend aus *Konjunkten*
- heuristisch Konjunkt auswählen, das sich besonders gut durch Indexe auswerten läßt (etwa bei  $A = c$  und über  $A$  Index)
- ausgewähltes Konjunkt auswerten; für Ergebnis-TID-Liste andere Konjunkte tupelweise
- oder mehrere geeignete Konjunkte auswerten und die sich ergebenden TID-Listen schneiden

# Selektion: Filtermethoden

---

- bei *Filtermethode* alle Bedingungen auf **true** setzen, die nicht durch eine Zugriffsmethode unterstützt werden
- resultierendes Prädikat:  $\varphi'$ .
- $r' = \sigma_{\varphi'}(r)$  unter Ausnutzung der Indexe auswerten
- $\sigma_{\varphi}(r')$  auf dem (hoffentlich viel kleineren) Zwischenergebnis  $r'$  mittels tupelweisem Vorgehen auswerten
- Filtermethoden nur gut, wenn  $\varphi'$  tatsächlich Datenvolumen reduziert (Vorsicht bei Disjunktionen)

# Projektion

---

- Relationenalgebra: mit Duplikateliminierung
- SQL: keine Duplikateliminierung, wenn nicht mit **distinct** gefordert (modifizierter Scan)
- mit Duplikateliminierung:
  - ◆ sortierte Ausgabe eines Indexes hilft bei der Duplikateliminierung
  - ◆ Projektion auf indexierte Attribute ohne Zugriff auf gespeicherte Tupel

# Projektion (II)

---

- Projektion  $\pi_X(r)$ :
  1.  $r$  nach  $X$  sortieren
  2.  $t \in r$  werden in das Ergebnis aufnehmen, für die  $t(X) \neq \mathbf{previous}(t(X))$  gilt
- Zeitaufwand:  $O(n_r \log n_r)$
- Falls  $r$  schon sortiert nach  $X$ :  $O(n_r)$
- Schlüssel  $K \subseteq X$ :  $O(n_r)$



# Scan-Semantik

---

- bei Scan-basierten (positionalen) Änderungsoperationen: Festlegung einer Scan-Semantik  $\leadsto$  Wirkungsweise nachfolgender Scan-Operationen
- Beispiel: Löschen des aktuellen Satzes
- Zustände: vor dem ersten Satz, auf einem Satz, in Lücke zwischen zwei Sätzen, hinter dem letzten Satz, in leerer Menge
- weiterhin: Übergangsregeln für Zustände

# Scan-Semantik (II)

---

Halloween-Problem (System R):

- SQL-Anweisung:

```
update employee e  
  set salary = salary * 1.05
```

- satzorientierte Auswertung mittels Index-Scan über  $I_{\text{employee}}(\text{salary})$  und sofortige Index-Aktualisierung
- ohne besondere Vorkehrungen: unendliche Anzahl von Gehaltserhöhungen

# Binäre Operationen: Mengenoperationen

---

Binäre Operationen meist auf Basis von tupelweisem Vergleich der einzelnen Tupelmengen

- *Nested-Loops-Technik* oder *Schleifeniteration*
  - ◆ für jedes Tupel einer äußeren Relation  $s$  wird die innere Relation  $r$  komplett durchlaufen
  - ◆ Aufwand:  $O(n_s * n_r)$
- *Merge-Technik* oder *Mischmethode*
  - ◆  $r$  und  $s$  (sortiert) schrittweise in der vorgegebenen Tupelreihenfolge durchlaufen
  - ◆ Aufwand:  $O(n_s + n_r)$
  - ◆ Falls Sortierung noch vorzunehmen:  
*Sort-Merge-Technik*
  - ◆ Aufwand  $n_r \log n_r$  und/oder  $n_s \log n_s$

# Mengenoperationen (II)

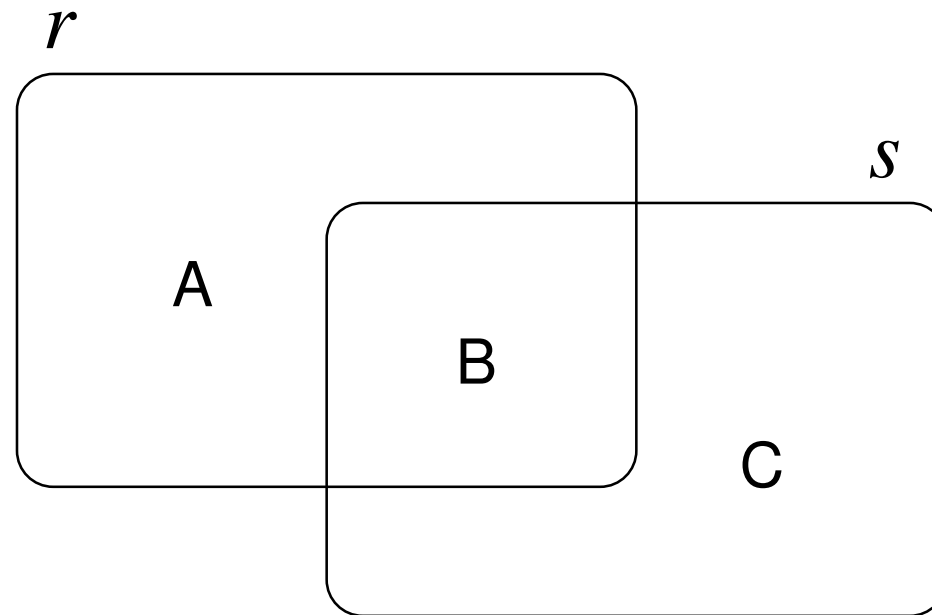
---

## ■ *Hash-Methoden*

- ◆ kleinere der beiden Relationen in Hash-Tabelle
- ◆ Tupel der zweiten Relation finden ihren Vergleichspartner mittels Hash-Funktion
- ◆ idealerweise Aufwand  $O(n_s + n_r)$

# Klassen binärer Operationen

---



# Klassen binärer Operationen (II)

Ergebnis- extensionen	Übereinstimmung auf allen Attributen	Übereinstimmung auf einigen Attributen
$A$	Differenz $r - s$	Anti-Semi- Verbund
$B$	Schnitt $r \cap s$	Verbund, Semi- Verbund
$C$	Differenz $s - r$	Anti-Semi- Verbund
$A \cup B$		Left Outer Join
$A \cup C$	symmetrische Differenz $(r - s) \cup (s - r)$	Anti-Verbund
$B \cup C$		Right Outer Join
$A \cup B \cup C$	Vereinigung $r \cup s$	Full Outer Join

# Vereinigung mit Duplikateliminierung

---

## Vereinigung durch Einfügen

- Variante der Nested-Loops-Methoden
- Kopie einer der beiden Relationen  $r_2$  unter dem Namen  $r'_2$  anlegen, dann Tupel  $t_1 \in r_1$  in  $r'_2$  einfügen (Zeitaufwand abhängig von Organisationsform der Kopie)

## Spezialtechniken für die Vereinigung

- $r$  und  $s$  verketteten
- Projektion auf alle Attribute der verketteten Relation

Zeitaufwand:  $O((n_r + n_s) \times \log(n_r + n_s))$  (wie Projektion)

# Vereinigung (II)

---

Vereinigung durch Merge-Techniken (**merge-union**)

1.  $r$  und  $s$  sortieren, falls nicht bereits sortiert
2.  $r$  und  $s$  mischen
  - $t_r \in r$  kleiner als  $t_s \in s$ :  $t_r$  in das Ergebnis, nächstes  $t_r \in r$  lesen
  - $t_r \in r$  größer als  $t_s \in s$ :  $t_s$  in das Ergebnis, nächstes  $t_s \in s$  lesen
  - $t_s = t_r$ :  $t_r$  in das Ergebnis, nächste  $t_r \in r$  bzw.  $t_s \in s$  lesen
- Zeitaufwand:  $O(n_r \times \log n_r + n_s \times \log n_s)$  mit Sortierung,  $O(n_r + n_s)$  ohne Sortierung



# Berechnung von Verbunden

---

## Varianten

- Nested-Loops-Verbund
- Block-Nested-Loops-Verbund
- Merge-Join
- Hash-Verbund
- ...

# Nested-Loops-Verbund

---

doppelte Schleife iteriert über alle  $t_1 \in r$  und alle  $t_2 \in s$  bei einer Operation  $r \bowtie s$

$r \bowtie_{\varphi} s$ :

```
for each  $t_r \in r$  do  
begin  
    for each  $t_s \in s$  do  
    begin  
        if  $\varphi(t_r, t_s)$  then put $(t_r \cdot t_s)$  endif  
    end  
end
```

# Nested-Loops-Verbund mit Scan

---

```
R1ScanID := open-rel-scan(R1ID);
R1TID := next-TID(R1ScanID);
while not end-of-scan(R1ScanID) do
begin
    R1Puffer := fetch-tupel(R1ID,R1TID);
    R2ScanID := open-rel-scan(R2ID);
    R2TID := next-TID(R2ScanID);
    while not end-of-scan(R2ScanID) do
    begin
        .../* Scan über innere Relation */
    end;
    close (R2ScanID);
    R1TID := next-TID(R1ScanID);
end;
close (R1ScanID);
```

# Nested-Loops-Verbund mit Scan II

---

```
/* Scan über innere Relation */  
R2Puffer := fetch-tupel(R2ID, R2TID);  
if R1Puffer.X = R2Puffer.Y  
then insert into ERG  
    (R1.Puffer.A1, ..., R1.Puffer.An, R1.Puffer.X,  
     R2.Puffer.B1, ..., R1.Puffer.Bm);  
endif;  
R2TID := next-TID(R2ScanID);
```

Verbesserung: Nested-Loops-Verbund verbindet alle  $t_1 \in r$  mit Ergebnis von  $\sigma_{X=t_1(X)}(s)$  (gut bei Index auf  $X$  in  $r_2$ )

# Block-Nested-Loops-Verbund

---

statt über Tupel über Blöcke iterieren

```
for each Block  $B_r$  of  $r$  do  
begin  
  for each Block  $B_s$  of  $s$  do  
    begin  
      for each Tupel  $t_r \in B_r$  do  
        begin  
          for each Tupel  $t_s \in B_s$  do  
            begin  
              if  $\varphi(t_r, t_s)$  then put( $t_r \cdot t_s$ ) endif  
            end  
          end  
        end  
      end  
    end  
  end  
end
```

Aufwand:  $b_r * b_s$

# Merge-Techniken

---

$X := R \cap S$ ; falls nicht bereits sortiert, zuerst Sortierung von  $r$  und  $s$  nach  $X$

1.  $t_r(X) < t_s(X)$ , nächstes  $t_r \in r$  lesen
2.  $t_r(X) > t_s(X)$ , nächstes  $t_s \in s$  lesen
3.  $t_r(X) = t_s(X)$ ,  $t_r$  mit  $t_s$  und allen Nachfolgern von  $t_s$ , die auf  $X$  mit  $t_s$  gleich, verbinden
4. beim ersten  $t'_s \in s$  mit  $t'_s(X) \neq t_s(X)$  beginnend mit ursprünglichem  $t_s$  mit den Nachfolgern  $t'_r$  von  $t_r$  wiederholen, solange  $t_r(X) = t'_r(X)$  gilt

# Merge-Techniken: Aufwand

---

- alle Tupel haben den selben  $X$ -Wert:  $O(n_r \times n_s)$
- $X$  Schlüssel von  $R$  oder  $S$ :  $O(n_r \log n_r + n_s \log n_s)$
- bei vorsortierten Relationen sogar:  $O(n_r + n_s)$

# Merge-Join mit Scan

---

- Verbund-Attribute auf beiden Relationen  
Schlüsseleigenschaft
- **min** (X) und **max** (X) : minimaler bzw. maximaler  
gespeicherter Wert für X



# Merge-Join mit Scan (II)

---

```
R1ScanID := open-index-scan (R1XIndexID,  
    min (X) , max (X) ) ;  
R1TID := next-TID (R1ScanID) ;  
R1Puffer := fetch-tupel (R1ID, R1TID) ;  
R2ScanID := open-index-scan (R2YIndexID,  
    min (Y) , max (Y) ) ;  
R2TID := next-TID (R2ScanID) ;  
R2Puffer := fetch-tupel (R2ID, R2TID) ;  
while not end-of-scan (R1ScanID)  
    and not end-of-scan (R2ScanID) do  
begin  
    .../* merge */  
end;  
close (R1ScanID) ;  
close (R2ScanID) ;
```

# Merge-Join mit Scan (III)

---

```
/* merge */
if R1Puffer.X < R2Puffer.Y
then R1TID := next-TID(R1ScanID);
    R1Puffer := fetch-tupel(R1ID,R1TID);
else if R1Puffer.X > R2Puffer.y
    then R2TID := next-TID(R2ScanID);
        R2Puffer := fetch-tupel(R2ID,R2TID);
    else insert into ERG
        (R1.Puffer.A1, ..., R1.Puffer.An, R1.Puffer.X,
         R2.Puffer.B1, ..., R1.Puffer.Bm);
        R1TID := next-TID(R1ScanID);
        R1Puffer := fetch-tupel(R1ID,R1TID);
        R2TID := next-TID(R2ScanID);
        R2Puffer := fetch-tupel(R2ID,R2TID);
    endif;
endif;
```

# Verbund durch Hashing

---

- Idee:

- ◆ Ausnutzung des verfügbaren Hauptspeichers zur Minimierung der Externspeicherzugriffe
- ◆ Finden der Verbundpartner durch Hashing
- ◆ Anfragen der Form  $r \bowtie_{r.A=s.B} s$

