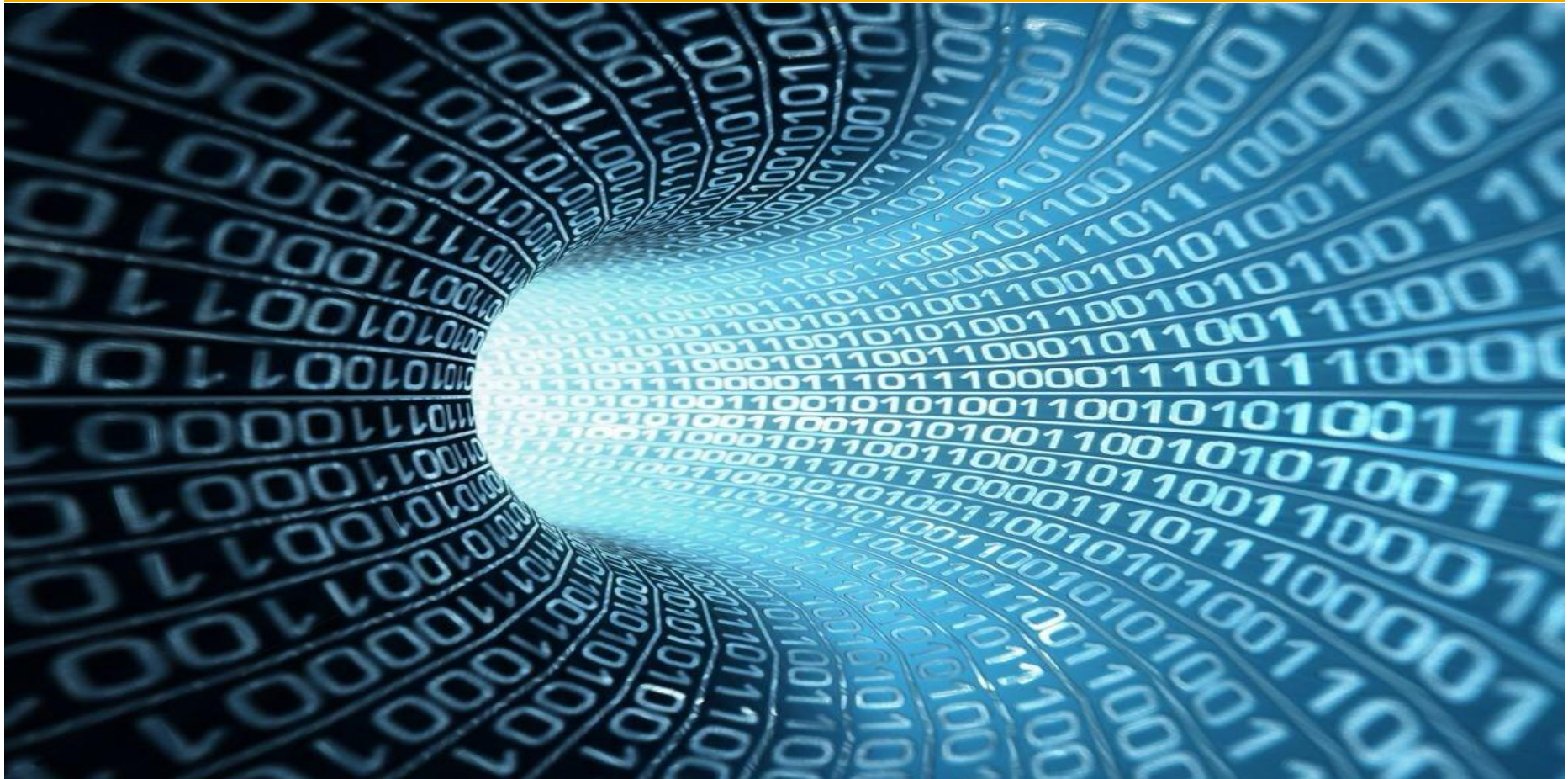


# **Vorlesung Datenbanken 2**

## **Teil I: Funktionsweise relationaler Datenbanksysteme**

Prof. Dr. Zoltán Nochta



---

# 1. Einleitung

Was ist eine **Datenbank (DB)** eigentlich?

**Konzeptionelle Sicht:** Strukturierte Menge von in einem gegebenen Anwendungskontext zusammengehörigen Daten.


■ **Beispiel:** Tabellen mit Daten über Firmenmitarbeiter („*HRDB*“)

**Technische Sicht:** Datenbestand zur Verwaltung einer oder mehrerer konzeptioneller Datenbanken.


Wird via *CREATE DATABASE* in relationalen Datenbanksystemen erzeugt.

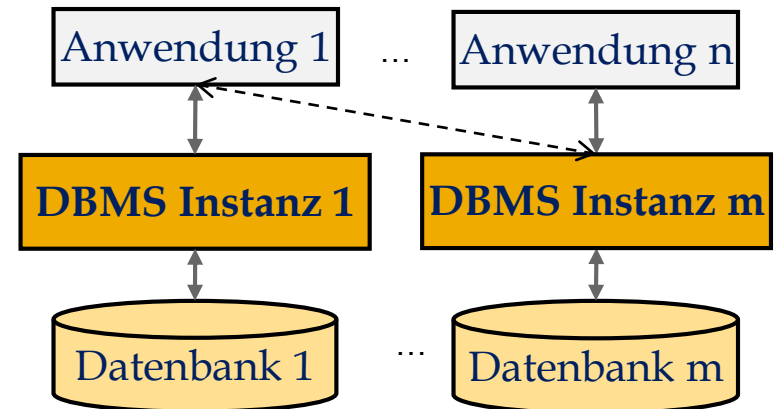
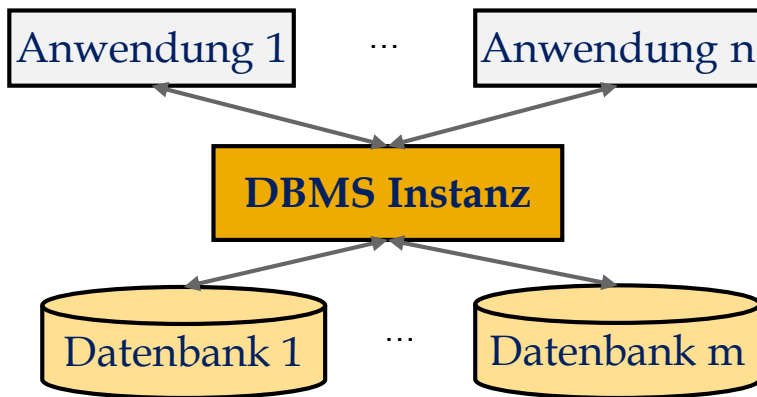
■ **Beispiel:** DB-, Index-, Log-, Temp-, Konfig- und Backup-Dateien zu *HRDB*


■ Beachte:

- Es gibt **produkt- bzw. herstellerspezifische** Definitionen des Begriffs
- In einer Datenbank (technische Sicht) können Daten aus mehreren, voneinander vollkommen unabhängigen konzeptionellen Datenbanken gespeichert sein. 

# Datenbankmanagementsystem (DBMS)

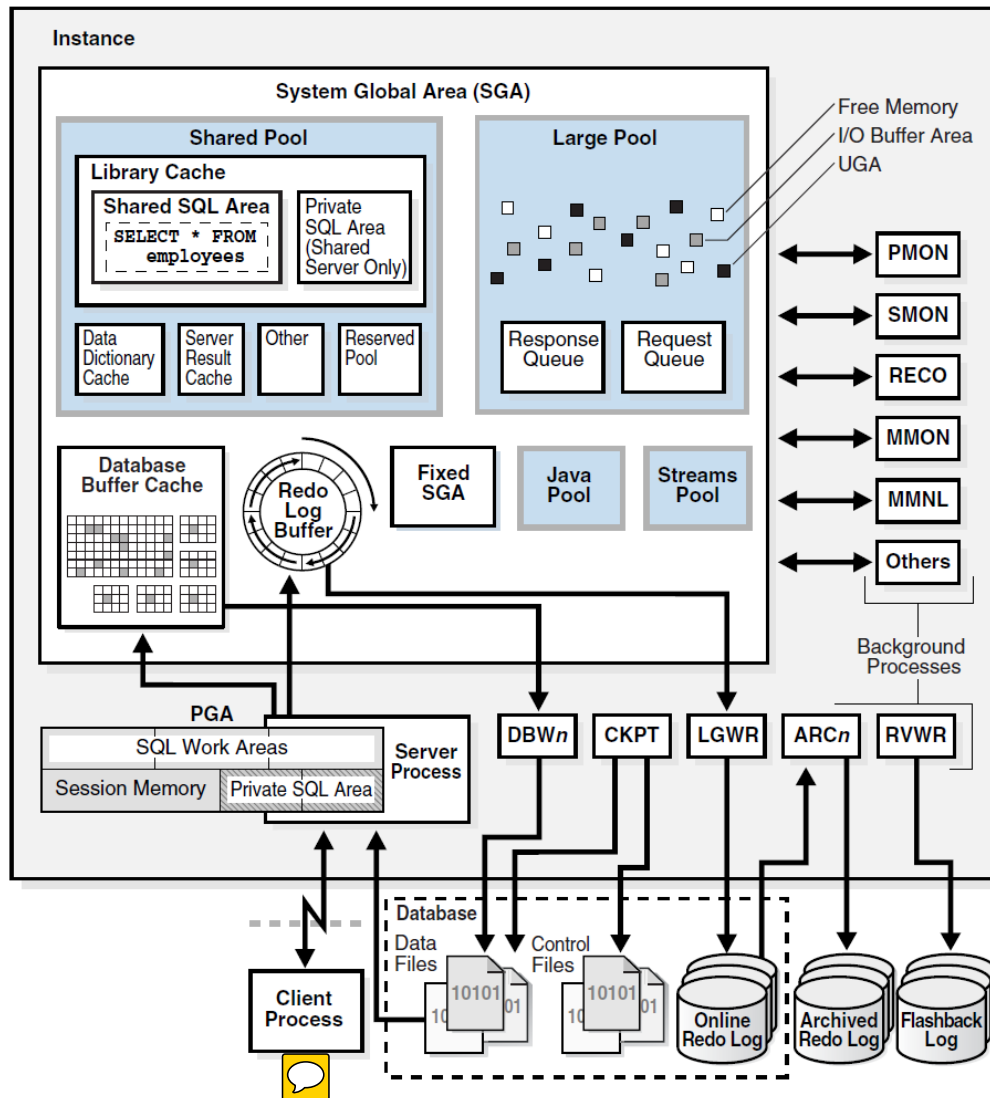
- **DBMS** - “Betriebssystem der Datenbank” 
  - **Softwaresystem** für die umfassende Verwaltung von Datenbanken
  - Eine **DBMS-Instanz** belegt zur Laufzeit dezidierten Bereich im Hauptspeicher, verfügt über ihr zugeordnete Prozesse. Sie verwaltet eine oder mehrere Datenbanken, je nach DBMS-Produkt:



- Auf demselben Rechner/OS können mehrere DBMS-Instanzen **parallel** laufen.
- Beachte: Neben Anlegen einer DB kann *CREATE DATABASE* auch eine neue DBMS-Instanz erzeugen. 