# Classic Hashing

---

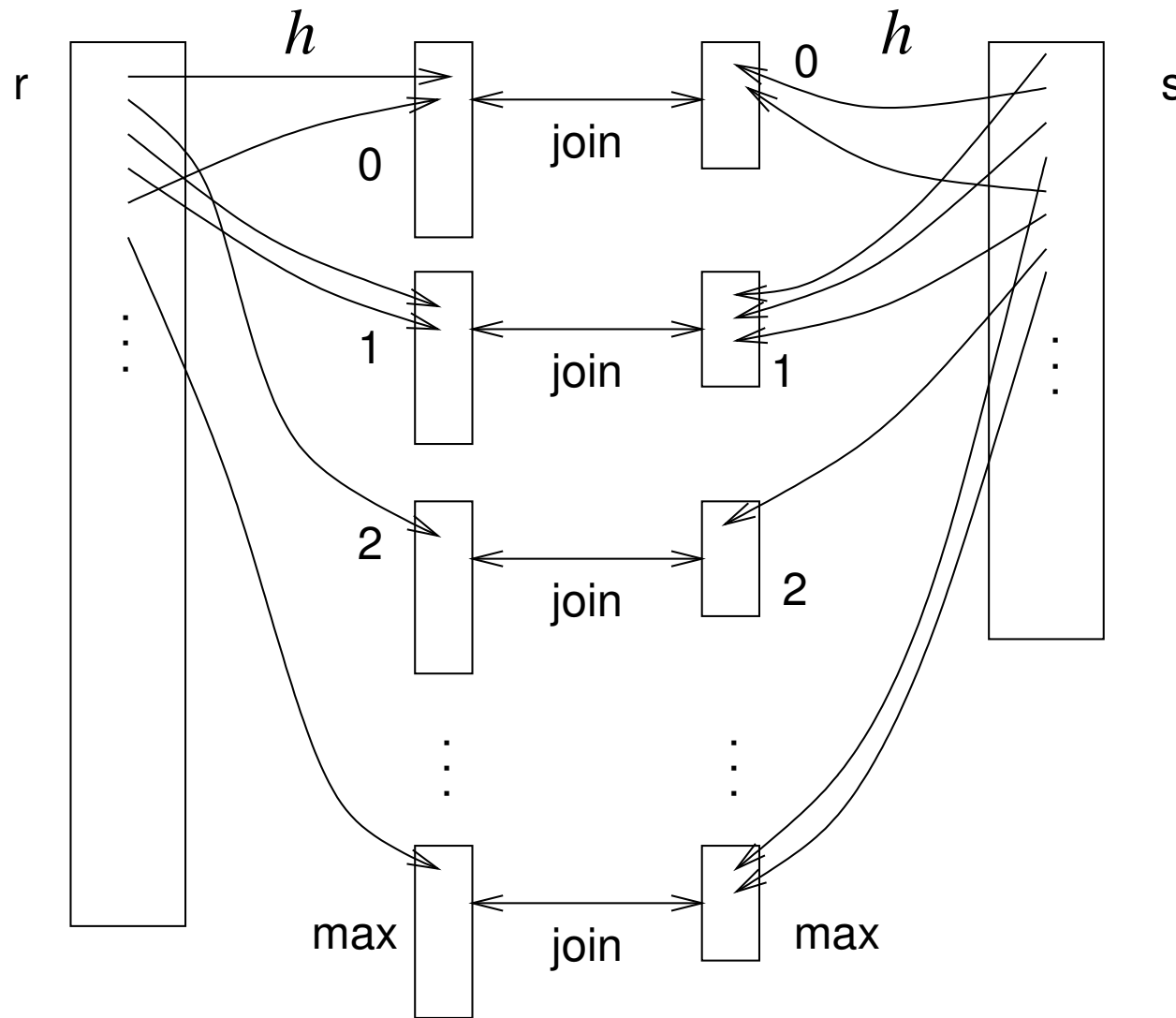
- Vorbereitung: kleinere Relation wird  $r$
- Ablauf
  1. Tupel von  $r$  mittels Scan in Hauptspeicher lesen und mittels Hashfunktion  $h(r.A)$  in Hashtabelle  $H$  einordnen
  2. wenn  $H$  voll (oder  $r$  vollständig gelesen):  
Scan über  $S$  und mit  $h(s.B)$  Verbundpartner suchen
  3. falls Scan über  $r$  nicht abgeschlossen:  
 $H$  neu aufbauen und erneuten Scan über  $S$  durchführen
- Aufwand:  $O(b_r + p * b_s)$  mit  $p$  ist Anzahl der Scans über  $S$

# Partitionierung mittels Hashfunktion

---

- Tupel aus  $r$  und  $s$  über  $X$  in gemeinsame Datei mit  $k$  Blöcken (*Buckets*) „gehasht“
- Tupel in gleichen Buckets durch Verbundalgorithmus verbinden

# Partitionierung mittels Hashfunktion (II)



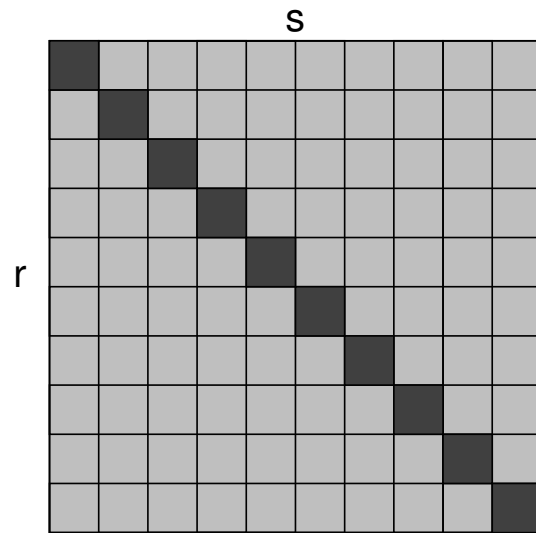
# Partitionierung mittels Hashfunktion (III)

---

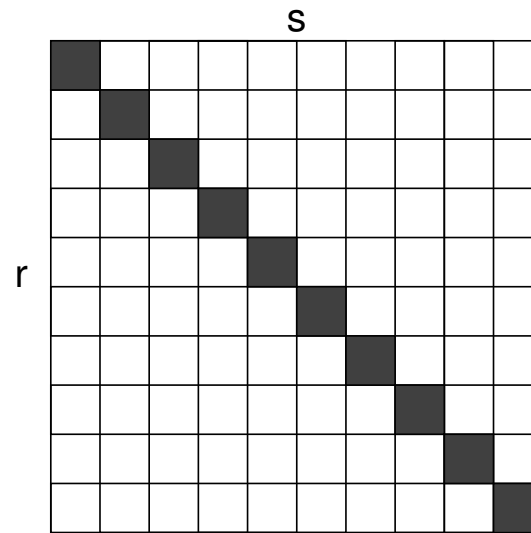
```
for each  $t_r$  in  $r$  do
  begin
     $i := h(t_r(X));$ 
     $H_i^r := H_i^r \cup t_r(X);$ 
  end;
for each  $t_s$  in  $s$  do
  begin
     $i := h(t_s(X));$ 
     $H_i^s := H_i^s \cup t_s(X);$ 
  end;
for each  $k$  in  $0 \dots \max$  do
   $H_k^r \bowtie H_k^s;$ 
```

# Vergleich der Techniken

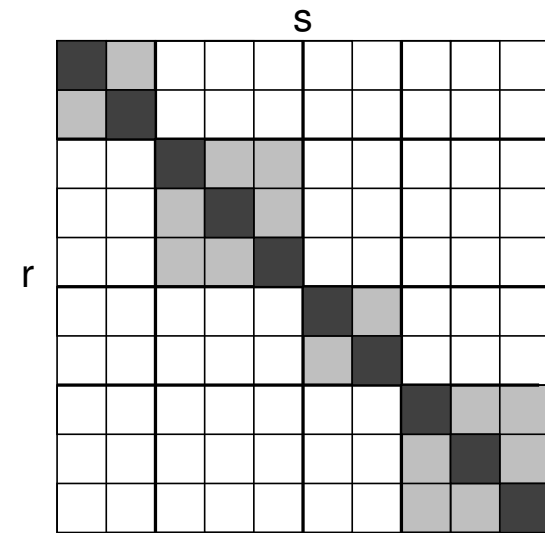
---



Nested-Loops-Join



Merge-Join



Hash-Join



# Aggregation & Gruppierung

---

- Anfragen:

```
select A, count ( * )  
from T  
group by A
```

- Algebraoperator:  $\gamma_{\text{count}(*),A}(r(t))$

- Implementierungsvarianten:

- ◆ Nested Loops
- ◆ Sortierung
- ◆ Hashing

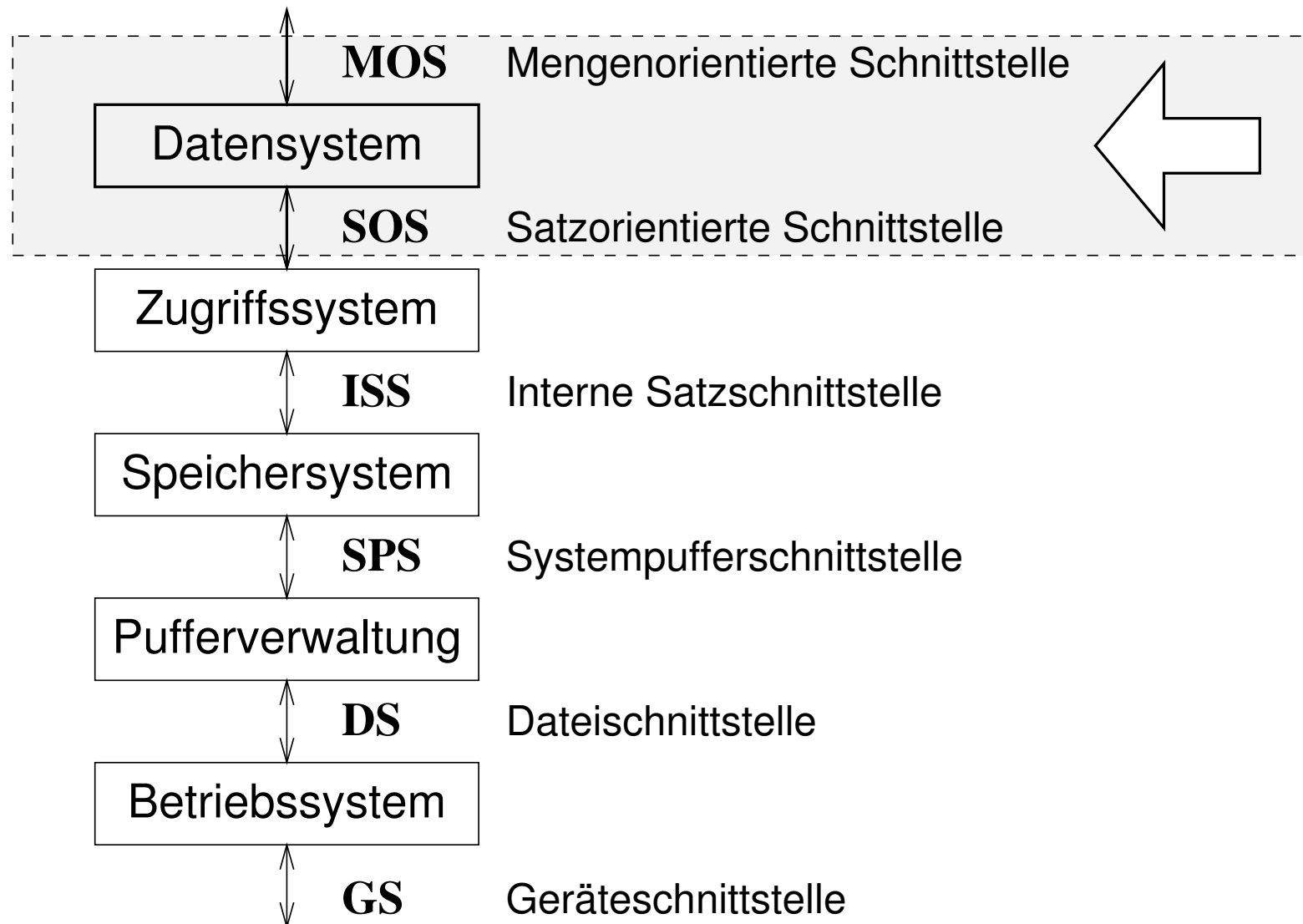
# 7. Optimierung von Anfragen

---

- Grundprinzipien, motivierende Beispiele
- Phasen der Anfrageverarbeitung
- Übersetzung von SQL in Relationenalgebra
- Logische Optimierung (algebraisch, Tableau)
- Interne Optimierung
- Kostenbasierte Auswahl

# Einordnung

---



# Grundprinzipien

---

## Basissprachen

- SQL
- Relationenkalküle
- hier: Relationenalgebra

## Ziel der Optimierung

- möglichst schnelle Anfragebearbeitung  $\Rightarrow$
- möglichst wenig Seitenzugriffe bei der Anfragebearbeitung  $\Rightarrow$
- möglichst in allen Operationen so wenig wie möglich Seiten (Tupel) berücksichtigen

# Teilziele einer Optimierung

---

1. Selektionen so früh wie möglich
2. Basisoperationen zusammenfassen und ohne Zwischenspeicherung realisieren
3. Redundante Operationen, Idempotenzen oder leere Zwischenrelationen entfernen
4. Zusammenfassen gleicher Teilausdrücke:  
Wiederverwendung von Zwischenergebnissen

# Beispiel

---

KUNDE { KName, Kadr, Kto }  
AUFTRAG { KName, Ware, Menge }

```
select KUNDE.KName, Kto  
from KUNDE, AUFTRAG  
where KUNDE.KName = AUFTRAG.KName  
       and Ware = 'Kaffee'
```

- Relation KUNDE: 100 Tupel; eine Seite: 5 Tupel
- Relation AUFTRAG: 10.000 Tupel; eine Seite: 10 Tupel
- 50 der Aufträge betreffen Kaffee
- Tupel der Form (KName, Kto): 50 auf eine Seite
- 3 Zeilen von KUNDE  $\times$  AUFTRAG auf eine Seite
- Puffer für jede Relation Größe 1, keine Spannsätze

# Direkte Auswertung

---

1.  $R_1 := \text{KUNDE} \times \text{AUFTRAG}$

Seitenzugriffe:

■  $l : (100/5 * 10.000/10) = 20.000$

■  $s : (100 * 10.000)/3 = 333.000 \text{ (ca.)}$

2.  $R_2 := \sigma_{\text{SEL}}(R_1)$

■  $l : 333.000 \text{ (ca.)}$

■  $s : 50/3 = 17 \text{ (ca.)}$

3.  $ERG := \pi_{\text{PROJ}}(R_2)$

■  $l : 17$

■  $s : 1$

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

# Optimierte Auswertung

---

1.  $R_1 := \sigma_{\text{Ware}='Kaffee'}(\text{AUFTRAG})$

■  $l : 10.000/10 = 1.000$

■  $s : 50/10 = 5$

2.  $R_2 := \text{KUNDE} \bowtie_{\text{KName}=\text{KName}} R_1$

■  $l : 100/5 * 5 = 100$

■  $s : 50/3 = 17$

3.  $ERG := \pi_{\text{PROJ}}(R_2)$

■  $l : 17$

■  $s : 1$

ca. 1.140 Seitenzugriffe (Faktor 500 verbessert)



# Auswertung mit Indexausnutzung

---

Indexe  $I(\text{AUFTRAG}(\text{Ware}))$  und  $I(\text{KUNDE}(\text{KName}))$

1.  $R_1 := \sigma_{\text{Ware}='Kaffee'}(\text{AUFTRAG})$  über  
 $I(\text{AUFTRAG}(\text{Ware}))$

■  $l$  : minimal 5, maximal 50;  $s$  :  $50/10 = 5$

2.  $R_2 := \text{sortiere } R_1 \text{ nach KName}$

■  $l + s$  :  $5 * \log 5 = 15$  (ca.)

3.  $R_3 := \text{KUNDE} \bowtie_{\text{KName}=\text{KName}} R_2$

■  $l$  :  $100/5 + 5 = 25$ ;  $s$  :  $50/3 = 17$

4.  $ERG := \pi_{\text{PROJ}}(R_3)$

■  $l$  : 17;  $s$  : 1

maximal ca. 130 und minimal ca. 85 Seitenzugriffe

# Gegenüberstellung der Varianten

---

Variante der Ausführung	Lese- und Schreibzugriffe	Seiten für Zwischenergebnisse
direkte Auswertung	ca. 687.000	ca. 333.000
optimierte Auswertung	ca. 1.140	17
Auswertung mit Index	min. 85	17
mit Pipelining	max. 130 51 bis 96	17 5 (plus sortieren)

# Phasen der Anfrageverarbeitung

---

1. *Übersetzung und Sichtexpansion*
  - in Anfrageplan arithmetische Ausdrücke vereinfachen
  - Unteranfragen auflösen
  - Einsetzen der Sichtdefinition
2. *Logische oder auch algebraische Optimierung*
  - Anfrageplan unabhängig von der konkreten Speicherungsform umformen; etwa Hineinziehen von Selektionen in andere Operationen

# Phasen der Anfrageverarbeitung II

---

## 3. *Interne Optimierung*

- konkrete Speicherungstechniken (Indexe, Cluster) berücksichtigen
- Algorithmen auswählen
- mehrere alternative interne Pläne

## 4. *Kostenbasierte Auswahl*

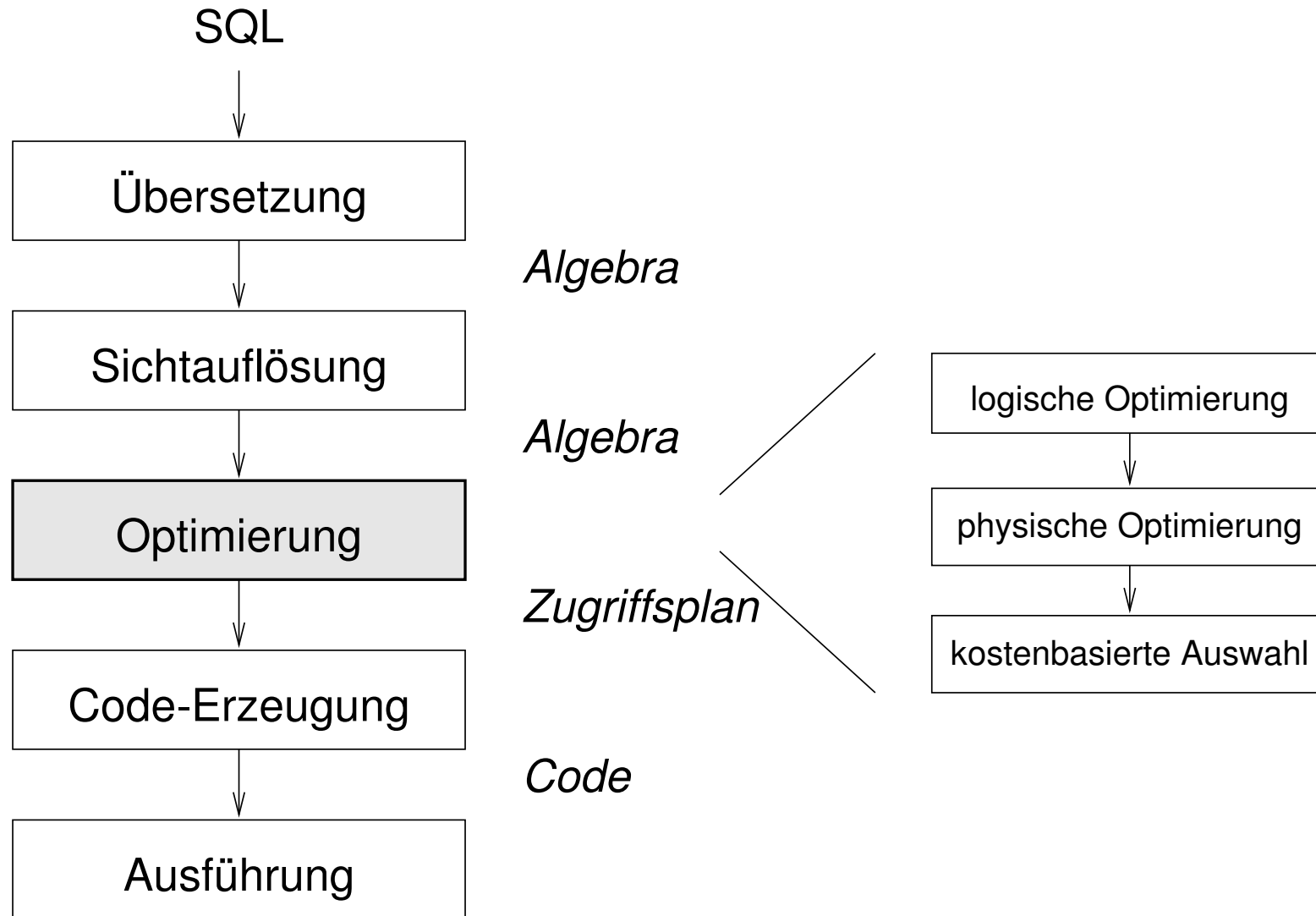
- Statistikinformationen (Größe von Tabellen, Selektivität von Attributen) für die Auswahl eines konkreten internen Planes nutzen

## 5. *Code-Erzeugung*

- Umwandlung des Zugriffsplans in ausführbaren Code

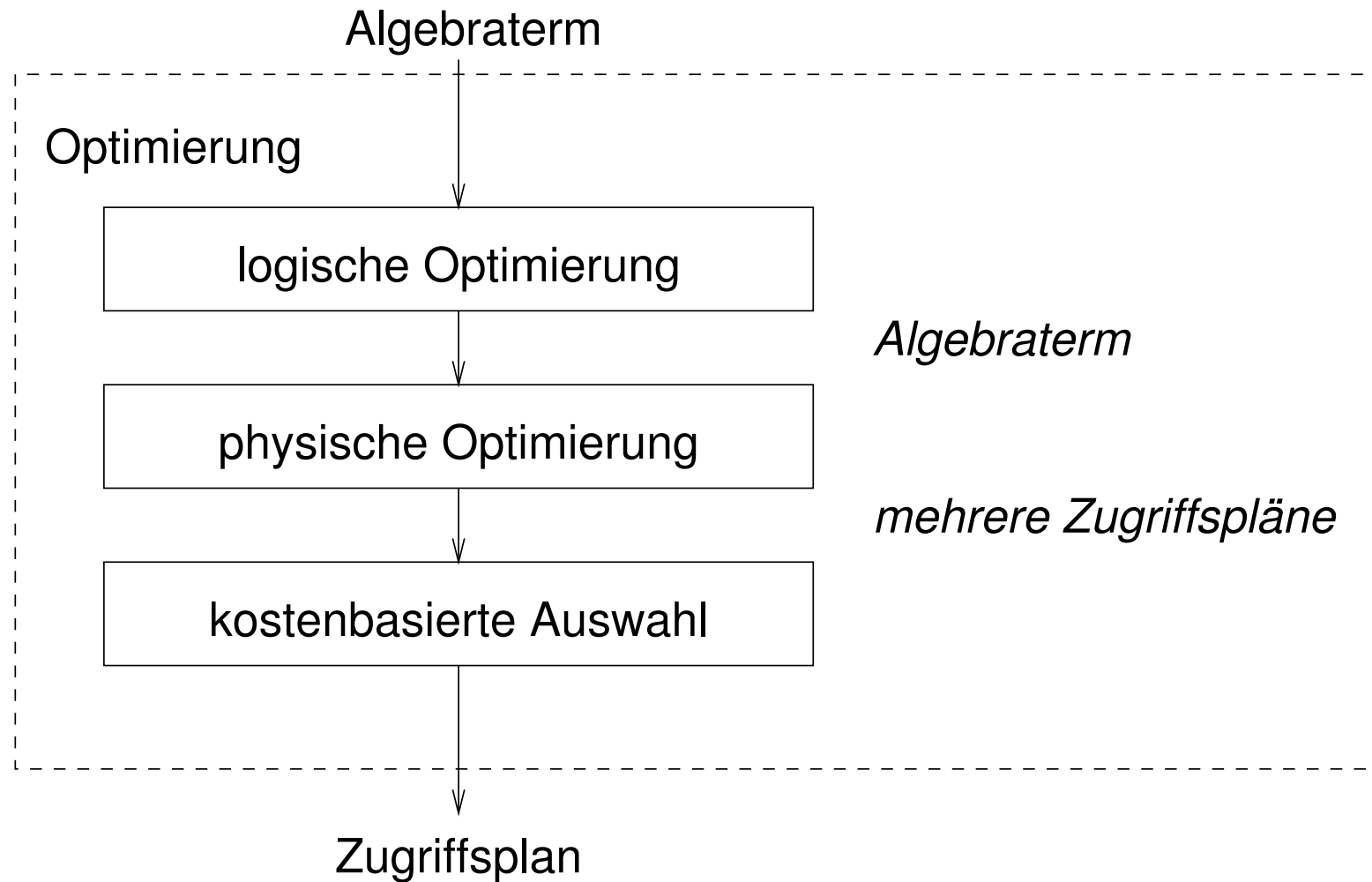
# Ablauf und Sprachen

---



# Phasen der Optimierung

---



# Übersetzung in Relationenalgebra

---

```
select A1, ..., Am  
from R1, R2, ..., Rn  
where F
```

Umsetzung in Relationenalgebra:

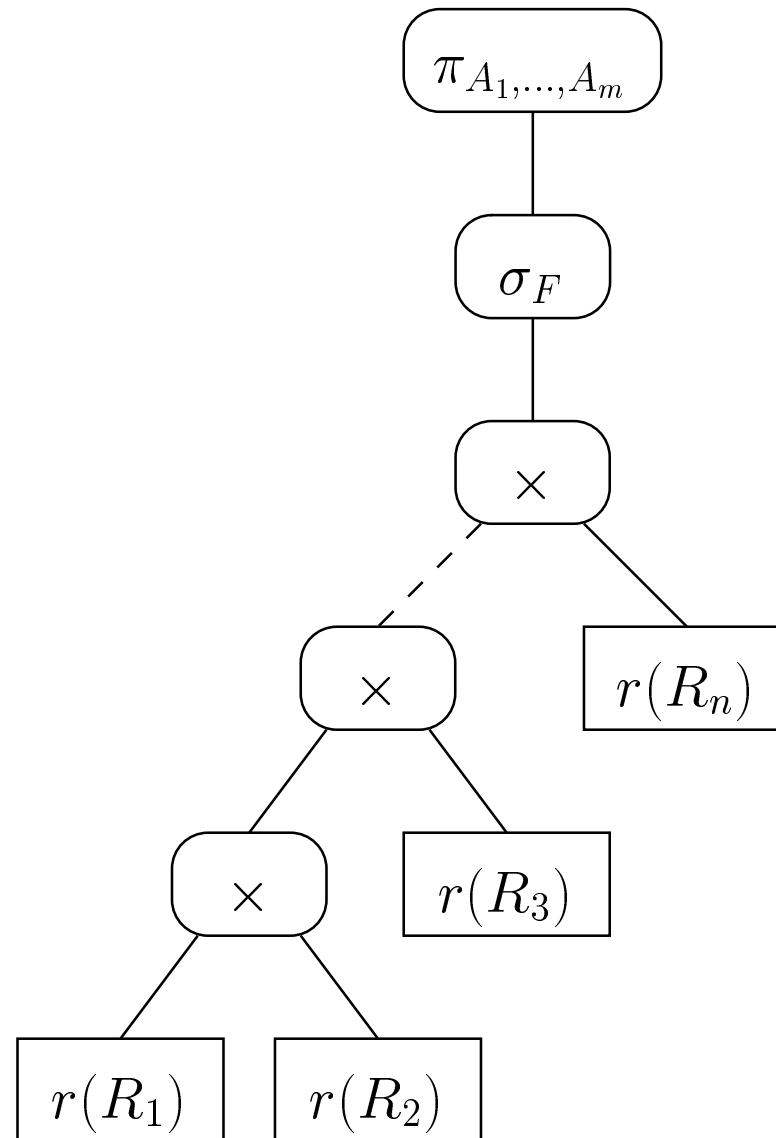
$$\pi_{A_1, \dots, A_m}(\sigma_F(r(R_1) \times r(R_2) \times r(R_3) \times \dots \times r(R_n)))$$

Anfragebaum (folgende Folie) verbessern gemäß:

- *Erkennen von Verbunden* statt Kreuzprodukten
- *Auflösung von Unteranfragen* (**not exists**-Anfragen in Differenz)
- SQL-Konstrukte, die in der Relationenalgebra kein Gegenstück haben: **group by**, **order by**, Arithmetik, Multimengensemantik

# Umsetzung des SFW-Blocks

---





# Normalisierung

---

- Vereinfachung der folgenden Optimierungsschritte durch ein einheitliches (kanonisches) Anfrageformat
- speziell für Selektions- und Verbundbedingungen
  - ◆ *konjunktive Normalform vs. disjunktive Normalform*
  - ◆ konjunktive Normalform (KNF) für einfache Prädikate  $p_{ij}$ :

$$(p_{11} \vee p_{12} \vee \cdots \vee p_{1n}) \wedge \cdots \wedge (p_{m1} \vee p_{m2} \vee \cdots \vee p_{mn})$$

- ◆ disjunktive Normalform (DNF):

$$(p_{11} \wedge p_{12} \wedge \cdots \wedge p_{1n}) \vee \cdots \vee (p_{m1} \wedge p_{m2} \wedge \cdots \wedge p_{mn})$$