# Speicherkomponenten und Prozesse einer DBMS-Instanz

## Beispiel Oracle



## Komponenten/Prozesse eines DBMS

**DBW:** Database Write

**CKPT:** Checkpointing

**LGWR:** Redo Log-writer

**ARCn:** Log-Archivierung

**RVWR:** Undo-Versionslog für „Zeitreisen“

**PMON:** Prozessmonitor




**SMON:** Systemmonitor

**RECO:** Recovery verteilter Transaktionen

**MMON/MMNL:** Manageability Monitor

# Funktionale Anforderungen an DBMS



---

- Aufgaben eines DBMS nach Codd (=>1982...) 
- **Integration:** einheitliche, nicht-redundante Datenverwaltung gemäß einem *konzeptuellen Datenmodell* (relational, objekt-orientiert, usw.)
- **Operationen:** Speichern, Suchen, Ändern von Daten
- Anbieten von **Benutzersichten** auf die Daten 
- **Zugriffskontrolle:** Ausschluss unauthorisierter Zugriffe auf die Daten 
- Verwaltung und Zugriff auf **Metadaten** im Katalog (engl. Data Dictionary)
- Überwachung der **Konsistenz/Integrität:** Korrektheit der Daten
- **Transaktionen:** Mehrere DB-Operationen als atomare Funktionseinheit
- **Synchronisation:** Koordinierung nebenläufiger Transaktionen
- **Datensicherung:** Wiederherstellung von Daten nach Systemfehlern



# Einige nicht-funktionale Anforderungen an DBMS heute

---

- Gleichzeitige Bedienung von **OLAP**- (On-Line Analytical Processing) und **OLTP**-Workloads (On-Line Transaction Processing) 
- Umgang mit großen Datenmengen, vielfältigen Zugriffsmustern, Datenquellen und -Formaten 
- **Ausnutzung** aktueller Fähigkeiten der **Hardware**:
  - Multi-CPU / Multi-Core Architekturen
  - Parallele Datenverarbeitung
  - Neue Speichertechnologien
  - Hochleistungsnetze
- Gute **Performance** (Durchsatz, Antwortzeit) auch unter hoher Last
- Vertikale und horizontale **Skalierbarkeit**
- **Verteiltes Daten- und Transaktionsmanagement**

# Relationales Datenmodell

## Überblick Grundkonzepte und Begriffe #1


---

<b><u>Begriffe:</u></b>	<b><u>Informelle Bedeutung:</u></b>
<b>Attribut</b>	Spalte einer Tabelle
<b>Wertebereich/Domäne</b>	mögliche Werte eines Attributs
<b>Attributwert</b>	Element eines Wertebereichs
<b>Relationenschema</b>	Menge von Attributen
<b>Relation</b>	Menge von Zeilen einer Tabelle
<b>Tupel</b>	Zeile einer Tabelle
<b>Datenbankschema</b>	Menge von Relationenschemata
<b>Datenbank</b>	Menge von Relationen (Basisrelationen)
<b>Basisrelation</b>	In der Datenbank aktuell vorhandene Relation zu einem bestimmten Relationenschema



# Relationales Datenmodell

## Überblick Grundkonzepte und Begriffe #2

<u>Begriffe:</u>	<u>Informelle Bedeutung:</u>
<b>Schlüssel</b>	 Minimale Menge von Attributen, deren Werte ein Tupel der Relation eindeutig identifizieren
<b>Primärschlüssel</b>	Ein beim Datenbankentwurf ausgezeichneteter Schlüssel
<b>Fremdschlüssel</b>	Attributmenge, die in einer anderen Relation Schlüssel ist
<b>Fremdschlüsselbedingung</b>	Alle Attributwerte des Fremdschlüssels tauchen in der anderen Relation als Werte des Schlüssels auf
<b>Nicht-Primattribut</b>	Attribute, die nicht in einem Schlüssel auftauchen

# Relationales Datenmodell

## Relationale Algebra

---

### Relationale Algebra

- Theoretische Grundlage der in SQL realisierten **Mengenoperatoren**
- Standard Symbolik und Formalismus
  - Der Formalismus kann **keine Rekursion, Sortierung**, etc. ausdrücken
  - Wir greifen in der Vorlesung auf die **hervorgehobenen Symbole** zurück

$\sigma$	: <b>relation</b> $\times$ $\Theta \rightarrow$ <b>relation</b>	(Selektion)
$\pi$	: <b>relation</b> $\times$ <b>attributfolge</b> $\rightarrow$ <b>relation</b>	(Projektion)
$\times$	: <b>relation</b> $\times$ <b>relation</b> $\rightarrow$ <b>relation</b>	(Kartesisches Produkt)
$\cup$	: <b>relation</b> $\times$ <b>relation</b> $\rightarrow$ <b>relation</b>	(Vereinigung)
$/$	: <b>relation</b> $\times$ <b>relation</b> $\rightarrow$ <b>relation</b>	(Differenz)
$\cap$	: <b>relation</b> $\times$ <b>relation</b> $\rightarrow$ <b>relation</b>	(Durchschnitt)
$\bowtie_{\theta}$	: <b>relation</b> $\times$ $\Theta \times$ <b>relation</b> $\rightarrow$ <b>relation</b>	(Theta-Verbindung)
$\bowtie$	: <b>relation</b> $\times$ <b>relation</b> $\rightarrow$ <b>relation</b>	(natürliche Verbindung)
$\div$	: <b>relation</b> $\times$ <b>relation</b> $\rightarrow$ <b>relation</b>	(Division)
$\Theta$	: <b>Universum logischer Prädikate (Bedingungen);</b>	

# Relationales Datenbankmanagementsystem (RDBMS)

---

- **RDBMS** ermöglicht die Definition, Speicherung und Verwaltung von Datenbankobjekten gemäß **relationalem Modell** über definierte Schnittstellen und Sprachen (SQL DDL, DML, DCL).
- Elemente und Beziehungen eines **konzeptionellen und normalisierten** (Redundanzfreiheit) Datenmodells (manifestiert als *UML-, ER-Modell*) sowie **Integritätskriterien** werden im RDBMS als Tabellendefinitionen via *CREATE TABLE* festgehalten.
- Pro Tabelle können außerdem **Zugriffswege** (z.B. Indexe), Art der **Speicherung** (z.B. Partitionierung), **Zugriffskriterien** (Zugriffskontrolle) etc. festgelegt werden. Dadurch entstehen weitere (Meta-)Daten, die das DBMS ebenfalls konsistent verwalten muss.
- Neben Tabellen gibt es zahlreiche weitere Datenbankobjekte, wie z.B. **Sichten** (engl. *views*), **Trigger**, **Sequenzen**, **Synonyme**, u.v.m.

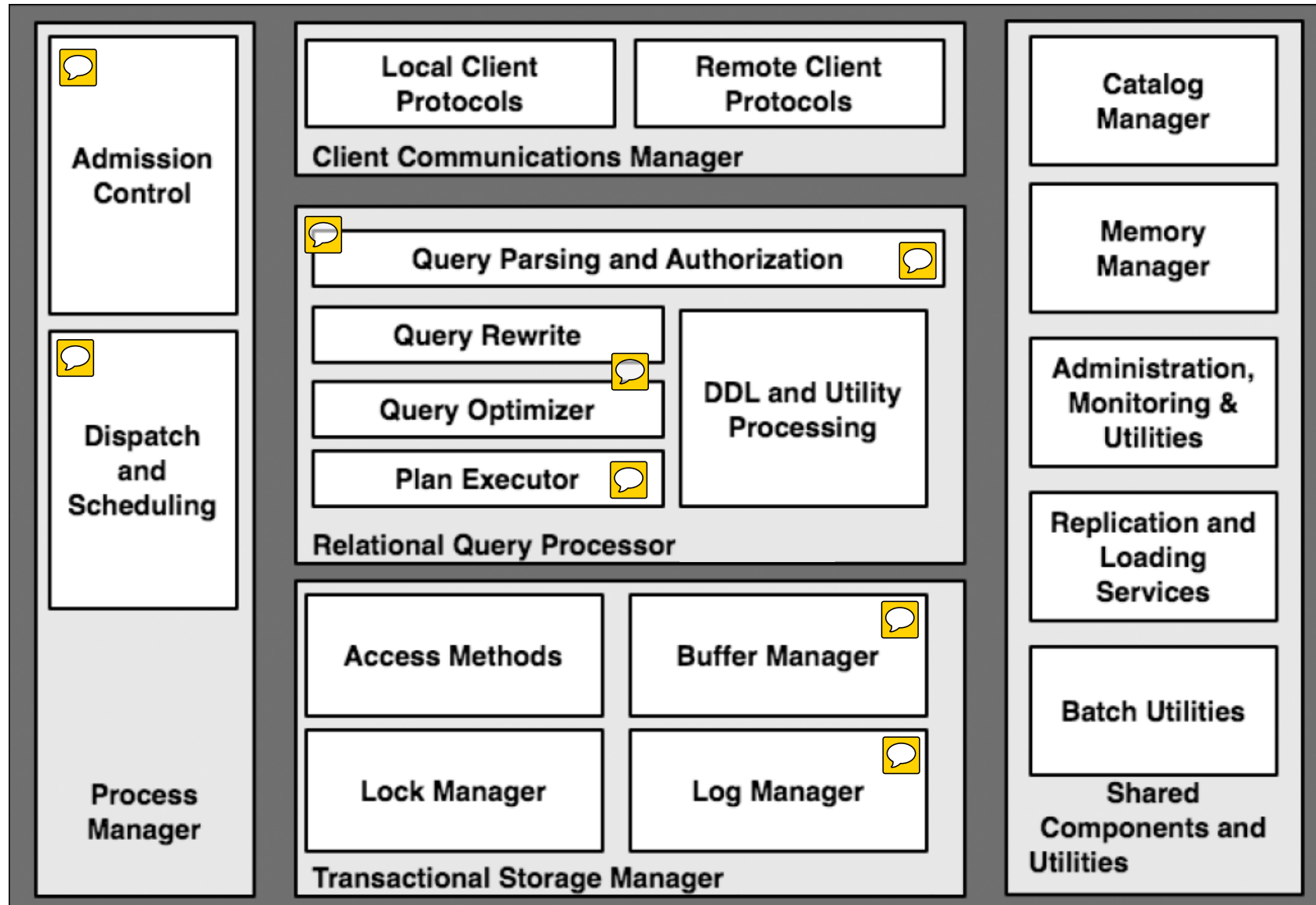


---

## 2. Aufbau relationaler Datenbankmanagementsysteme (RDBMS)

# Komponenten eines RDBMS

## Konzeptioneller Überblick



- Das Architekturbild zeigt einige Komponenten, ohne deren Interaktionen, die in den meisten RDBMS-Produkten üblicherweise realisiert sind.
  - **Funktionalität und Bezeichnungen** einzelner Komponenten sind **herstellerspezifisch** und können in Produktdokumentationen, sofern verfügbar, nachgeschlagen werden.
- Es gibt keinen Standard, **wie** die Daten intern gespeichert bzw. wie lesende Anfragen oder schreibende Transaktionen in DBMS-Produkten verarbeitet werden.
  - Lediglich einige **Schnittstellen** sind **standardisiert** (z.B. SQL, ODBC).
  - Es existieren **proprietäre Zugriffsformen**, Protokolle, Datenstrukturen, usw. je nach Optimierungszweck und besonders adressiertem Anwendungssegment des jeweiligen DBMS-Produkts.
  - Dies **erschwert die Portierbarkeit** von Anwendungsprogrammen, Datendefinitionen und auch der Daten selbst und führt auch zu Problemen insbesondere in verteilten (heterogenen) Datenbanksystemen.