- ◆ Überführung in KNF/DNF durch Anwendung von Äquivalenzbeziehungen für logische Operationen

# Normalisierung /2

---

## ■ Äquivalenzbeziehungen

◆  $p_1 \wedge p_2 \longleftrightarrow p_2 \wedge p_1$  **und**  $p_1 \vee p_2 \longleftrightarrow p_2 \vee p_1$

◆  $p_1 \wedge (p_2 \wedge p_3) \longleftrightarrow (p_1 \wedge p_2) \wedge p_3$  **und**  
 $p_1 \vee (p_2 \vee p_3) \longleftrightarrow (p_1 \vee p_2) \vee p_3$

◆  $p_1 \wedge (p_2 \vee p_3) \longleftrightarrow (p_1 \wedge p_2) \vee (p_1 \wedge p_3)$  **und**  
 $p_1 \vee (p_2 \wedge p_3) \longleftrightarrow (p_1 \vee p_2) \wedge (p_1 \vee p_3)$

◆  $\neg(p_1 \wedge p_2) \longleftrightarrow \neg p_1 \vee \neg p_2$  **und**  
 $\neg(p_1 \vee p_2) \longleftrightarrow \neg p_1 \wedge \neg p_2$

◆  $\neg(\neg p_1) \longleftrightarrow p_1$

# Normalisierung: Beispiel

---

## ■ Anfrage:

```
select * from Projekt P, Zuordnung Z
where P.PNr = Z.PNr and
      Budget > 100.000 and
      (Ort = 'MD' or Ort = 'B')
```

## ■ Selektionsbedingung in KNF:

$$P.PNr = Z.PNr \wedge Budget > 100.000 \wedge (Ort = 'MD' \vee Ort = 'B')$$

## ■ Selektionsbedingung in DNF:

$$(P.PNr = Z.PNr \wedge Budget > 100.000 \wedge Ort = 'MD') \vee \\ (P.PNr = Z.PNr \wedge Budget > 100.000 \wedge Ort = 'B')$$

# Logische Optimierung

---

- heuristische Methoden
  - ◆ etwa algebraische Optimierung
  - ◆ für Relationenalgebra + Gruppierung, ...
- exakte Methoden
  - ◆ Tableauoptimierung
  - ◆ Anzahl Verbunde minimieren
  - ◆ für spezielle Relationenalgebra-Anfragen

# Algebraische Optimierung

---

- Termersetzung von Termen der Relationenalgebra anhand von Algebraäquivalenzen
- Äquivalenzen gerichtet als Ersetzungsregeln
- heuristische Methode: Operationen verschieben, um kleinere Zwischenergebnisse zu erhalten; Redundanzen erkennen

# Prinzipien algebraischer Optimierung

---

Beispiel:

BÜCHER = { Titel, Autor, Verlag, ISBN }

VERLAGE = { Verlagsname, VerlagsAdr }

ENTLEIHER = { EntlName, EntlAdr, EntlKarte }

AUSLEIHE = { EntlKarte, ISBN, Datum }

# Entfernen redundanter Operationen

---

- bei Anfragen mit Sichten nötig

$$r(\text{LANGEWEG}) = r(\text{BÜCHER}) \bowtie$$

$$\pi_{\text{ISBN,DATUM}}(\dots\dots\dots \sigma_{\text{DATUM} < '31.12.1995'}(r(\text{AUSLEIHE})))$$

- Anfrage an Sicht:

$$\pi_{\text{TITEL}}(r(\text{BÜCHER}) \bowtie r(\text{LANGEWEG}))$$

- Sichtexpansion:

$$\pi_{\text{TITEL}}(r(\text{BÜCHER}) \bowtie r(\text{BÜCHER}) \bowtie \pi_{\dots}(\dots))$$

- Regel: Idempotenz

$r = r \bowtie r, \text{ d.h. } \bowtie \text{ ist idempotent}$

# Verschieben von Selektionen

---

$$\sigma_{\text{AUTOR}='Heuer'}(r(\text{BÜCHER}) \bowtie \pi_{\text{ISBN,DATUM}}(\dots))$$

günstiger:

$$(\sigma_{\text{AUTOR}='Heuer'}(r(\text{BÜCHER}))) \bowtie \pi_{\text{ISBN,DATUM}}(\dots)$$

Regel:

Selektion und Verbund kommutieren
-----------------------------------

nur, wenn die Attribute der Selektionsprädikate dies zulassen



# Reihenfolge von Verbunden

---

- Kenntnis der Statistikinformationen des Katalogs nötig

$$(r(\text{VERLAGE}) \bowtie r(\text{AUSLEIHE})) \bowtie r(\text{BÜCHER})$$

- erster Verbund: kartesisches Produkt, daher:

$$r(\text{VERLAGE}) \bowtie (r(\text{AUSLEIHE}) \bowtie r(\text{BÜCHER}))$$

Regel:

$\bowtie$  ist assoziativ und kommutativ

keine eindeutige Vorzugsrichtung bei der Anwendung dieser Regel (daher interne Optimierung, Kostenbasierung)

# Algebraische Regeln

---

- **KommJoin:** Operator  $\bowtie$  ist kommutativ:

$$r_1 \bowtie r_2 \longleftrightarrow r_2 \bowtie r_1$$

- **AssozJoin:** Operator  $\bowtie$  ist assoziativ:

$$(r_1 \bowtie r_2) \bowtie r_3 \longleftrightarrow r_1 \bowtie (r_2 \bowtie r_3)$$

- **ProjProj:** bei Operator  $\pi$  dominiert in der Kombination der äußere Parameter den inneren:

$$\pi_X(\pi_Y(r_1)) \longleftrightarrow \pi_X(r_1)$$

# Algebraische Regeln (II)

---

- **SelfSel**: Kombination von Prädikaten bei  $\sigma$  entspricht dem logischen Und  $\Rightarrow$  Formeln können in der Reihenfolge vertauscht werden

$$\sigma_{F_1}(\sigma_{F_2}(r_1)) \longleftrightarrow \sigma_{F_1 \wedge F_2}(r_1) \longleftrightarrow \sigma_{F_2}(\sigma_{F_1}(r_1))$$

(Ausnutzung der Kommutativität des logischen Und)

# Algebraische Regeln (III)

---

- **SelfProj**: Operatoren  $\pi$  und  $\sigma$  kommutieren, sofern das Prädikat  $F$  auf den Projektionsattributen definiert ist:

$$\sigma_F(\pi_X(r_1)) \longleftrightarrow \pi_X(\sigma_F(r_1))$$

falls  $attr(F) \subseteq X$

anderenfalls Vertauschung möglich, wenn Projektion um die notwendigen Attribute erweitert wird:

$$\pi_{X_1}(\sigma_F(\pi_{X_1 X_2}(r_1))) \longleftrightarrow \pi_{X_1}(\sigma_F(r_1))$$

falls  $attr(F) \supseteq X_2$

# Algebraische Regeln (IV)

---

- **SelJoin:** Operatoren  $\sigma$  und  $\bowtie$  kommutieren, falls Selektionsattribute alle aus einer der beiden Relationen stammen:

$$\sigma_F(r_1 \bowtie r_2) \longleftrightarrow \sigma_F(r_1) \bowtie r_2$$

$$\text{falls } attr(F) \subseteq R_1$$

falls Selektionsprädikat derart aufgesplittet werden kann, daß in  $F = F_1 \wedge F_2$  beide Teile der Konjunktion passende Attribute haben, gilt:

$$\sigma_F(r_1 \bowtie r_2) \longleftrightarrow \sigma_{F_1}(r_1) \bowtie \sigma_{F_2}(r_2)$$

$$\text{falls } attr(F_1) \subseteq R_1 \text{ und } attr(F_2) \subseteq R_2$$

# Algebraische Regeln (V)

---

- **SelJoin** (fortg.): in jeden Fall: Abspalten eines  $F_1$  mit Attributen der Relation  $R_1$ , wenn  $F_2$  Attribute von  $R_1$  und  $R_2$  betrifft:

$$\sigma_F(r_1 \bowtie r_2) \longleftrightarrow \sigma_{F_2}(\sigma_{F_1}(r_1) \bowtie r_2)$$

falls  $\text{attr}(F_1) \subseteq R_1$

# Algebraische Regeln (VI)

---

- **SelUnion**: Kommutieren von  $\sigma$  und  $\cup$ :

$$\sigma_F(r_1 \cup r_2) \longleftrightarrow \sigma_F(r_1) \cup \sigma_F(r_2)$$

- **SelDiff**: Kommutieren von  $\sigma$  und  $-$ :

$$\sigma_F(r_1 - r_2) \longleftrightarrow \sigma_F(r_1) - \sigma_F(r_2)$$

oder (da Tupel nur aus der ersten Relation herausgestrichen werden):

$$\sigma_F(r_1 - r_2) \longleftrightarrow \sigma_F(r_1) - r_2$$

# Algebraische Regeln (VII)

---

- **ProjJoin**: Kommutieren von  $\pi$  und  $\bowtie$ :

$$\pi_X(r_1 \bowtie r_2) \longleftrightarrow \pi_X(\pi_{Y_1}(r_1) \bowtie \pi_{Y_2}(r_2))$$

mit

$$Y_1 = (X \cap R_1) \cup (R_1 \cap R_2)$$

und

$$Y_2 = (X \cap R_2) \cup (R_1 \cap R_2)$$

Hineinziehen der Projektion in einen Verbund, wenn durch Berechnung von  $Y_i$  dafür gesorgt wird, daß die für den natürlichen Verbund benötigten Verbundattribute erhalten bleiben (Herausprojizieren erst nach dem Verbund)



# Algebraische Regeln (VIII)

---

- **ProjUnion:** Kommutieren von  $\pi$  und  $\cup$ :

$$\pi_X(r_1 \cup r_2) \longleftrightarrow \pi_X(r_1) \cup \pi_X(r_2)$$

- Distributivgesetz für  $\bowtie$  und  $\cup$ , Distributivgesetz für  $\bowtie$  und  $-$ , Kommutieren der Umbenennung  $\beta$  mit anderen Operationen, etc.

# Weitere Regeln

---

## ■ *Idempotenzen*

<b>IdemUnion:</b>	$r_1 \cup r_1 \longleftrightarrow r_1$
<b>IdemSchnitt:</b>	$r_1 \cap r_1 \longleftrightarrow r_1$
<b>IdemJoin:</b>	$r_1 \bowtie r_1 \longleftrightarrow r_1$
<b>IdemDiff:</b>	$r_1 - r_1 \longleftrightarrow \{\}$

## ■ Verknüpfung mit einer leeren Relation:

<b>LeerUnion:</b>	$r_1 \cup \{\} \longleftrightarrow r_1$
<b>LeerSchnitt:</b>	$r_1 \cap \{\} \longleftrightarrow \{\}$
<b>LeerJoin:</b>	$r_1 \bowtie \{\} \longleftrightarrow \{\}$
<b>LeerDiffRechts:</b>	$r_1 - \{\} \longleftrightarrow r_1$
<b>LeerDiffLinks:</b>	$\{\} - r_1 \longleftrightarrow \{\}$

- für  $\bowtie$ ,  $\cup$  und  $\cap$  gelten *Kommutativ-* und *Assoziativgesetz* (Bezeichnung: **Komm\*** und **Assoz\***)

# Ein einfacher Optimierungsalgorithmus

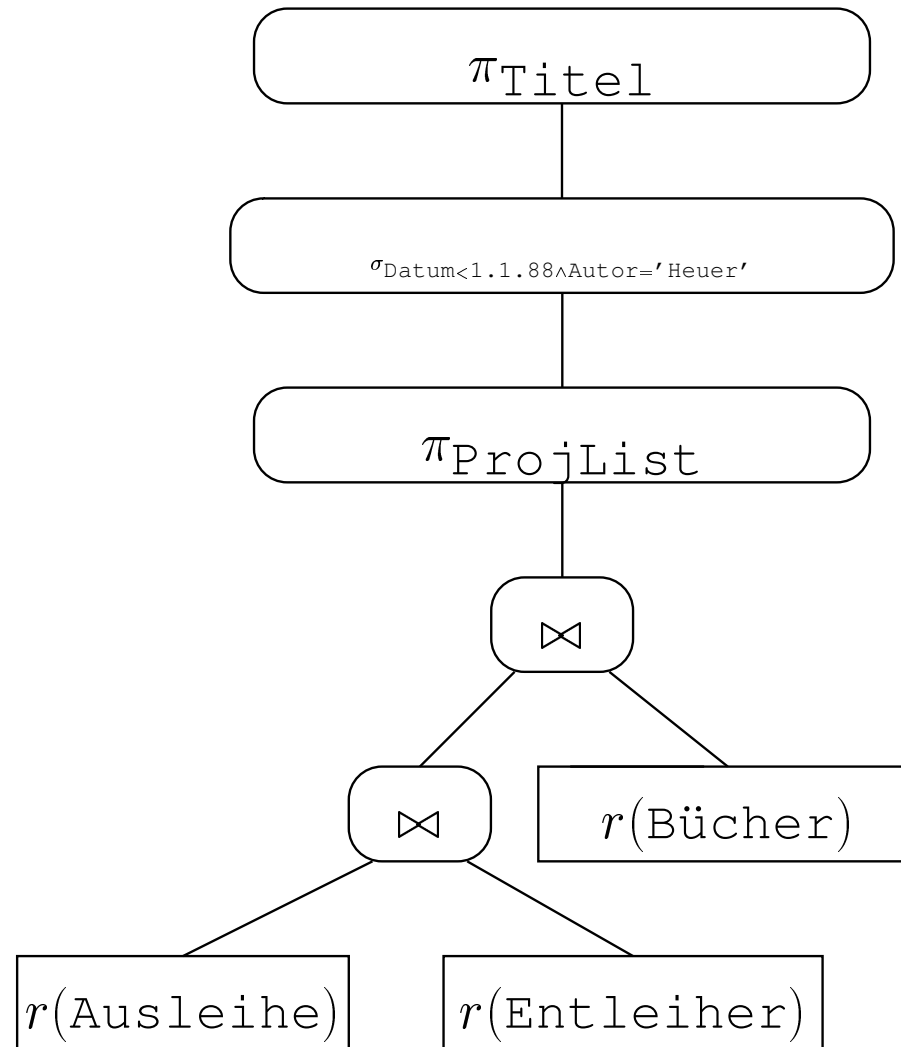
---

- komplexe Selektionsprädikate auflösen (Regel **SelSel**, ggf. Regeln der Auflösung für  $\neg$  und  $\vee$ )
- mittels der Regeln **SelJoin**, **SelProj**, **SelUnion** und **SelDiff** Selektionen möglichst weit in Richtung der Blätter verschieben, ggf. Selektionen gemäß Regel **SelSel** vertauschen
- Verschieben der Projektionen in Richtung Blätter mittels der Regeln **ProjProj**, **ProjJoin** und **ProjUnion**