# Komponenten eines RDBMS

## Erläuterungen #1

---

### ■ Client Communications Manager:

- **Verbindungsaufbau und Kommunikation** mit Clients (SQL-Konsole, Webserver, ...) über unterstützte Protokolle und API (*ODBC, JDBC, ...*).
- Leitet SQL-Anfragen eines Clients an i.d.R. einen dedizierten *DBMS-Worker* zur Verarbeitung weiter, sendet Ergebnisse/Fehlermeldungen an Client zurück

### ■ Process Manager: Startet DBMS-Worker entweder in Form von **OS-Prozessen** (*Oracle, PostgreSQL*) oder als **Threads** im selben OS-Prozess (*MS SQL Server, IBM DB2, MySQL*).

- **Zugangskontrolle (admission control):** Überwacht die *Anzahl aktiver Worker* je nach gemessenem oder geschätztem CPU- und Hauptspeicherbedarf der SQL-Anfragen. Je nach Auslastung verzögert das Starten neuer Worker.

- **Scheduler (dispatch):** Verteilt Ressourcen, insb. CPU-Zeit, zwischen konkurrierenden Worker dynamisch, gemäß DBMS-spezifischen Verfahren und Protokollen (Transaktionsverwaltung, Sperrprotokoll, Logging, usw.)

# Komponenten eines RDBMS

## Erläuterungen #2

---

- **Relational Query Processor:** Überprüft und setzt *mengenorientierte* SQL-Anfragen auf DB-interne *satzorientierte* Strukturen und Operatoren um
  - **Syntaxanalyse und Zugriffskontrolle (parsing and access control):** Ist die Anfrage syntaktisch korrekt, verletzt sie evtl. Integritätsbedingungen, hat der angemeldete Benutzer die erforderlichen Zugriffsrechte?
  - **Umformung (query rewrite):** Löst Sichten auf, ersetzt Synonyme durch ‚echte‘ Namen der DB-Artefakte, usw.
  - **Optimierung (query optimizer):** Erzeugt einen optimierten **Ausführungsplan** für die Anfrage. Der Plan enthält interne **spezialisierte Operatoren** für Table-Scans, verschiedene Join-Arten, Aggregatfunktionen (AVG, SUM, ...), Sortierung, usw.
  - **Ausführung (plan executor):** Führt den erzeugten Plan aus und benutzt dabei Komponenten, die ihrerseits auf interne/ physische Datenstrukturen im Sinne der ACID-Kriterien zugreifen.
- **DDL and Utility Processing:** Setzt Befehle gemäß *Data Definition Language (DDL)* wie **create table**, **alter table**, **drop table**, ... um. Meistens keine Umformung, Optimierung, etc. solcher Befehle notwendig.

# Komponenten eines RDBMS

## Erläuterungen #3

---

- **Transactional storage manager:** Stellt die Einhaltung der **ACID-Kriterien** in der Datenbank sicher und optimiert die Datenzugriffe. Komponenten sind funktional eng miteinander verwoben und aufeinander abgestimmt.
  - **Zugriff auf Nutzdaten und Indexdaten (access methods):**
    - Algorithmen zum Lesen (SELECT) und zur Pflege (INSERT, UPDATE, DELETE) der Nutzdaten
    - Algorithmen zur Suche in DB über vorhandene Indexstrukturen (B<sup>+</sup>-Bäume, Hashtabellen, etc.)
  - **Sperrverwalter (lock manager):** Verwaltet **Sperren** (*lock, unlock, upgrade*) auf Daten gemäß Sperrprotokoll (*2PL, S2PL, SS2PL*) bzw. überwacht nicht-sperrende Verfahren (*MVCC, Optimistische Verfahren*)
  - **Logging (log manager):** Stellt die **reihenfolgetreue Persistierung** aller schreibenden Operationen nach dem WAL-Prinzip (*Write-ahead Logging*) sicher.
  - **Pufferverwaltung (buffer manager):** Koordiniert, welche **Blöcke** von der Festplatte wann in Hauptspeicher (RAM) geladen bzw. wieder verdrängt und somit in DB auf der Festplatte persistiert werden.

# Komponenten eines RDBMS

## Erläuterungen #4

---

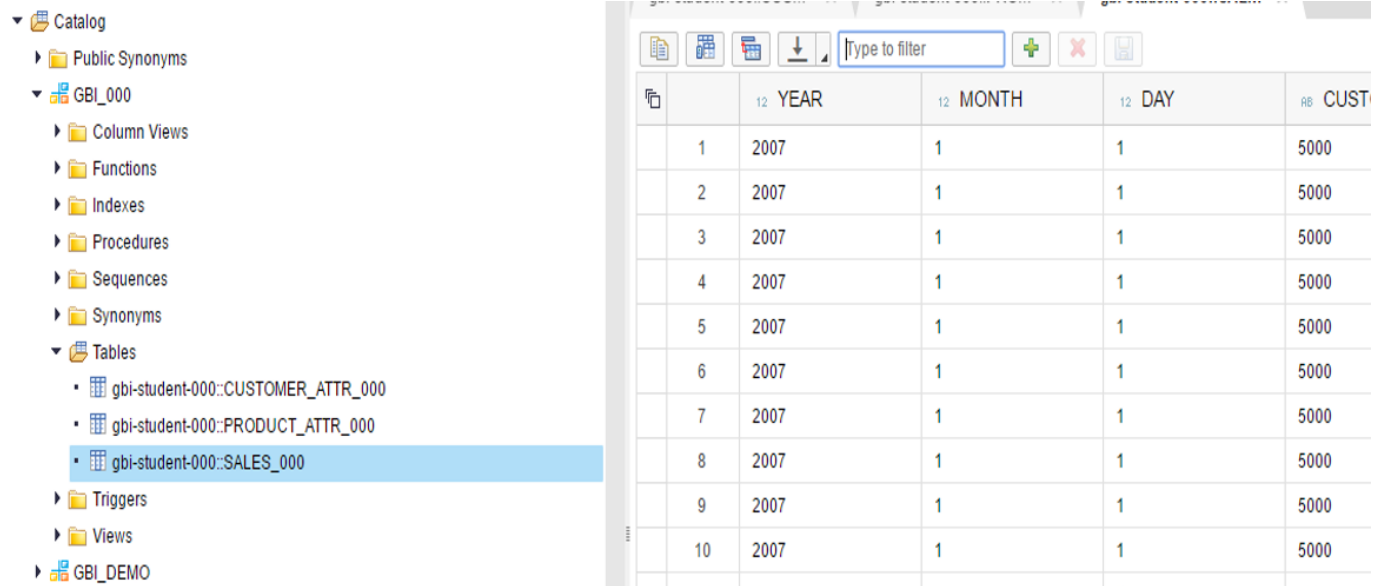


- **Katalogsystem** (engl. **catalog**, **data dictionary**): Beinhaltet alle **statischen** und **dynamischen** Informationen, die zur **Verwaltung** des gesamten Datenbestands notwendig sind.
  
- Welche Daten gehören dazu? Beispiele:
  - **Metadaten** über alle in der Datenbank angelegten Artefakte (inkl. Tabellen- und Sichtendefinitionen)
  - **Zugriffsrechte**
  - Name und Speicherort angelegter **Datendateien, Log-Dateien, usw.**
  - **Konfigurationsparameterwerte**
  - Informationen über aktuell laufende **Transaktionen** mit ihren Datenbankoperationen (physical reads / writes, ...)
  - **Statistiken** über Zugriffsverhalten, Speicherverwaltung etc. für die Optimierung der Anfragen und für das Tuning
  - u.v.m.

# Komponenten eines RDBMS

## Erläuterungen #5

- Der Katalog wird üblicherweise selbst in Form von **Systemtabellen** repräsentiert, die von Benutzern je nach Rolle und Autorisierung gelesen werden können.
- Bspw. in Oracle Datenbanken gibt es Sichten (views) wie *user\_tables*, *user\_trigger*, ..., *dba\_tables*, *dba\_trigger*, *dba\_objects*, ... .
- Viele Systeme bzw. Hersteller bieten auch Zugriff über GUI an:



The screenshot displays a database management interface. On the left, a 'Catalog' tree shows a hierarchy of database objects. The 'Tables' folder is expanded, and 'gbi-student-000::SALES\_000' is selected. On the right, a data table is shown with columns for row number, year, month, day, and customer ID. The table contains 10 rows of data, all with a customer ID of 5000.

	12	12	12	12	12
	YEAR	MONTH	DAY	CUST	
1	2007	1	1	5000	
2	2007	1	1	5000	
3	2007	1	1	5000	
4	2007	1	1	5000	
5	2007	1	1	5000	
6	2007	1	1	5000	
7	2007	1	1	5000	
8	2007	1	1	5000	
9	2007	1	1	5000	
10	2007	1	1	5000	

# Komponenten eines RDBMS

## Erläuterungen #6

---

- Der Katalog wird üblicherweise selbst in Form von **Systemtabellen** repräsentiert, die von Benutzern je nach Rolle und Autorisierung gelesen werden können.
- Der für das Datenbanksystem reservierte **Hauptspeicher** muss so gewählt sein, dass der Katalog zu jedem Zeitpunkt komplett darin enthalten ist!
- Dies ist vor allem deshalb notwendig, da das laufende DBMS, das **Betriebssystem der Datenbank**, stets Informationen aus dem Katalog braucht.




---

### 3. Speicherung und Verwaltung der Daten in RDBMS

- **Relationale** DBMS erlauben die Manipulation und das Abfragen der Daten gemäß *relationalem Datenmodell* mittels **SQL**, die i.W. eine **mengenorientierte** Anfragesprache ist.
- Das DBMS speichert, adressiert und verwaltet Tabellendaten intern **satzorientiert**: Ein **interner, physischer Datensatz** (engl. **record**) entspricht typischerweise **einer Tabellenzeile**.
- Die Kommunikation zwischen Endbenutzern und RDBMS erfordert also die interne **Transformation** *mengenorientierter SQL-Anfragen* gegen Tabellen in *physische satzorientierte* Lese- bzw. Schreiboperationen, sowie Adress-Informationen in Speichermedien (Festplatte, SSD, usw.).

# Abbildung Datenbankobjekte auf interne Datenstrukturen

---

- Pro Relation einer DB wird DBMS-intern eine „**logische Datei**“ angelegt.
  - Sie fasst **interne Datensätze** der Relation zusammen, welche neben **Nutzdaten** (Tupel) auch Verwaltungsdaten (**Overhead**) enthalten. 
- Logische Dateien einer DB werden häufig in einer einzigen **physischen**, d.h. fürs Betriebssystem sichtbaren Datei, auf der Festplatte gespeichert.
  - Dateien bestehen aus **Seiten** fixer Größe (z.B. 8 KB), welche in **Festplattenblöcken** ggfs. der gleichen Größe persistiert werden.
  - **Lokalität**: Datensätze einer Tabelle liegen oft in einem zusammenhängenden Speicher- bzw. Adressbereich. Vorteilhaft für Table-Scans, weniger passend für Verbundberechnung (Join).
- Zugriffspfade, wie z.B. **Indexbäume**, werden nach dem gleichen Prinzip persistiert.
- Beachte: Die Abbildung **variiert** je nach **DBMS-Design und Hersteller**.

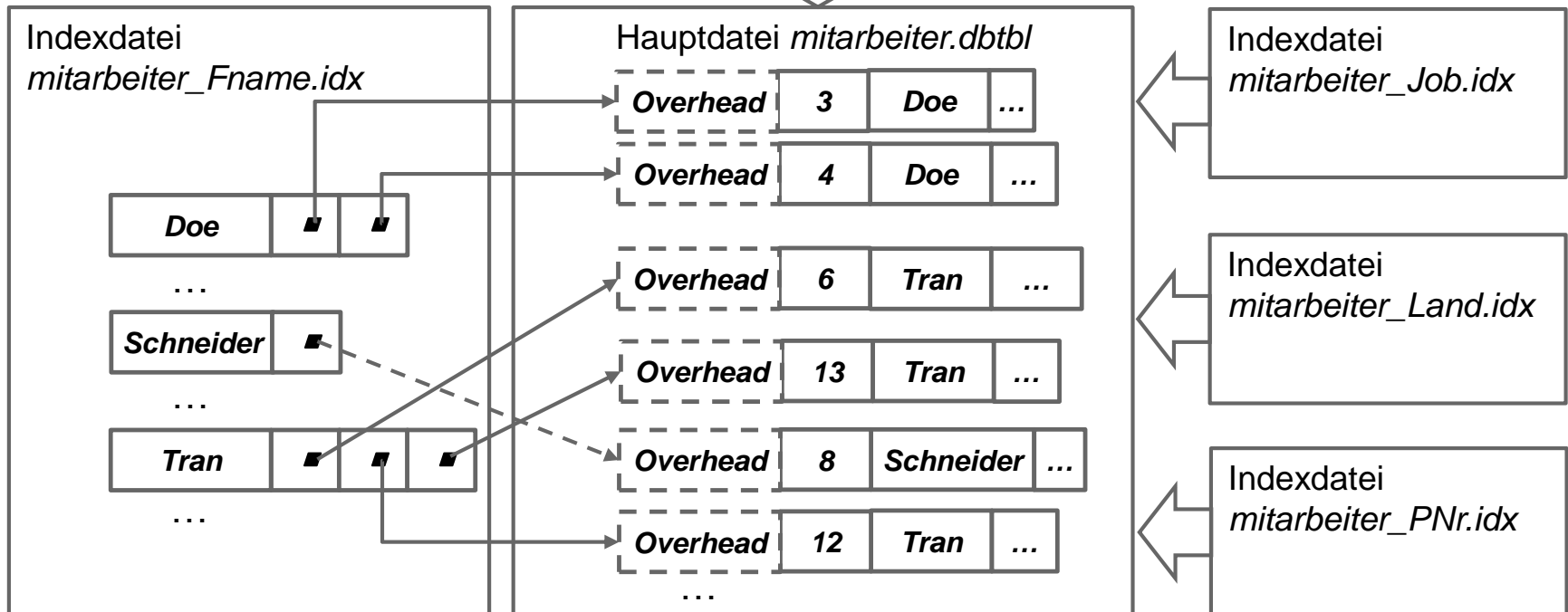


# Speicherung von Datenbankobjekten

## Logische Dateien

- **Datensätze der Relation *mitarbeiter* und zugehörige Indexdaten** werden in (meistens) nur DBMS-intern sichtbaren **logischen Dateien** gespeichert:

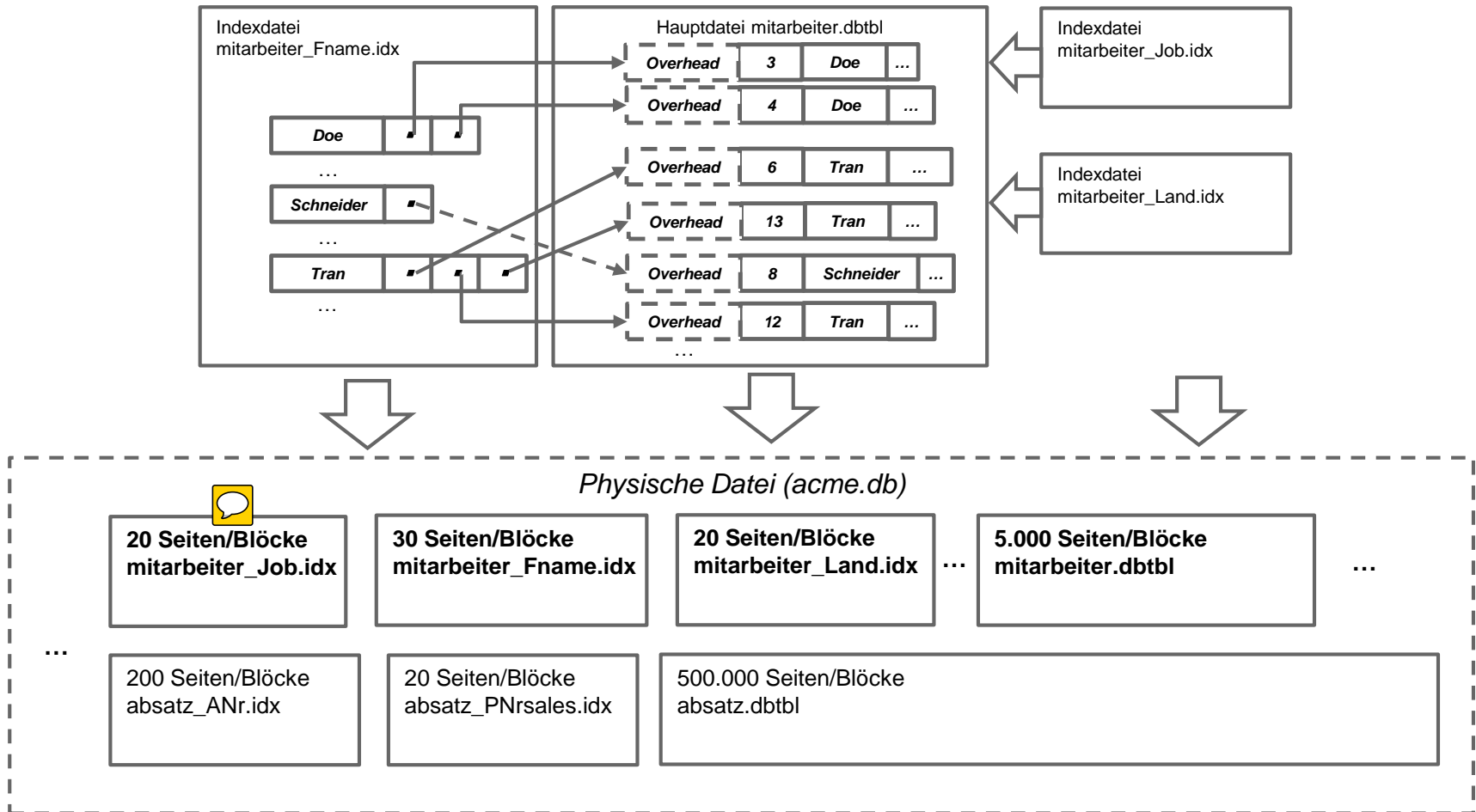
PNr	Fname	Land	Job	Gehalt
1	Pham	VN	Admin	4500
2	Schmidt	DE	Sales	45000
3	Doe	US	Sales	70000
4	Doe	US	Director	150000
5	Mueller	DE	CEO	500000
6	Tran	VN	Engineer	15000
7	Le	VN	Developer	12000
8	Schneider	DE	Admin	12000
9	Smith	US	Admin	18000
10	Klein	DE	Developer	40000
11	Pham	VN	Engineer	16000
12	Tran	VN	Sales	21000
13	Tran	VN	Expert	8000
14	Le	VN	Expert	8500



# Speicherung von Datenbankobjekten

## Physische Dateien

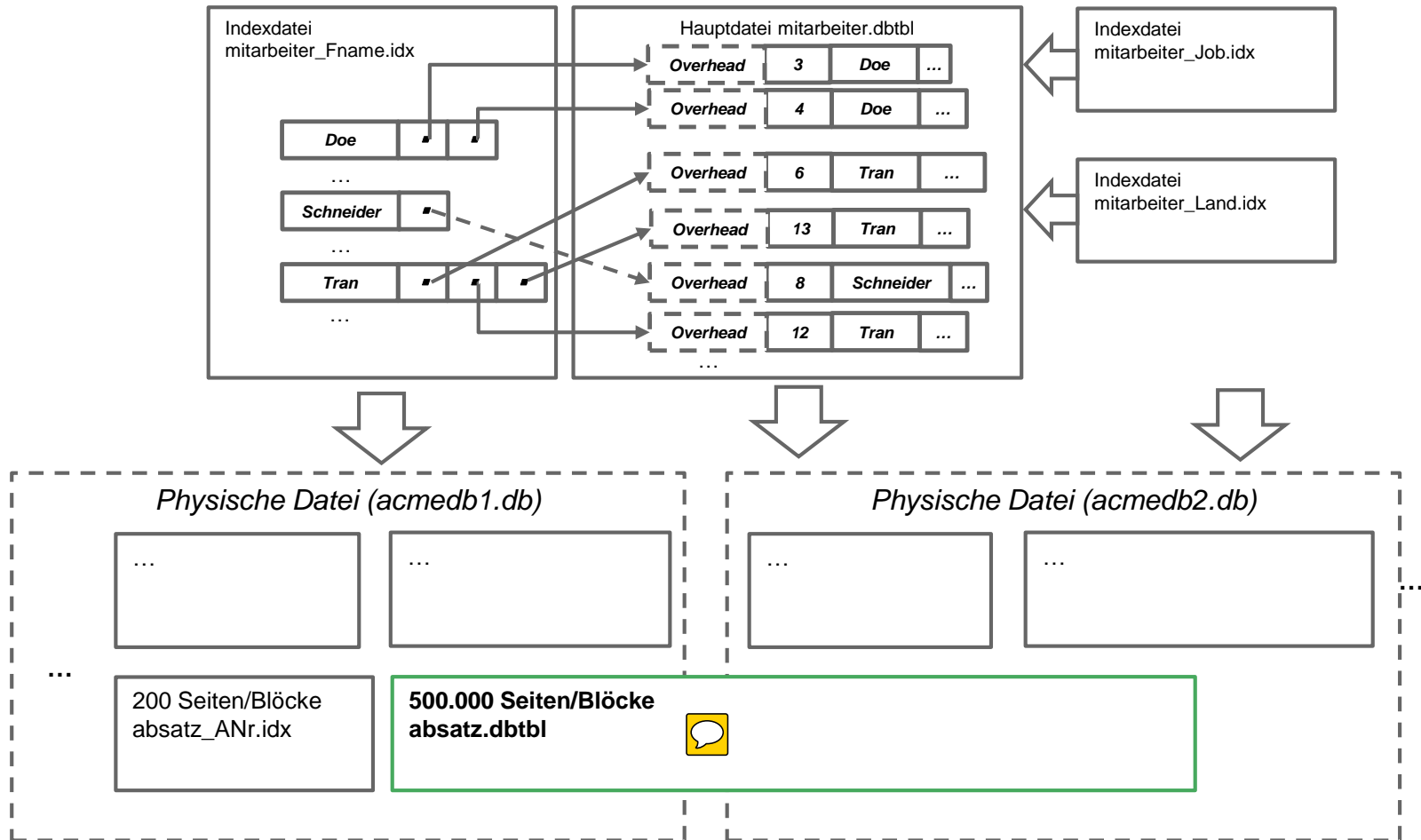
- Logische Dateien bilden zusammenhängende **Bereiche** in einer auch fürs Betriebs- bzw. Filesystem sichtbaren **physischen DB-Datei**:



# Speicherung von Datenbankelementen

## Physische Dateien

- Eine logische Datei kann auch in **mehreren DB-Dateien** persistiert sein, wobei jeder Datensatz i.d.R. in genau einer DB-Datei abgelegt wird.

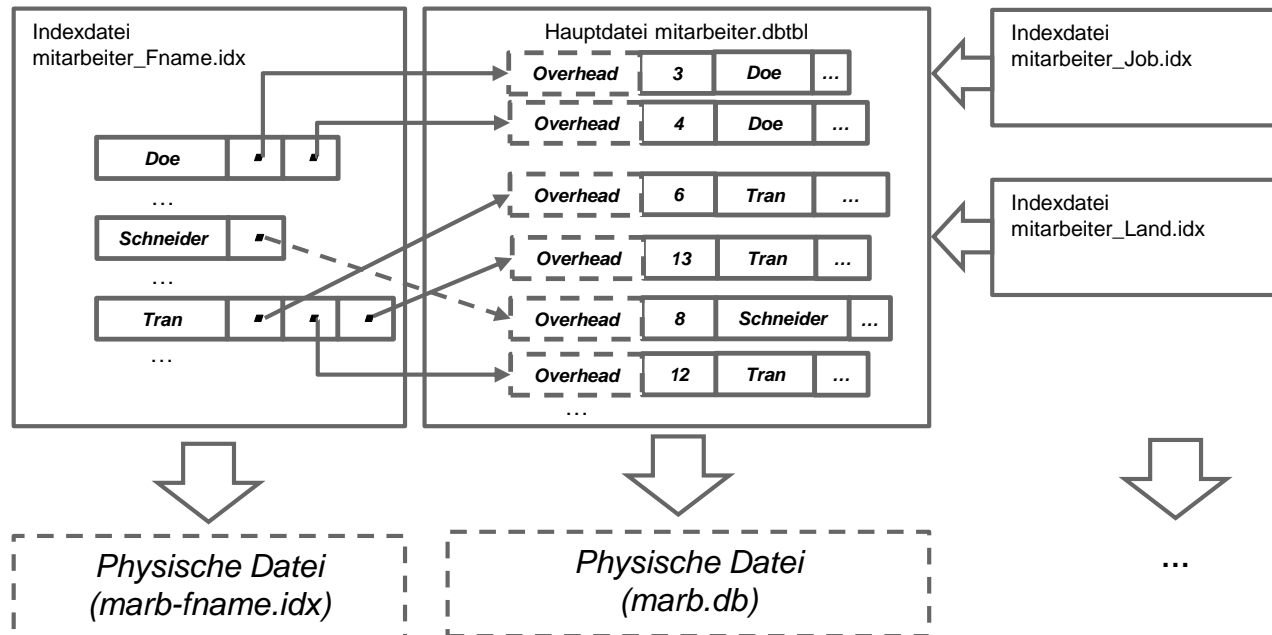




# Speicherung von Datenbankelementen

## Physische Dateien

- Einige RDBMS verwalten **eine physische Datei pro DB-Objekt**
- Alle Objekte/Dateien einer Datenbank werden in einem deziidierten Verzeichnis im Filesystem abgelegt.



- Siehe bspw. in PostgreSQL die Tabellen im Katalog *pg\_database* (*oid*=Verzeichnisname mit DB-Objekten) und *pg\_class* (*oid*=Dateiname eines DB-Objekts)

# Dateiverwaltung in RDBMS

## Aufgaben

---

Aufgaben der (physischen) **Dateiverwaltung (file manager)** sind:

- Bedienung der **Geräteschnittstelle** der Festplatte (SATA, PATA, SCSI, USB, ...)
  - **Dateien** anlegen, öffnen, schließen
  - Einzelne **Blöcke** der Datei je nach Anforderung lesen und schreiben
  - **Prüfung** von Lese- u. Schreiboperationen
  - **Freispeicherverwaltung** auf der Platte
- 
- Die Dateiverwaltung kann vom **Betriebssystem** erledigt werden.
  - Häufig steuert DBMS selbst die Festplatte an (**raw device**) und arbeitet mit den Blöcken der DB-Datei in ihrer Ursprungsform ohne das Betriebssystem zu fragen.
    - Welche Vor- und Nachteile dieser Strategie können Sie erkennen?

# Speicherung von Datensätzen in Seiten

---



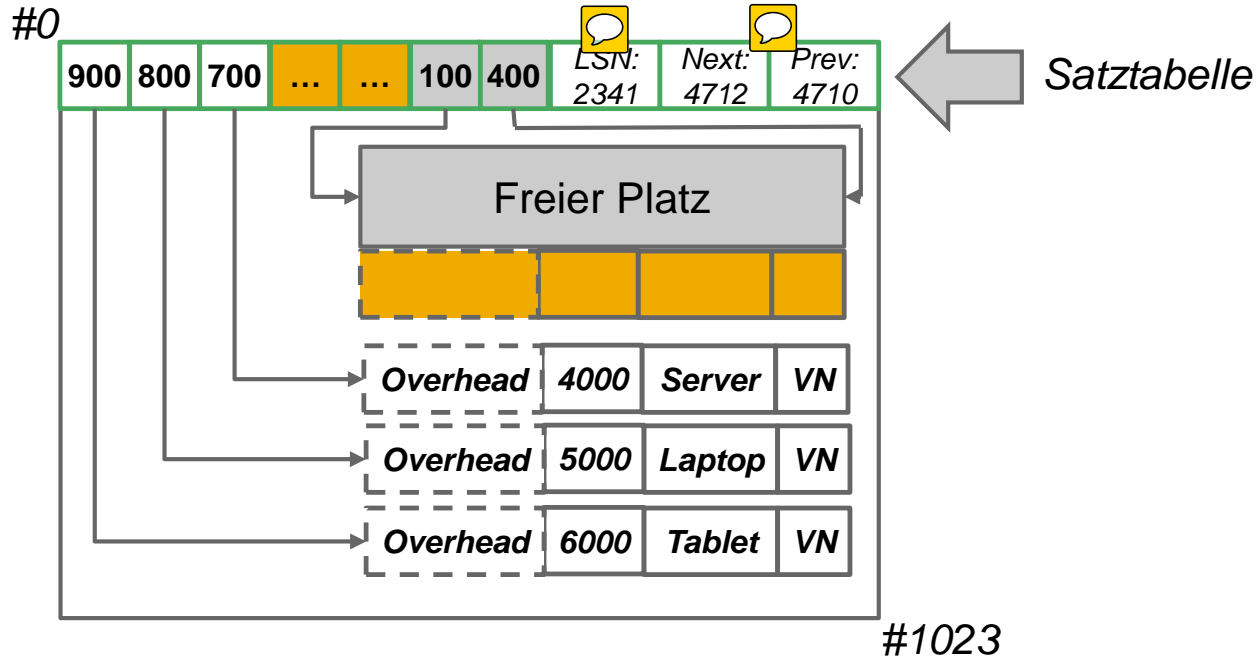
- Seiten der DB-Datei bzw. Festplattenblöcke haben i.d.R. **fixe Größe**
  - Manche Hersteller erlauben den Default-Wert zu verändern
- Seiten speichern **Datensätze** und **Verwaltungsinformation** in Form einer **Satztabelle**.

Dazu gehören, bspw.:
- *Offset-Adressen* (sog. **Slots**) der in der Seite abgelegten Datensätze relativ zum Seitenbeginn, d.h. zum ersten Byte der Seite.
- Beginn und Ende des *freien Platzes* in der Seite
- Verknüpfung mit *Logeinträgen* (LSN) im Logfile der gegebenen Datenbank
- *Verweise* auf andere Seiten (*Next*, *Prev.*) um sequentielle Suchen im Datenbestand zu beschleunigen.

# Speicherung von Datensätzen in Seiten

## Beispiel

- Seite ,4711' mit 1024 Byte Kapazität speichert Tupel aus *produkt*:



- Slots sind 100 Byte lang => Datensätze könnten auch dichter gepackt sein.
- Freiplatz beträgt aktuell 300 Bytes, also drei Slots.
- Beachte: Seiten sind „*linear adressierte Speicher*“, s. Byte #0 - #1023 oben



# Konfiguration der Seitennutzung

---

Einige RDBMS erlauben die Konfiguration der Seitennutzung für DB-Objekte.

Beispiel: Oracle DBMS im Falle der Tabelle *produkt*:

```
CREATE TABLE produkt (  
  ANr int, PName varchar(30), PLand varchar(3), primary key (ANr),  
  ...  
  pctfree 10, pctused 40 );
```

-  **PCTFREE** *x*: Einfügen (INSERT) neuer Datensätze möglich, wenn danach noch mind. *x*% des Blocks frei bleiben.

 Folge: Blöcke werden nie zu mehr als 100%-*x*% befüllt.

- **PCTUSED** *y*: Das Einfügen neuer Datensätze ist möglich, sobald mindestens *y*% eines vorher „vollen“ Blocks wieder frei sind.  
Folge: Einfügen i.d.R. nach einigen Löschoperationen (DELETE) möglich.