Einzelschritte werden in der genannten Reihenfolge solange ausgeführt, bis keine Ersetzungen mehr möglich sind

# Unoptimierter Anfrageplan

---

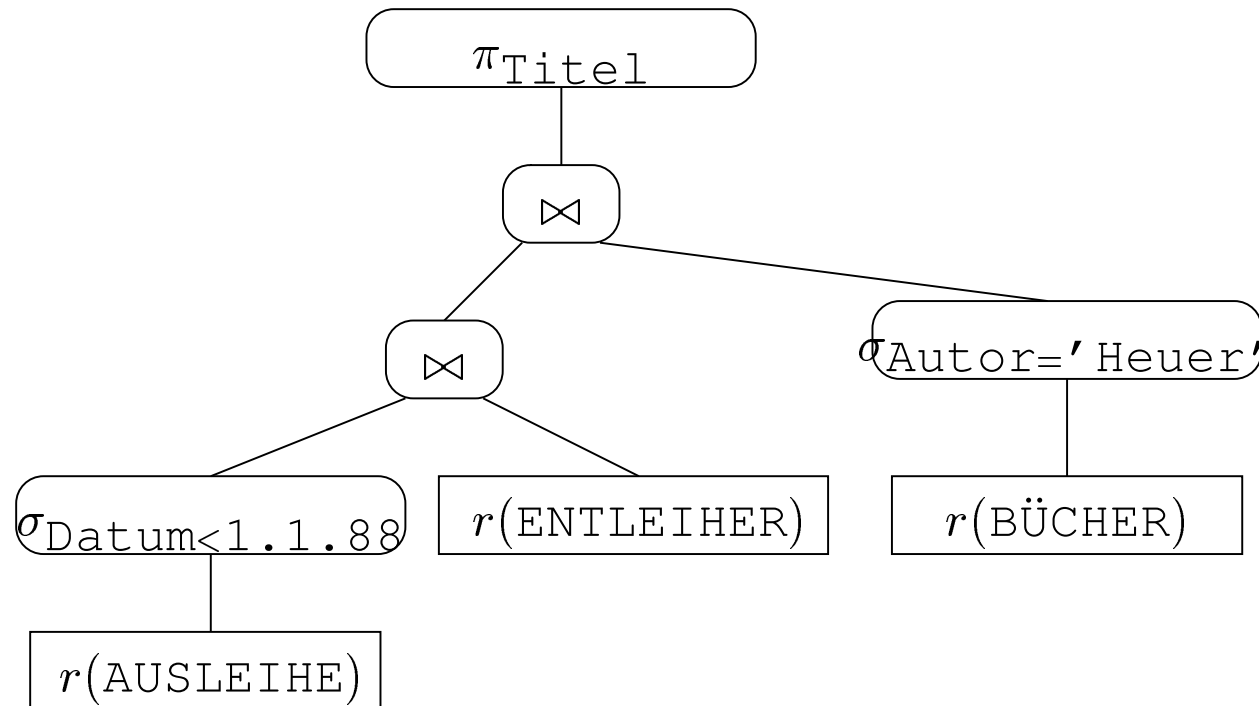


ProjList=Titel,Autor,Verlag,ISBN,EntlName,EntlAdr...

# Anfrageplan (II)

---

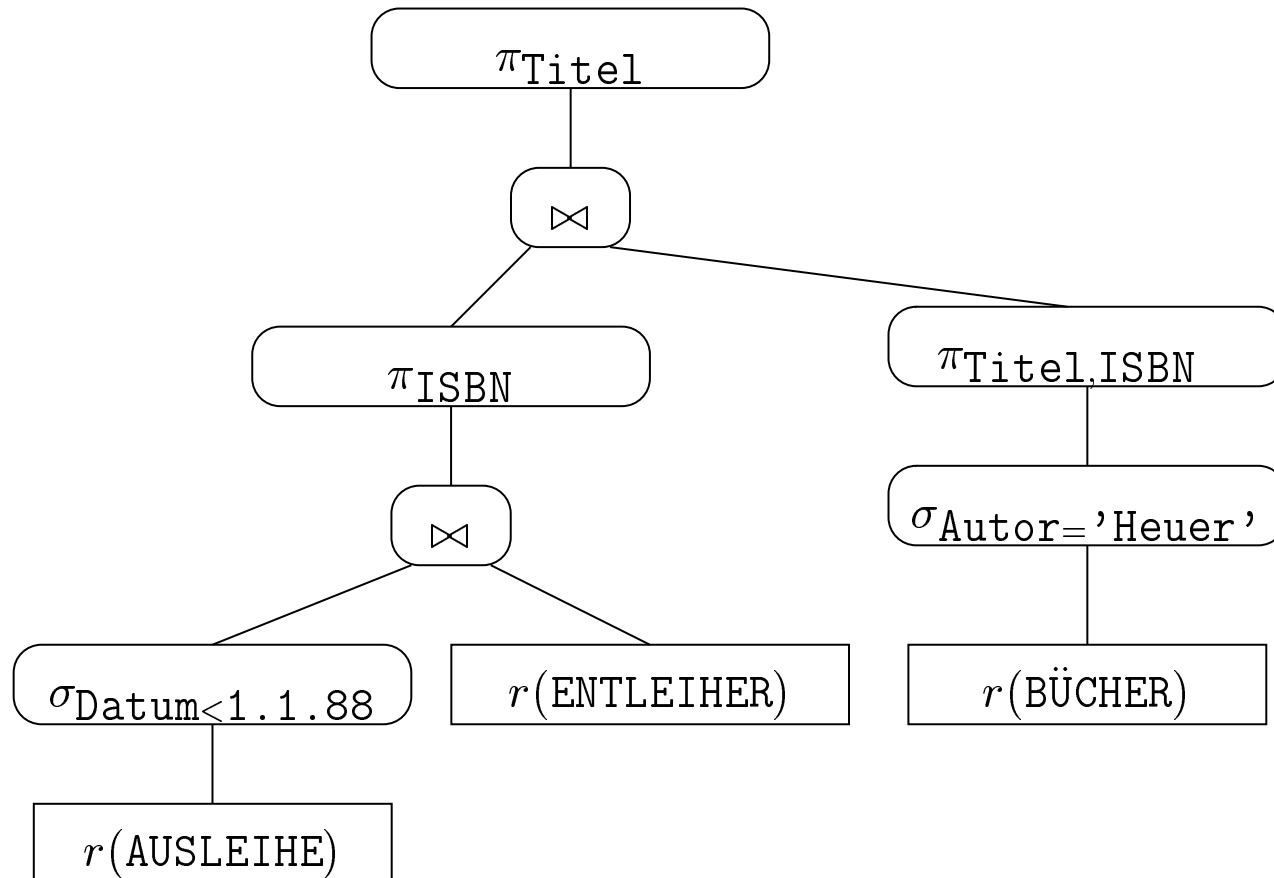
nach Verschieben der Selektionen



# Anfrageplan (III)

---

mit zusätzlichen Projektionen



# Verbundoptimierung mit Tableaus

---

- exakte Optimierung (minimale Anzahl von Verbunden)
- für eine eingeschränkte Klasse von Anfragen ( $\sigma, \pi, \bowtie$ )

sinnvoll bei

- Optimierung von Anfragen über Sichten
- Anfrageinterpretation von Universalrelationenschnittstellen

# Tableaus — Informale Einführung

---

matrixförmige Darstellung einer Relationenalgebra- oder Relationenkalkül-Anfrage

- Spalten der Matrix: Attribute des Universums
- erste Zeile des Tableaus: *summary*; *blanks* und *ausgezeichnete Variablen*  $a_i$
- weitere Zeilen: *blanks*, ausgezeichnete Variablen, Konstanten und *nichtausgezeichnete Variablen*  $b_j$ ; besitzt *tag* (Namen einer Basisrelation)



# Äquivalente Klassen von Anfragen

---

- *Tableau-Anfragen*
- *Konjunktive Anfragen* Teilmenge des Bereichskalküls

$$\{a_1 \dots a_n \mid \exists b_1 \dots \exists b_m : F_1 \wedge \dots \wedge F_k\}$$

wobei  $F_i = R(c_1 \dots c_r)$ , d.h.  $c_1 \dots c_r \in r(R)$ ;  $c_j$   
ausgezeichnete oder nichtausgezeichnete Variablen  
oder Konstanten

- *Eingeschränkte relationenalgebraische Ausdrücke.*

$\sigma_{A=c}, \sigma_{A=B}, \sigma_F, \pi, \bowtie$  mit  $F$  bestehend aus  
 $A = c, A = B, \wedge$ .

# Beispiel für Tableau-Anfragen

$\mathcal{U} = \{A, B, C\}$  mit  $R_1 = \{A, B\}$  und  $R_2 = \{B, C\}$

- Algebra-Ausdruck:  $\pi_A(\sigma_{C=3}(r(R_1) \bowtie r(R_2)))$
- Konjunktive Anfrage:  $\{a_1 \mid \exists b_1 : R_1(a_1, b_1) \wedge R_2(b_1, 3)\}$
- Tableau-Anfrage:

					tags
					↓
			A	B	C
summary	→	$w_0$	$a_1$		
rows	→	$w_1$	$a_1$	$b_1$	$R_1$
		$w_2$		$b_1$	$R_2$
				<b>3</b>	

# Interne Optimierung

---

- erster Schritt: Relationenalgebraoperationen in interne Operationen aus Kapitel 6
- weitere Operationen
  - ◆ Tupelmengen zum Teil durch sortierte Tupellisten ersetzen
  - ◆ statt Mengen Multimengen
  - ◆ neben Tupeln auch TID-Listen verarbeiten
  - ◆ Zugriffe auf Indexe

# Auswahl von Berechnungsalgorithmen

---

## ■ Projektion:

- ◆  $\pi_{AttList}^{\mathbf{REL/mit}}$ : **REL** Projektion durch Relationen-Scan (**mit** bezeichnet eine Projektion mit Duplikateliminierung)
- ◆  $\pi_{AttList}^{\mathbf{REL/ohne}}$ : Projektion durch Relationen-Scan ohne Duplikateliminierung
- ◆  $\pi_{AttList}^{\mathbf{SORT/mit}}$ : **SORT** Projektion durch Scan über einer nach `AttList` sortierten Relation mit Duplikateliminierung
- ◆  $\pi_{AttList}^{\mathbf{SORT/ohne}}$ : Projektion durch Scan über einer nach `AttList` sortierten Relation ohne Duplikateliminierung

# Auswahl von Berechnungsalgorithmen II

---

- Selektion:
  - ◆  $\mathbf{REL}_{\sigma_{\varphi}^{\mathbf{REL}}}$ : Selektion durch Relationen-Scan
  - ◆ Selektion über Index (später detailliert)
- Verbund:
  - ◆  $\bowtie^{\mathbf{DIRECT}}$ : Verbund durch Nested-Loops **DIRECT**
  - ◆  $\bowtie^{\mathbf{MERGE}}$ : Verbund durch Mischen **MERGE**  
(Voraussetzung: Eingaberelationen nach Verbundattribut(en) sortiert).
  - ◆  $\bowtie^{\mathbf{HASH}}$ : Verbund durch Hash-Join **HASH**
- Operatoren für Vereinigung, Differenz und Durchschnitt analog zum Verbund; Vereinigung mit oder ohne Duplikateliminierung

# Indexzugriff

---

- Zugriff über Index

$$\sigma_{A\Theta a}^{\text{IND}}(\text{I}(\text{R}(A))) \rightarrow \mathbf{list(tid)}$$

liefert TID-Liste

- Spezialfall  $\sigma_{\text{true}}^{\text{IND}}(\text{I}(\text{R}(A)))$ : *alle* Indexeinträge sortiert nach  $A$
- Varianten:
  - ◆ duplikatfreier Primärindex: Ergebnis bei  $A = a$  ein einzelnes Tupel
  - ◆ Tupel direkt im Index: Ergebnis des Indexzugriffs ist sortierte Liste von Tupeln
  - ◆ *Bereichsanfrage*: Prädikat  $a_1 \leq A \leq a_2$

# Indexzugriff II

---

- Index kann auch Projektion unterstützen
- Ergebnis von  $\sigma_{\text{true}}^{\text{IND}}(\text{I}(\text{R}(\text{A})))$  als Eingabe für  $\pi^{\text{SORT/-}}$  Operator nehmen
- kombinierter Zugriff:  $\pi_{\text{AttList}}^{\text{IND/mit}}$  bzw.  $\pi_{\text{AttList}}^{\text{IND/ohne}}$
- in  $\pi_A^{\text{IND/-}}(\text{I}(\text{R}(\text{A})))$  kann auf Zugriff auf Basisrelation ganz verzichtet werden

# Neue Operatoren

---

- Für TID-Listen: ‘*Realisierung*’-Operator  $\rho$ :

$$\rho(\langle \text{TID-Liste für } R\text{-Tupel} \rangle, r(R))$$

- Auf TID-Listen  $\cup$ ,  $\cap$  und  $-$
- *Sortierung* von Tupelmengen:  $\omega$

$$\omega_{\text{AttList}}(\langle \text{Tupel-Folge} \rangle)$$

- Relation sortieren:

$$\rho(\sigma_{\text{true}}^{\text{IND}}(\text{I}(\text{R}(\text{AttList}))), r(R)) = \omega_{\text{AttList}}(r(R))$$

erste Variante Aufwand  $O(|R|)$ , zweite Variante  $O(|R| \times \log |R|)$

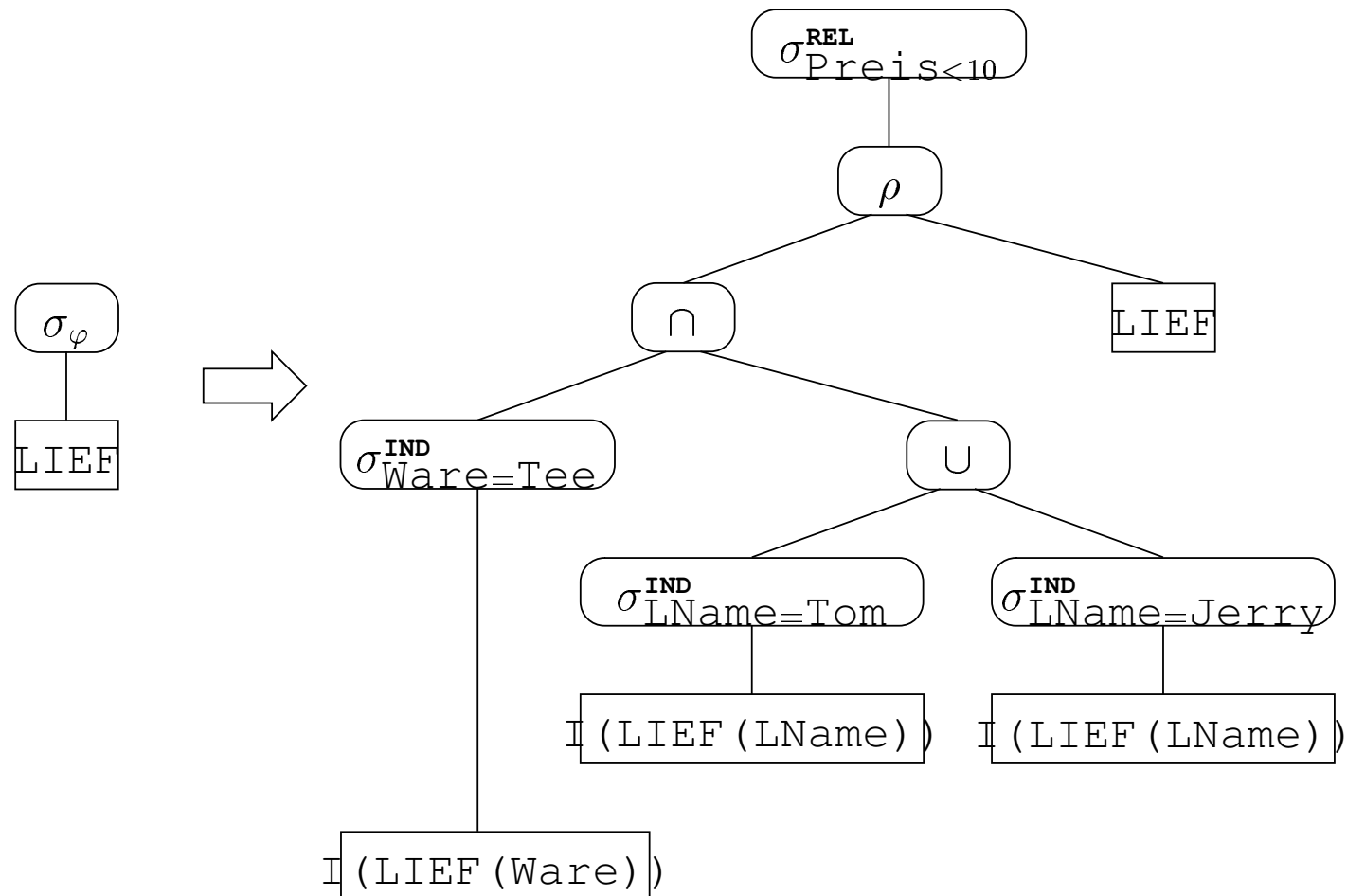


# Beispiele für interne Zugriffspläne

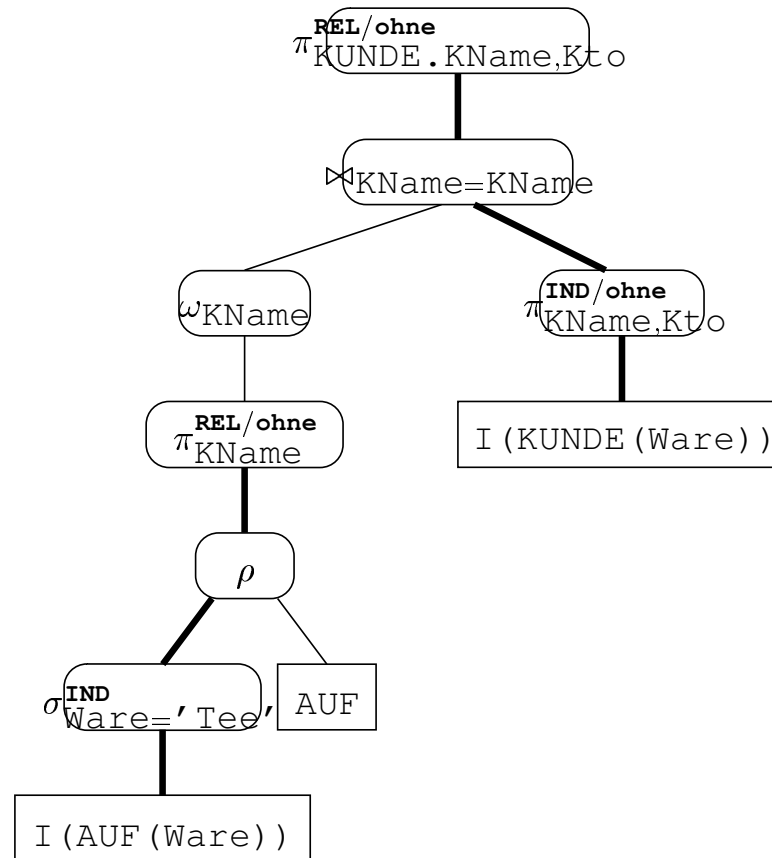
---

```
select *  
from LIEFERANT  
where Ware = 'Tee' and  
      ( LName = 'Tom' or LName = 'Jerry' )  
and Preis < 10
```

# Zwei Zugriffspläne



# Pipelining von Operationen



$$\pi_A(\sigma_\varphi(r(R))) \longleftrightarrow \langle \pi_A \circ \sigma_\varphi \rangle(r(R))$$

# Pipelining von Operationen II

---

Typische Verschmelzungen:

- Kombination von Selektion und Projektion
- Kombination einer Selektion mit Verbund
- die Integration einer Selektion in die äußere Schleife eines Nested-Loops-Verbund
- Integration von Selektionen in den Merge-Join
- Kopplung der Selektion mit der Realisierung

```
select K.KName, Kto  
from KUNDE K, AUFTRAG A  
where K.KName = A.KName and A.Ware = 'Tee'
```

# Gemeinsame Teilanfragen

---

- Aufgabe: Erkennung *gemeinsamer Teilanfragen*  $\leadsto$  Gleichsetzen der zugehörigen Teilbäume des Operatorbaums
- Probleme:
  - ◆ unterschiedliche syntaktische Form:  $r_1 \cup r_2$  identisch zu  $r_2 \cup r_1$
  - ◆ *Überdeckung* ( $\sigma_\varphi$  überdeckt n  $\sigma_{\varphi \wedge \psi}$ )

# Kostenbasierte Auswahl

---

Berücksichtigung:

- tatsächliche Größe der Datenbankrelationen
- Existenz von Indexen (Primär-, Sekundär-) und ihre Größe
- Clustering mehrerer Relationen
- Selektivität eines Attributs, über das ein Index aufgebaut wurde

# Relevante Datenbankparameter

---

- *Systemparameter* aus Katalog:  $s$ : Länge einer Seite (nutzbarer Seitenbereich in Byte)
- Größe der Datenbank: Anzahl  $S$  der belegten Seiten (wird benötigt, wenn Tupel von Relationen gestreut gespeichert werden)
- statistische Daten über Relationen und Indexe:
  - ◆  $T_R$ : Anzahl der Tupel in Relation  $R$
  - ◆  $L_R$ : durchschn. Länge eines Tupels in  $R$
  - ◆  $W_{A,R}$ : Anzahl der verschiedenen Werte des Attributes  $A$  in  $R$  (Indexinformation oder Statistik)

Statistiken → Aktualisierung bei Änderungsoperation oder Aufruf entsprechender Kommandos

# Selektivität von Attributen

---

Anzahl der verschiedenen Werte des Attributs  $A$  in  $R$ :  $W_{A,R}$

- Gleichheit:

$$sel(A=c, R) = \frac{1}{W_{A,R}}$$

- Ungleichheit:

$$sel(\mathbf{not} A=c, R) = 1 - sel(A=c, R) = 1 - \frac{1}{W_{A,R}}$$

- Vergleich mittels  $<, >, \dots$ :

$$sel(A < c, R) = sel(A > c) = \frac{1}{2}$$



# Selektivität von Attributen II

---

Verfeinerung:

$$sel(A \leq c, R) = \frac{A_{max} - c}{A_{max} - A_{min}}$$

Bereichsanfragen:

$$sel(c_u \leq A \leq c_o, R) = \frac{c_o - c_u}{A_{max} - A_{min}}$$

Selektivitäten für Verbunde:

$$sel_{\bowtie}(\varphi, R, S) \approx \frac{|R \bowtie_{\varphi} S|}{|R \times S|}$$

# Selektivitätsabschätzung

---

## 1. *Parametrisierte Funktionen*

Parametrisierung einer Funktion, die die Datenverteilung gut widerspiegelt, möglichst genau angeben (etwa Normalverteilung)

## 2. *Histogramme*

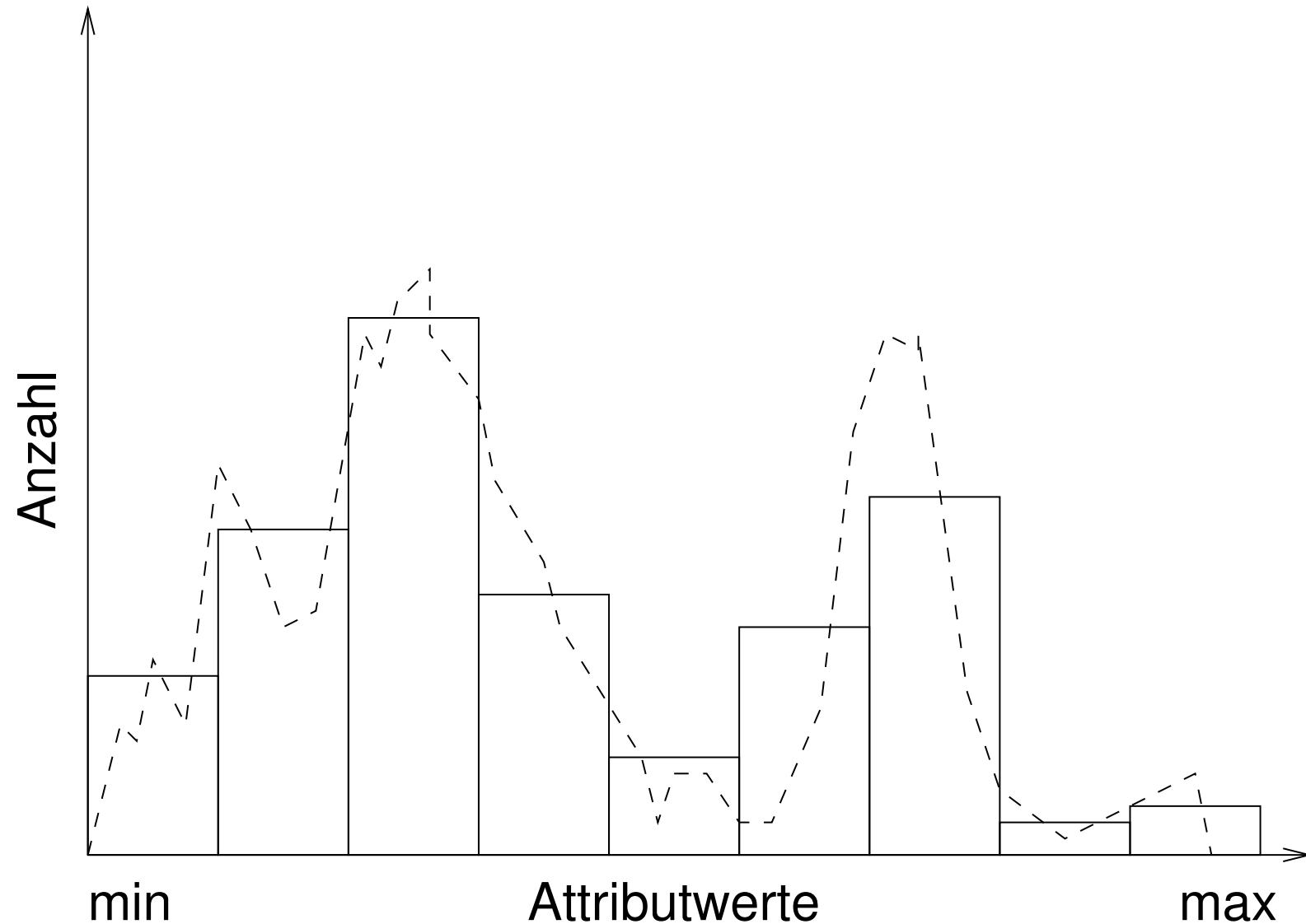
Wertebereich in Unterbereiche aufteilen und die tatsächlich in diese Unterbereiche fallenden Werte zählen

## 3. *Stichproben*

Selektivität anhand einer zufälligen Stichprobe der gespeicherten Datensätze bestimmen

# Histogramme für Attributwerte

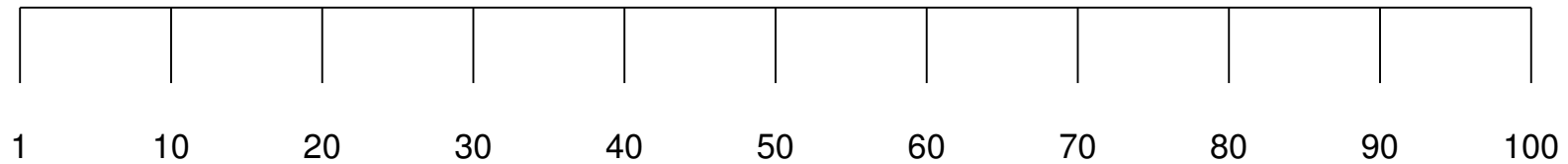
---



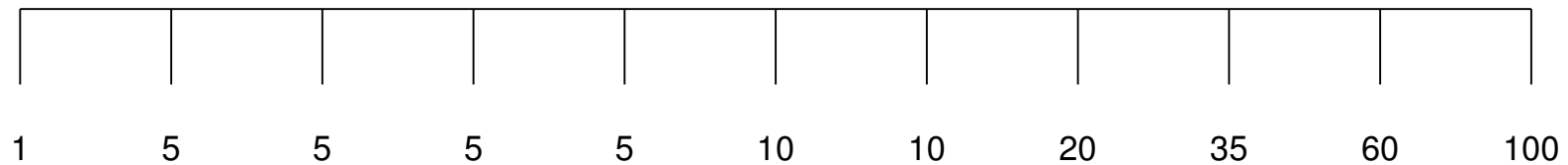
# Höhenbalancierte Histogramme

---

a) gleichverteilte Werte



b) ungleichverteilte Werte



# Kostenberechnung am Beispiel

---

Selektionen über Relation `AUFTRAG` mit Attribut `Ware`

- $S = 4.000$  (Die Datenbank belegt insgesamt 4000 Seiten)
- $T_{\text{AUFTRAG}} = 10.000$  (in der Relation `AUFTRAG` sind insgesamt 10.000 Tupel gespeichert)
- $s/L_{\text{AUFTRAG}} = 10$  (auf eine Seite passen durchschnittlich 10 Auftrags-Tupel)
- $W_{\text{Ware}, \text{AUFTRAG}} = 50$ . (50 verschiedene Waren als Wert des `Ware`-Attributs)

Selektion  $\sigma_{\varphi}$  mit  $\varphi = (A\theta a \wedge \psi)$

# Kostenberechnung: Variante A

---

- Index  $I(R(A))$  über Selektionsattribut  $A$

$$\langle \sigma_{\psi}^{\mathbf{REL}} \circ \rho \circ \sigma_{A\theta a}^{\mathbf{IND}} \rangle (I(R(A)), r(R))$$

- A1 Index mit Cluster-Bildung:  $S_R \times sel(A\theta a, R)$ .
- A2 Index ohne Cluster-Bildung:  $T_R \times sel(A\theta a, R)$ .  
(Maximalwert, wenn die Tupel jeweils auf verschiedenen Seiten liegen)

# Kostenberechnung: Variante B

---

- Index  $I(R(B))$  mit  $B \neq A$

$$\langle \sigma_{A\theta a \wedge \psi}^{\mathbf{REL}} \circ \rho \circ \sigma_{\mathbf{true}}^{\mathbf{IND}} \rangle (I(R(B)), r(R))$$

B1 Index mit Cluster-Bildung:  $S_R$ .