# Prinzip der indirekten Adressierung von Datensätzen

## Tupel Identifier

---

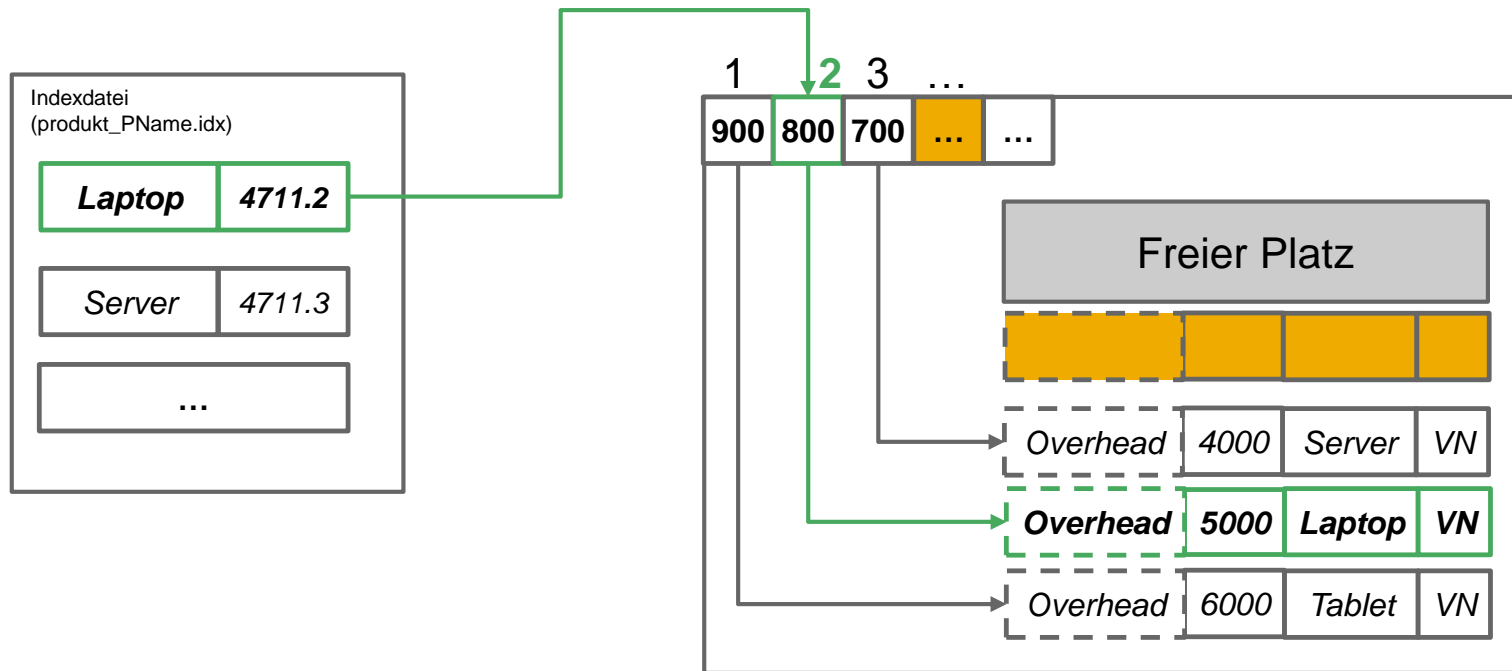
- **Prinzip indirekter Adressierung:** In Indexen (Suchbäumen) wird *nicht* die physische Adresse eines Datensatzes vermerkt, sondern dessen **Tupel-Identifier** (kurz TID), a.k.a. **record identifier, row identifier**.
- Ein TID ist also im Grunde ein Zeiger zum Datensatz. Er spezifiziert die **Seite und die Position der Offset-Adresse des Datensatzes** innerhalb der Satztabelle der jeweiligen Seite.
- **Vorteil der Indirektion:** Wenn Datensätze verändert (SQL: *UPDATE*), oder aus Platzgründen verschoben werden, müssen die Index-Strukturen nicht jedes Mal aktualisiert (d.h. gesperrt, neu berechnet, persistiert, ...) werden.



# Konzept Tupel Identifier

## Beispiel

- Seite mit der Nummer **4711** speichert  $k$  Zeilen von **produkt**
- Die Satztabelle enthält  $k$  Zeiger zu den Datensätzen in 4711
- Zeiger  $i$  gibt Offset des Datensatzes relativ zum Seitenanfang in Bytes an
- TID im Beispiel sind (4711.1), (**4711.2**), (4711.3), ...



Seite 4711

# Konzept Tupel Identifier

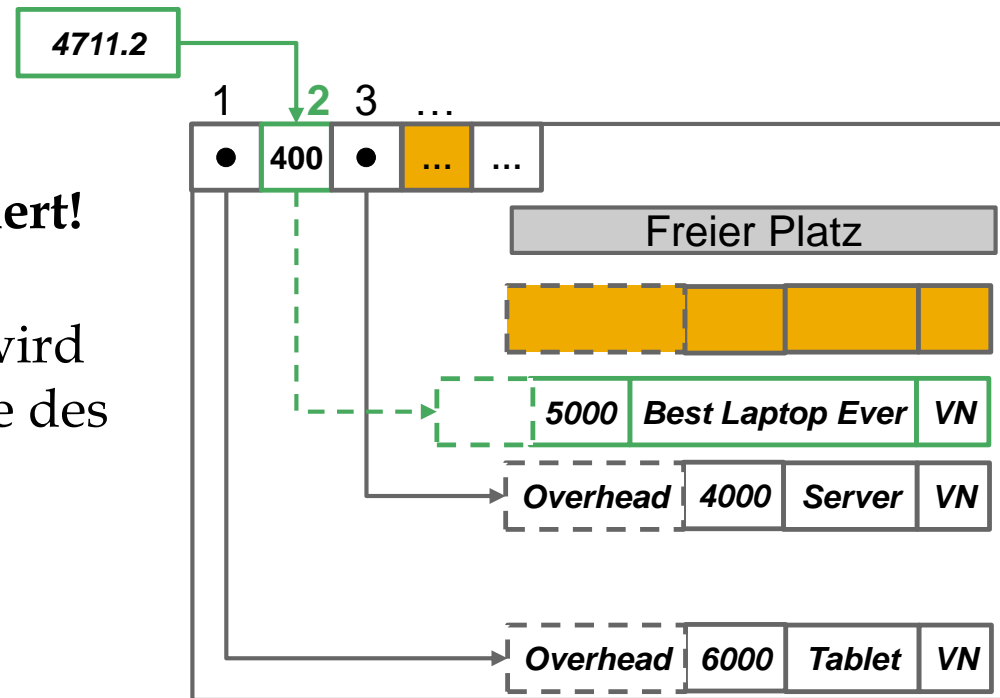
## Aktualisierung eines Datensatzes

*UPDATE produkt SET PName=,Best Laptop Ever' WHERE ANr=5000;*

- Da der Platz zwischen den Tupeln mit  $ANr=4000$  und  $ANr=6000$  nicht ausreicht, wird das Tupel mit  $ANr=5000$  innerhalb der Seite **verschoben**.

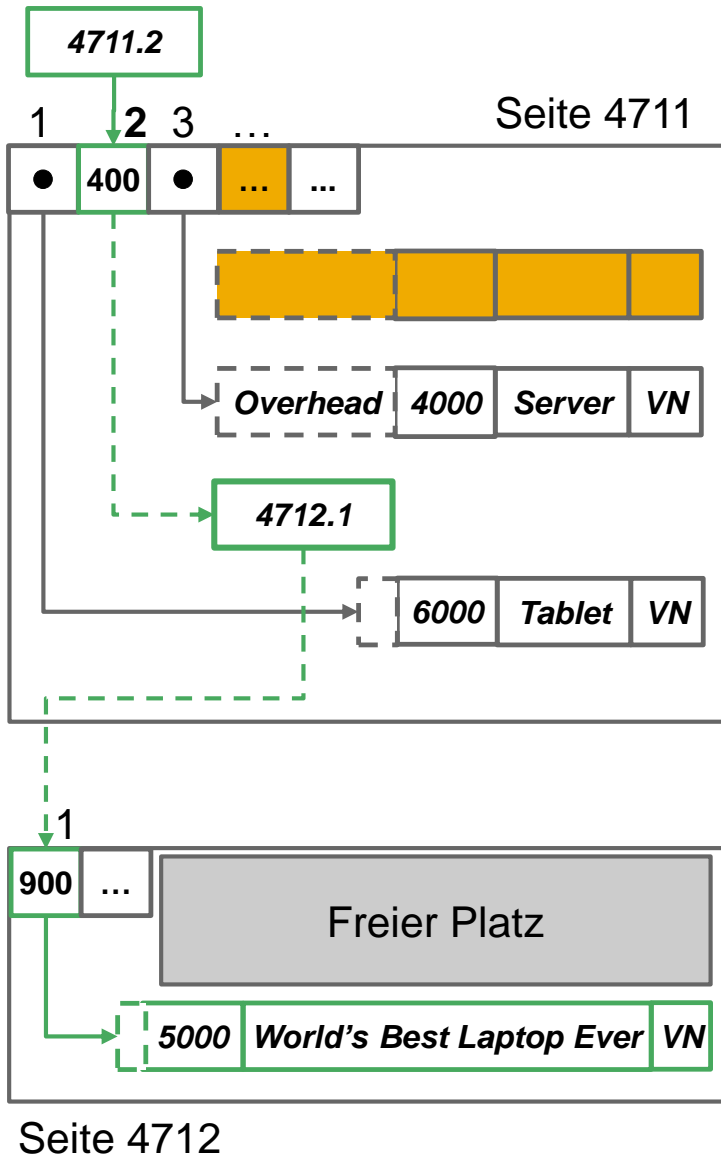
Beachte:

- **TID** des Tupels bleibt **unverändert!**
- **Offset-Pointer** an Position „2“ wird entsprechend der neuen Adresse des Datensatzes **angepasst**
- Freiplatz wird verkleinert
- Seite stärker fragmentiert als vorher



# Konzept Tupel Identifier

## Verschiebung von Datensätzen zwischen Seiten



- Eine weitere Aktualisierung des Tupels macht dessen **Verschiebung** auf die (neue) Seite **4712** notwendig.
- Seite 4711 enthält nun einen **internen Verweis** (4712.1) auf die neue Position des Tupels innerhalb der Seite ,4712‘
- **TID des Tupels bleibt unverändert!**
- Seite 4711 ist nun **stark fragmentiert**
- Um Tupel ANr=5000 zu erreichen, sind jetzt **zwei Blockzugriffe** notwendig ☹
- Dafür müssen Indexdateien nicht verändert werden ☺

# Tupel Identifier

## Abfrage und Kodierung

Abfrage TID der Tupel in **produkt** am Beispiel von PostgreSQL:

```
SELECT CTID, * FROM acme.produkt;
```

	ctid tid	ANr integer	PName character varying (30)	PLand character varying (3)
1	(0,1)	1000	Monitor	US
2	(0,2)	2000	Printer	DE
3	(0,3)	3000	PC	VN
4	(0,4)	4000	Server	VN
5	(0,5)	5000	Laptop	VN
6	(0,6)	6000	Tablet	VN
7	(0,7)	7000	Camera	VN
8	(0,8)	8000	Phone	VN
9	(0,9)	9000	Mouse	DE

- Beachte das Format: (*PageID*, *Slotnumber*)
- *PageID* gibt Position der Seite innerhalb der Datei an, die zur Speicherung von **produkt** im Filesystem angelegt wurde.
- Wie man sieht, passen alle Tupel in eine Seite mit der Nummer 0
  - Seiten haben in PostgreSQL einheitlich die Kapazität von 8 KB.

# Struktur interner Datensätze

## Das zeilenorientierte Design

---

- **Interner Datensatz (engl.: record)**
  - speichert meistens genau **ein Tupel**, d.h. eine Zeile einer Relation
    - Attributwerte **variabler** (z.B. *varchar(n)*) und **fixer** (z.B. *int*) **Länge**
    - ermöglicht das **Hinzufügen/Entfernen** von Attributen einer Tabelle (in SQL via *ALTER TABLE*)
    - kodiert NULL-Werte
- **Zeilenorientiertes Design:**
  - Lese- oder Schreiboperationen beziehen sich auf ganze Zeilen
  - Attribute einer Tabellenzeile bilden einen **zusammenhängenden Bereich** im Speicher (ob im Hauptspeicher oder auf Festplatte)
- Alternative dazu: **Spaltenorientiertes Design**
  - Ermöglicht effizienten Zugriff auf einzelne Attribute

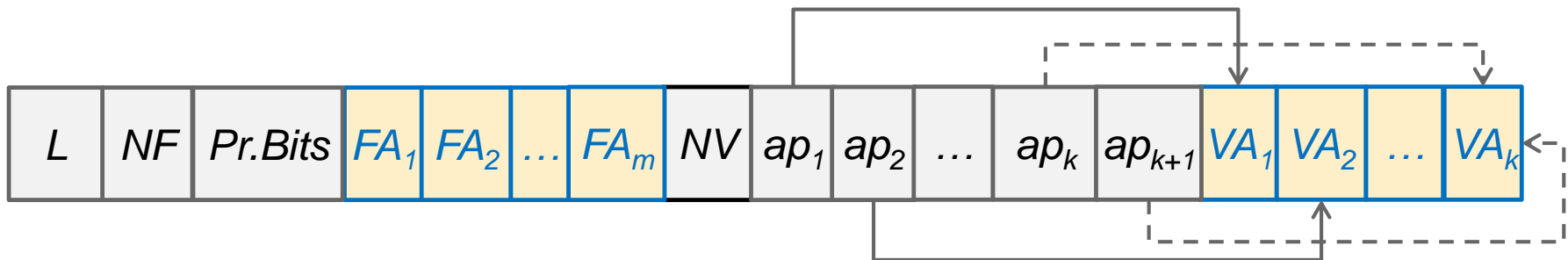
=> Wird in der Vorlesung **In-Memory Datenbanken** behandelt

# Struktur interner Datensätze

## Aufbau von Datensätzen

**Schematische Struktur** interner Datensätze um ein Tupel zu speichern:

- **Nutzdaten** sind Attributwerte in der gegebenen Tabellenzeile:
  - **$VA_i$** : Attributwerte des Tupels mit variabler Länge (varchar, etc.)
  - **$FA_i$** : Attributwerte des Tupels mit fixer Länge (integer, double, etc.)



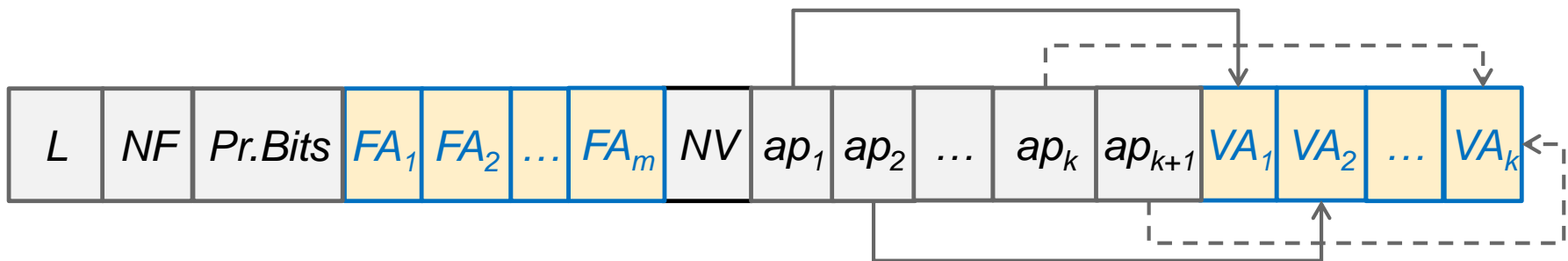
- **Overhead**
  - **$L$** : Länge des Satzes in Bytes
  - **$NF$** : Anzahl der Attributwerte mit fixer Länge ( $m$ )
  - **Präsenzbits**: Wenn Attributwert  $FA_i = NULL$ , wird  $Bit_i = 0$  gesetzt
  - **$NV$** : Anzahl der Attributwerte mit variabler Länge ( $k$ )
  - **$ap_i$** : Pointer zum Attributwert  $VA_i$  bzw. Satzende
    - enthält Offset des ersten Byte von  $VA_i$  relativ zum **Satzbeginn**
    - Wenn  $VA_i = NULL$ , gilt  $ap_i = 0$

# Struktur interner Datensätze

## Aufbau von Datensätzen

### Hinweise:

- Struktur ist abhängig vom DBMS, es gibt **keine Standards**.
- Beim Lesen eines Tupels, kann das System anhand von  $L$  den benötigten Speicher vorsehen.
- Präsenzbits und Attributpointer kodieren NULL-Werte unabhängig von spezifischen Datentypen (keine dafür reservierten Werte notwendig).

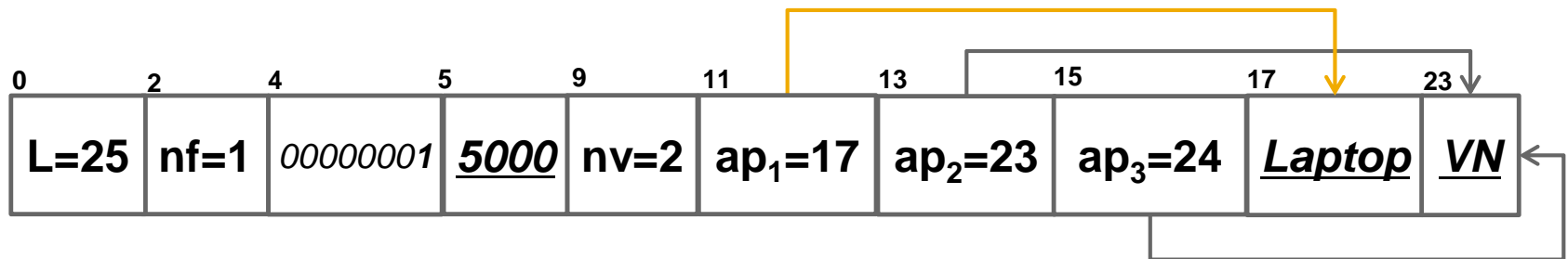


- Dank  $ap_i$  keine „Abschlusszeichen“ etc. am Ende von Zeichenketten (VARCHAR, TEXT, usw. in SQL) notwendig.
- Abschätzung des Overheads pro Tupel:
  - $L, NF, NV, ap_i$  können als Integer Werte repräsentiert sein
  - $m$  Präsenzbits für  $m$  Attributwerte fixer Länge

# Speicherung physischer Datensätze

## Am Beispiel

- Ein Tupel der Relation **produkt**: {5000, „Laptop“, „VN“}
- Der interne Datensatz zur Speicherung von insg. 12 Bytes Nutzdaten:



Beachte im Beispiel:

- Die Felder  $L$ ,  $nf$ ,  $nv$  und  $ap_i$  sind als 2-Byte-Integer gespeichert
  - Bspw. belegt  $L$  zwei Bytes (Byte #0 und Byte #1)
- Pro CHAR (in  $PName$  und  $Land$ ) wird je ein Byte belegt
- Offsets in  $ap_i$ : Position/Entfernung relativ zu Satzbeginn (Byte #0)
- 8 Präsenzbits (1 Byte) vorgesehen.



# Indizierung der Daten

---

- Datensätze sind in Festplattenblöcken i.d.R. ohne Sortierung gespeichert. Ohne eine Indizierung müsste bei jeder Anfrage ein sog. *Full-Table-Scan* erfolgen, um die Zeilen (gemäß *WHERE*-Prädikaten) zu finden.
- Durch Indizierung sind einzelne Datensätze auf dem physischen Datenträger **direkt erreichbar**.
- Indexe können über **einzelne Attribute und auch Attributsequenzen** gebildet werden, in SQL mittels *CREATE INDEX*.
- Einige RDBMS erzeugen Indexe über **Primärschlüssel automatisch**.
- Grundsätzlich können **mehrere Indexe pro Tabelle/DB-Objekt** gebildet werden.
- Indexe sind selbst **persistierte Datenbankobjekte**. Sie müssen in Transaktionen dementsprechend behandelt werden (z.B. gesperrt).

# Indizierung der Daten

## Beispiele

---

- Erzeugen eines Index über *mitarbeiter*, mit Attributwerten aus *Fname* als Suchschlüssel:

*CREATE INDEX mitarbeiter\_fname\_ix ON mitarbeiter (Fname)*

- Index über drei **konkatenierte Attributwerte** als Suchschlüssel:

*CREATE INDEX mitarbeiter\_fname\_ix ON mitarbeiter (Fname, Job, Gehalt)*

- Dies kann die Suche (GROUP BY) beschleunigen. Gegenbeispiel:  
„Suche alle Mitarbeiter mit Gehalt zwischen 22000 und 50000“ ☹ ☹ ☹
- Die richtige Verwendung von Indexen hat großen Einfluss auf die Systemperformanz. Wer soll aber Indexe erzeugen und pflegen?  
Anwendungsentwickler kennt eigene Queries bestens, der DB-Admin das jeweilige System...