B2 Index ohne Cluster-Bildung:  $T_R$ .

Kosten ergeben sich, da die Selektivität des Prädikates **true** 1 ist

# Kostenberechnung: Variante C

---

- Relationen-Scan
- Annahme: alle Seiten der Datenbank werden gelesen und alle auf den Seiten vorhandenen Tupel werden gefunden
- Kosten: durch die Anzahl  $S$  gegeben



# Kostenberechnung: Anfragen

---

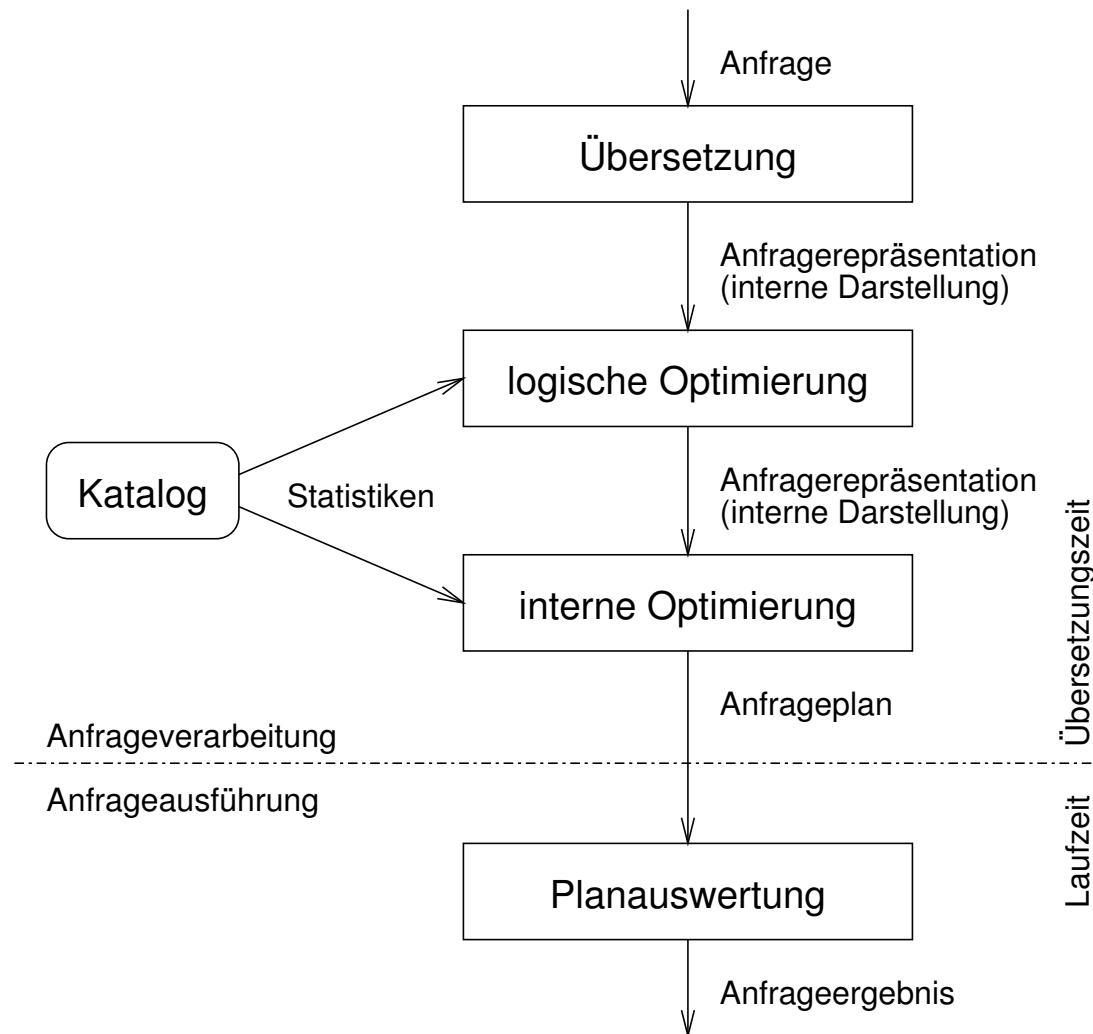
1.  $\sigma_{\text{Ware} = \text{'Tee'}}(r(\text{AUFTRAG}))$ .  
Selektivität  $sel: \frac{1}{50}$
2.  $\sigma_{\text{Ware} > \text{'Tee'}}(r(\text{AUFTRAG}))$ .  
Selektivität  $sel: \text{Annahme} \leadsto \frac{1}{2}$ .

# Kosten für Ausführungsvarianten

---

Variante	Kostenformel	Ware = 'Tee'	Ware > 'Tee'
A1	$S_R \times sel(A\theta_a, R)$	20	500
A2	$T_R \times sel(A\theta_a, R)$	200	5.000
B1	$S_R$	1.000	1.000
B2	$T_R$	10.000	10.000
C	$S$	4.000	4.000

# Optimierer-Architektur



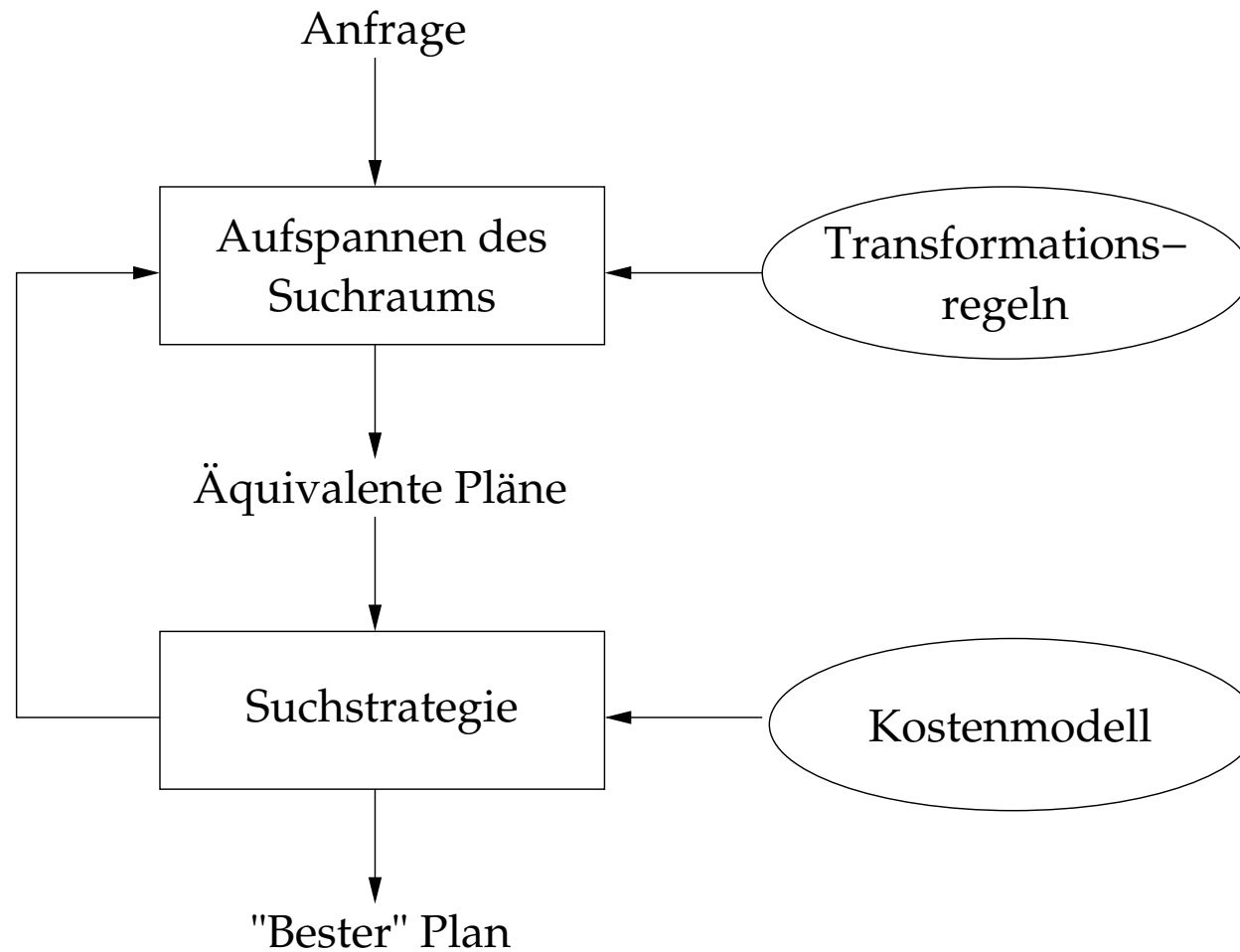
# Optimierer-Varianten

---

- *heuristische, regelbasierte Optimierer:*
  1. Erzeugung einer interner Anfragerepräsentation durch logische Optimierung
  2. mit interne Optimierung einen Anfrageplan erzeugen
- *kostenbasierte (Zwei-Phasen-) Optimierer:*
  1. Erzeugung verschiedener Anfragerepräsentation durch logische Optimierung
  2. Übergabe an interne Optimierung (Plangenerierung)
  3. Kostenbewertung
  4. Auswahl des besten Plans

# Optimierung: Überblick

---



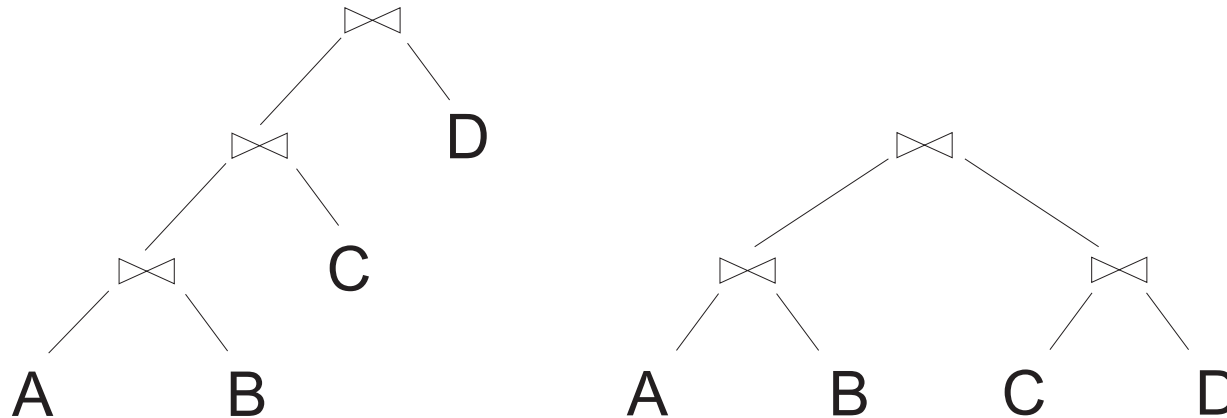
# Optimierung: Suchraum

---

- Suchraum: Menge aller äquivalenten Anfragepläne
- Aufspannen durch *Transformationsregeln* (algebraische Regeln)
- Schwerpunkt: *Join-Trees*
- für  $n$  Relationen:  $n!$  verschiedene Join-Trees!
- daher: Beschränkung des Suchraums durch
  - ◆ Heuristiken (algebraische Optimierungen)
  - ◆ vorgegebene „Form“ des Baumes

# Optimierung: Join-Trees

---



- lineare Folgen von Operatorbäumen
  - ◆ nur  $2^n$  Varianten
  - ◆ *left deep tree* und *right deep tree*  $\leadsto$  alle inneren Knoten des Baums besitzen mindestens einen Blattknoten (Basisrelation) als Kind
- *bushy trees*
  - ◆ höheres Parallelisierungspotential, jedoch großer Optimierungsaufwand

# Optimierung: Suchstrategien

---

- „Durchlaufen“ des Suchraums
- Auswahl des kostengünstigsten Plans
- Basis: Kostenmodell
- bestimmt
  - ◆ *welche Pläne* werden betrachtet (vollständiges / partielles Durchsuchen)
  - ◆ in welcher *Reihenfolge* werden Alternativen untersucht
- Varianten: deterministisch, zufallsbasiert



# Optimierung: Suchstrategien /2

---

- deterministisch:
  - ◆ systematisches Generieren von Plänen
  - ◆ beginnend bei Plänen für Zugriff auf Basisrelationen
  - ◆ Konstruktion komplexer Pläne durch Verbund einfacherer Pläne
  - ◆ erschöpfende Suche; garantiert besten Plan
  - ◆ Beispiel: *Dynamic Programming* (Breitensuche)
  - ◆ State of the art

# Optimierung: Suchstrategien /3

---

- zufallsbasiert:
  - ◆ ein oder mehrere Startpläne durch Greedy-Strategie (Tiefensuche)
  - ◆ Verbesserung der Startpläne durch Untersuchung von „Nachbarn“
  - ◆ Nachbar: Anwendung von Transformationsregeln, z.B. Vertauschen zweier zufällig gewählter Operationen
  - ◆ Beispiel: *Simulated Annealing*
  - ◆ bessere Performance bei großer Anzahl von Relationen
  - ◆ keine Garantie für besten Plan

# Optimierung in INGRES

---

- dynamisches Verfahren zur Optimierung
- rekursives Zerlegen einer Kalkülanfrage in Sequenzen von *Mono-Relationen-Anfragen* (Ein-Tupelvariablen-Anfragen)
- Verarbeitung der Mono-Relationen-Anfragen durch „One Variable Query Processor“ (OVQP)
- OVQP wählt beste Zugriffsmethode (Indexauswahl)

# Optimierung in INGRES /2

---

## ■ Prinzip:

### ◆ Anfrage $q$ :

```
select  $R_2.A_2, R_3.A_3, \dots, R_n.A_n$   
from  $R_1, R_2, \dots, R_n$   
where  $P_1(R_1.A'_1)$  and  $P_2(R_1.A_1, R_2.A_2, \dots, R_n.A_n)$ 
```

### ◆ Zerlegung in $q'$ :

```
select  $R_1.A_1$  into  $R'_1$   
from  $R_1$   
where  $P_1(R_1.A'_1)$ 
```

### ◆ und $q''$ :

```
select  $R_2.A_2, R_3.A_3, \dots, R_n.A_n$   
from  $R'_1, R_2, \dots, R_n$   
where  $P_2(R_1.A_1, R_2.A_2, \dots, R_n.A_n)$ 
```

# Optimierung in INGRES /3

---

- Behandlung von nicht reduzierbaren Multi-Relationen-Anfragen (insb. Verbunde)
  - ◆ Konvertierung in Mono-Relationen-Anfragen durch *Tupelsubstitution*
  - ◆ für Anfrage  $q$ : Auswahl einer Relation  $R_1$  und Ableitung von  $\text{card}(R_1)$  Anfragen  $q'$  mit  $n - 1$  Relationen

$q(R_1, R_2, \dots, R_n)$  ersetzen durch

$$\{q'(t_{1i}, R_2, R_3, \dots, R_n); t_{1i} \in R_1\}$$

# Optimierung in System R

---

- Dynamic Programming
- Bottom-Up-Konstruktion eines Plans
  1. Generierung einfacher Pläne (Zugriff auf eine Relation)
  2. Generierung komplexerer Pläne (2 Relationen, 3 ... ) durch Kombination (Verbund) einfacher Pläne
- dabei: *Pruning*
  - ◆ Begrenzung des Lösungsraums durch Löschen von „schlechten“ Plänen für die äquivalente Pläne existieren
    1. Permutationen mit kartesischen Produkten
    2. kommutative Strategien mit höchsten Kosten

# Optimierung in System R /2

---

**Input:** SPJ-Anfrage  $q$  auf den Relationen  $r_1, \dots, r_n$

**Output:** Anfrageplan für  $q$

**for**  $i := 1$  **to**  $n$  **do**

$optPlan(\{r_i\}) := accessPlans(r_i)$

$prunePlans(optPlan(\{r_i\}))$

**end**

**for**  $i := 1$  **to**  $n$  **do**

**forall**  $s \subseteq \{r_1, \dots, r_n\}$  **such that**  $|s| = i$  **do**

$optPlan(s) := \emptyset$

**forall**  $t \subset s$  **do**

$optPlan(s) := optPlan(s) \cup$

$joinPlans(optPlan(t), optPlan(s - t))$

$prunePlans(optPlan(s))$

**end end end**

**return**  $optPlan(\{r_1, \dots, r_n\})$

# Optimierung in System R /3

---

- Beispielanfrage:

```
select M.MName  
from Mitarbeiter M, Zuordnung Z, Projekt P  
where M.MNr = Z.MNo and Z.PNr = P.PNr  
and P.PName = 'DB-Entwicklung'
```

- **Indexe:** Mitarbeiter (MNr), Zuordnung (PNr),  
Projekt (PNr), Projekt (PName)

- Zugriffspläne nach 1. Iteration:

```
Mitarbeiter: Full-Table-Scan  
Zuordnung:   Full-Table-Scan  
Projekt:      Index-Scan auf PName
```



# Optimierung in System R /4

## ■ mögliche Verbundreihenfolgen



# Oracle9i: Statistiken

---

- Tabellen: Anzahl Tupel und Blöcke, durchschnittliche Tupellänge
- Spalten: Anzahl der verschiedenen Werte, Anzahl der **NULL**-Werte, Datenverteilung (Histogramme)
- Indexe: Anzahl der Blattseiten, Höhe des Baums, Clustering-Faktor
- System: I/O- bzw. CPU-Performance und -Auslastung

# Gewinnung von Statistiken

---

- Gewinnung durch
  - ◆ Abschätzung auf Basis von Stichproben (*row sampling, block sampling*)
  - ◆ exakte Berechnung (Aufwand: Table-Scan + Sortierung)
  - ◆ benutzerdefiniert
- Werkzeuge
  - ◆ **ANALYZE TABLE ...**
  - ◆ Package `DBMS_STATS`: Prozeduren für erweiterte Behandlung von Statistiken (von Oracle empfohlen)

# Pflege von Statistiken

---

- Aktualisierung der Statistiken von Hand (evtl. als Job)
- automatische Aktualisierung (Monitoring)
  - ◆ Beobachtung der Anzahl der Änderungsoperationen
  - ◆  $\geq 10\%$  betroffen  $\leadsto$  veraltete Daten  $\leadsto$  Aktualisierung

# Fehlende Statistiken

---

## ■ Default-Werte für fehlende Statistiken

### ◆ Tabellen

- durchschnittl. Tupelgröße: 100 Bytes
- Anzahl Blöcke: 1
- Kardinalität:  $num\_of\_blocks * (block\_size - cache\_layer) / avg\_row\_len$

### ◆ Indexe

- Höhe: 1
- Anzahl Blattseiten: 25
- Anz. unterschiedl. Schlüsselwerte: 100

# Erzeugen von Statistiken: ANALYZE

---

## ■ Aufruf:

**analyze table** *tabelle statistik-art*

## ■ Formen

- ◆ **estimate statistics**: Abschätzung durch Stichprobe; optionaler Parameter spezifiziert Stichprobengröße (**sample** *groesse*  
**rows** | **percent**)
- ◆ **compute statistics**: exakte Bestimmung

## ■ Beispiel:

```
analyze table emp  
  estimate statistics sample 10 percent;
```

# Erzeugen von Statistiken: DBMS\_STATS

---

- Aufruf der Package-Prozeduren:

- ◆ u.a. `gather_index_stats`,  
`gather_table_stats`, `gather_schema_stats`,  
...

- Beispiel:

```
execute dbms_stats.gather_table_stats(  
    ownname => 'scott',  
    tabname => 'emp',  
    estimate_percent => NULL,  
    method_opt => NULL);
```

# DBMS\_STATS (II)

---

## ■ Parameter:

- ◆ `estimate_percent`: Abschätzung auf Basis der gegebenen Stichprobengröße (in Prozent)
- ◆ `method_opt`: Spezifikation der zu erzeugenden Statistiken, u.a.
  - `FOR ALL [INDEXED ] COLUMNS`: alle Spalten
  - `FOR COLUMNS Spaltenliste`: für gegebene Spalten
  - `AUTO`: automatische Auswahl der Spalten für Histogramme



# Anzeige von Statistiken

---

- Statistikdaten im Data Dictionary
- Views: dba\_-, user\_-, all\_tables, -tab\_col\_statistics
- Beispiel: Tabellenstatistik (user\_tables)

TABLE_NAME	NUM_ROWS	BLOCKS	AVG_ROW_LEN
-----	-----	-----	-----
EMP	14	1	37

# Anzeige von Statistiken (II)

---

## ■ Beispiel: Spaltenstatistik (user\_tab\_col\_statistics)

COLUMN_NAME	NUM_DISTINCT	NUM_NULLS	NUM_BUCKETS
-----	-----	-----	-----
EMPNO	14	0	13
ENAME	14	0	13
JOB	5	0	4
MGR	6	1	5
HIREDATE	13	0	12
SAL	12	0	11
COMM	4	10	3
DEPTNO	3	0	2

# Oracle: Histogramme

---

## ■ Arten

- ◆ höhenbasiert (equi-height)
- ◆ wertbasiert (frequency)
  - jeder Wert der Spalte hat korrespondierendes Bucket
  - Bucketnummer entspricht Häufigkeit des Wertes
  - Anwendung, wenn Anzahl der verschiedenen Werte der Spalte  $\leq$  Anzahl der Buckets

## ■ Auswahl in Abhängigkeit von Häufigkeit der Werte

# Histogramme: Erzeugung

---

## ■ ANALYZE TABLE

```
analyze table emp compute statistics  
for column sal size 10;
```

## ■ DBMS\_STATS

```
execute dbms_stats.gather_table_stats(  
    ownname => 'scott',  
    tabname => 'emp',  
    method_opt => 'for columns size 10 sal');
```

## ■ Default-Anzahl der Buckets: 75

# Ausgabe von Histogrammen

---

- Data-Dictionary-Views: dba\_-, user\_-, all\_histograms

- Anfrage:

```
select endpoint_number, endpoint_value
from user_histograms
where table_name='EMP' and column_name='SAL' ;
```

ENDPOINT_NUMBER	ENDPOINT_VALUE
1	800
2	950
3	1100
...	
13	3000
14	5000

# Oracle: Ausgabe von Plänen

---

- Speichern eines Ausführungsplans (inkl. Kosten) zu einer Anfrage in einer Tabelle `plan_table`

```
explain plan set statement_id = 'MPLAN'  
for select title from movie, budget  
where id between 2000 and 40000  
       and movie.id = budget.movie  
       and budget < 100000 and year = 1998;
```

# Oracle: Ausgabe von Plänen (II)

---

- Auslesen des Plans durch Anfrage oder spezielle Tools

```
select substr(lpad(' ', 2*(level-1)), 1, 8) ||  
       substr(operation, 1, 16) "OPERATION",  
       substr(options, 1, 12) "OPTIONS",  
       substr(object_name, 1, 12) object_name,  
       id, parent_id, cost, cardinality, bytes  
from plan_table  
start with id=0 and statement_id = 'MPLAN'  
connect by prior id = parent_id and  
               statement_id = 'MPLAN';
```

# Oracle: Ausgabe von Plänen (III)

---

## ■ Ausgabe

OPERATION	OPTIONS	OBJECT_NAME	ID	PARENT_ID	
-----	-----	-----	----	-----	...
SELECT STATEMENT			0		
NESTED LOOPS			1	0	
TABLE ACCESS	BY INDEX ROW	MY_MOVIE	2	1	
INDEX	RANGE SCAN	MOVIE_PK	3	2	
TABLE ACCESS	BY INDEX ROW	BUDGET	4	1	
INDEX	UNIQUE SCAN	SYS_C006658	5	4	



# Oracle: Spalten der Plan-Tabelle

---

Spalte	Bedeutung
statement_id	ID aus <b>explain plan</b>
operations	Planoperator (1. Zeile: Statement)
options	Details zum Planoperator
object_name	Tabelle, Index etc.
id, parent_id	IDs der Operationen
position	(1. Zeile: Gesamtkosten, sonst: relative Pos.)
cost	Kosten der Operation (Funktion über CPU/IO-Kosten)
cardinality	Anzahl der verarbeiteten Tupel
bytes	Größe der verarbeiteten Bytes
cpu_cost, io_cost	CPU/IO-Kosten (proportional zu tats. Werten)
temp_space	benötigter temp. Speicher in Bytes

# Oracle: Operationen in der Plan-Tabelle

---

operation	option	Bedeutung
filter		Selektion bzgl. Bedingung
hash join		Hash-Join
merge join		Merge-Join
merge join	(outer, cartesian)	
nested loops		Nested-Loops-Join
index	unique scan	Zugriff auf einzelne Rowid
index	range scan	Bereichszugriff auf Index
sort	group by	Sortierung für Gruppierung
sort	unique	... für Duplikateliminierung
sort	join	... für Merge-Join
table access	full	table scan
table access	by index rowid	Tabellenzugriff über Rowid von Indexzugriff

# Oracle: Optimizer Hints

---

- gezielte Beeinflussung der Optimierung von Anfragen
- Aspekte:
  - ◆ Ziel: Durchsatz / Antwortzeit
  - ◆ Anfragetransformation (z.B. Star-Queries, materialisierte Sichten etc.)
  - ◆ Zugriffspfade
  - ◆ Verbundreihenfolge
  - ◆ Verbundoperation
- Angabe: durch Kommentare
  - /\* +hint \*/*
  - +hint*

# Hints: Durchsatz vs. Antwortzeit

---

- Optimierungsziel: größter Durchsatz (geringste Ressourcennutzung)

```
select /*+ all rows */ *  
from emp;
```

- Optimierungsziel: beste Antwortzeit, d.h.  $n$  erste Tupel möglichst schnell liefern  
(nicht für **group by**, **order by**, Mengenoperationen und **distinct**-Anfragen)

```
select /*+ first rows(10) */ *  
from emp;
```

# Hints: Zugriffspfade

---

- Auswahl verschiedener Strategien: `full`, `index`, ...
- Beispiel: Nutzung eines Index für `ename`

```
select /*+ index(emp ename_idx) */ *  
from emp  
where ename = 'JONES';
```

# Hints: Verbunde

---

## ■ Angabe der Join-Implementierung

◆ `/* +use_nl(inner_tbl) */:`  
Nested-Loops-Join

◆ `/* +use_merge(tbl1 tbl2) */:` Merge-Join

◆ `/* +use_hash(tbl1 tbl2) */:` Hash-Join

## ■ Angabe der Verbundreihenfolge

◆ `/* +ordered */:` Reihenfolge wie in  
**from**-Klausel

## ■ Beispiel:

```
select /*+ ordered */ *  
from tab1, tab2, tab3  
where tab1.col = tab2.col  
and tab1.col = tab3.col;
```