# Indexstrukturen

## B<sup>+</sup>-Bäume

---

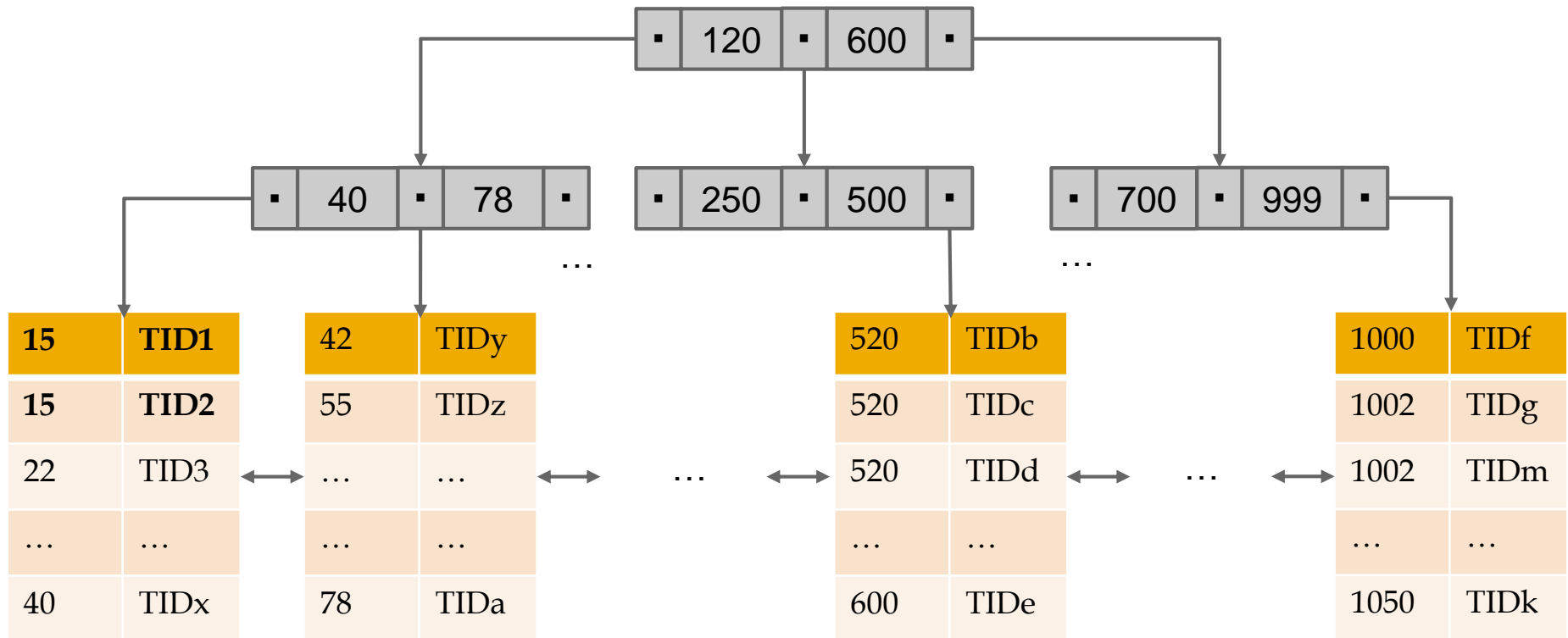
- **Suchbäume** ermöglichen eine effiziente Suche nach Datensätzen.  
Beachte: Suchschlüssel im Baum ist nicht immer Schlüssel der Relation.
- **B<sup>+</sup>-Bäume** sind balanciert, d.h. Pfadlänge zu den Blättern ist konstant.
- **Interne Baumknoten** speichern  $k$  **Suchschlüssel** und  $k+1$  **Verweise** auf *Kinderknoten* bzw. *Blätter*.
- **Blattknoten** speichern sortierte Suchschlüssel und je Schlüssel mind. einen Verweis zu den Datensätzen. Die Verweise referenzieren die Tupel eindeutig (s. TID).
- **Blattknoten** sind i.d.R. **verkettet**, um **Intervallsuchen** zu beschleunigen.
- Größe von  $k$  wird so gewählt, dass der Baum in möglichst wenig Blöcken persistent gespeichert werden kann.

# Indexstrukturen

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

Ein B<sup>+</sup>-Baum mit *zwei* Suchschlüsseln und *drei* Verweisen in jedem Knoten.

- Baum indiziert Relation über ein Nicht-Schlüsselattribut (s. 15, 520, 1002)
- Speicherort des Tupels  $i$  wird mittels Tupel-ID ( $TID_i$ ) referenziert.



# Indexstrukturen

## Praktische Hinweise

---

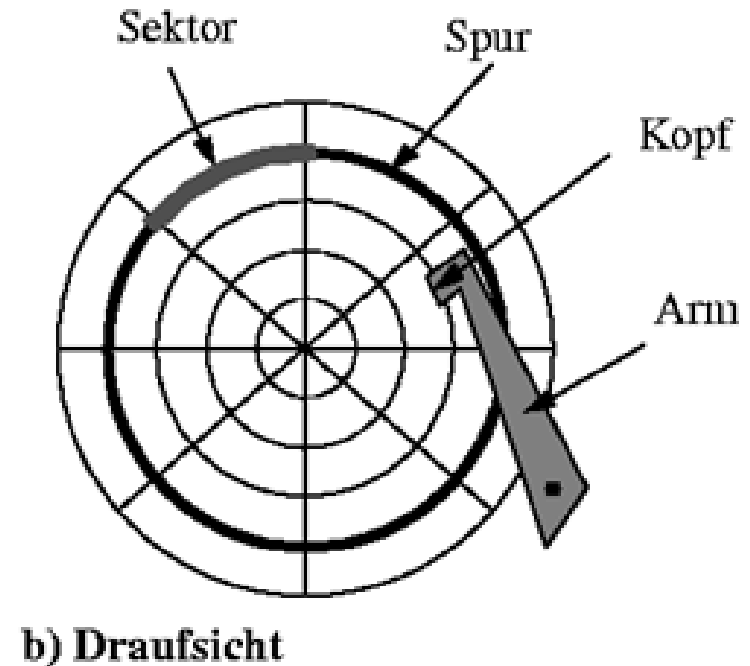
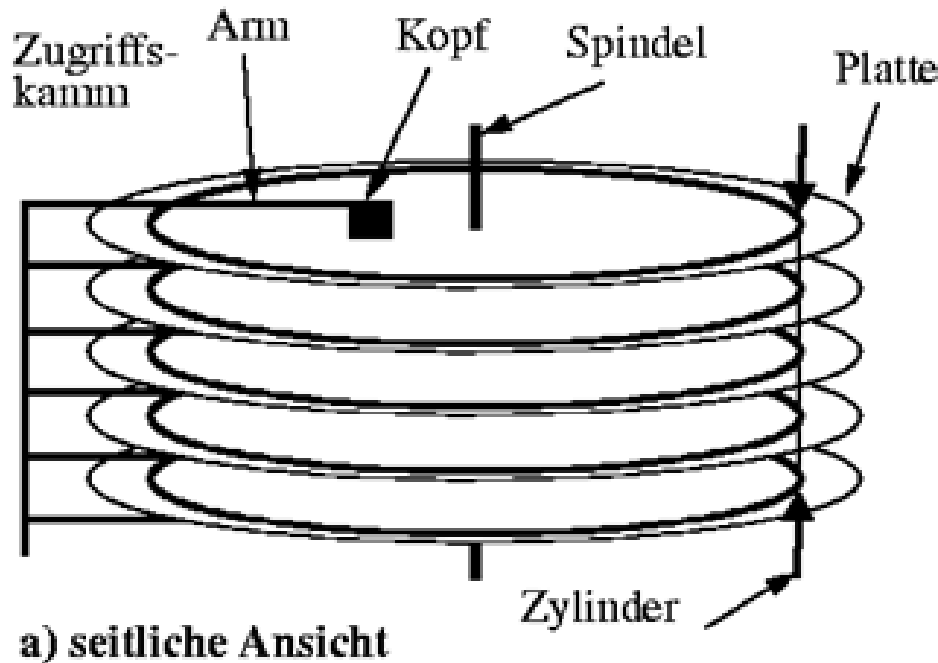
- Suche kann generell beschleunigt werden, wenn der Baum komplett im **Hauptspeicher** vorliegt.
- **Suchanfragen** nach indizierten Attributen können ausgehend vom Baum beantwortet werden, d.h. ganz **ohne Zugriffe auf DB-Dateien** („*Fast Full Index Scan*“).
- Eine Variante sieht sogar die **gemeinsame Speicherung** der Relationstupel und des nach **Primärschlüssel** aufgebauten Suchbaums vor. Das kann im System zur Reduktion der Festplattenzugriffe insgesamt führen.  
*CREATE TABLE mitarbeiter ( ... ORGANIZATION INDEX ... );*
- Bspw. bei sog. **Bulk-Loading** von Daten ist es sinnvoll, auf Index-Updates nach jedem Einfügen (INSERT) zu verzichten (*UNUSABLE*). Für Testzwecke kann der Index-Parameter *INVISIBLE* sinnvoll sein.
- Einige nützliche Tipps und Tricks zur Nutzung von Indexen beschreibt M. Winand in „*SQL Performance Explained*“.

---

## 4. Hintergrundspeicher in RDBMS

# Funktionsweise von Magnetplattenspeichern (a.k.a. Festplatten)

- Mehrere gleichförmig rotierende Speicherplatten
- **Pro Plattenoberfläche** ein beweglicher Lese-/Schreibkopf
- Jede Plattenoberfläche ist eingeteilt in Spuren (in 3D gesehen: Zylinder)
- Die Spuren sind als Sektoren (**Blöcke**) fester Größe formatiert
- Ein Block ist die kleinste Lese-/Schreibeeinheit der Platte (i.d.R. **1-8 KB**)



Bildquelle: Kemper, Eickler: Datenbanksysteme

# Funktionsweise von Magnetplattenspeichern (a.k.a. Festplatten)

---

- Mehrere gleichförmig rotierende Speicherplatten
- **Pro Plattenoberfläche** ein beweglicher Lese-/Schreibkopf
- Jede Plattenoberfläche eingeteilt in Spuren (in 3D gesehen: Zylinder)
- Die Spuren sind als Sektoren (**Blöcke**) fester Größe formatiert
- Ein Block ist die kleinste Lese-/Schreibeeinheit der Platte (i.d.R. **1-8 KB**)



Bildquelle: myheimat.de

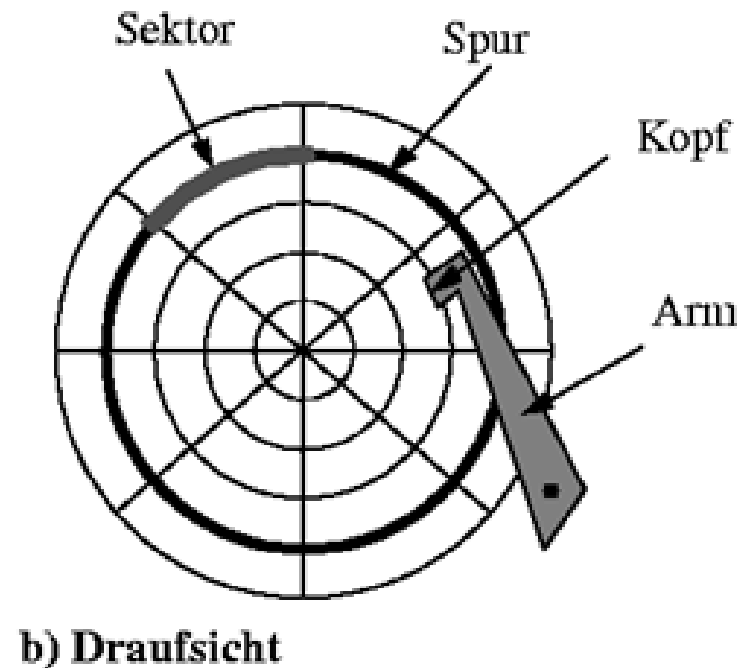
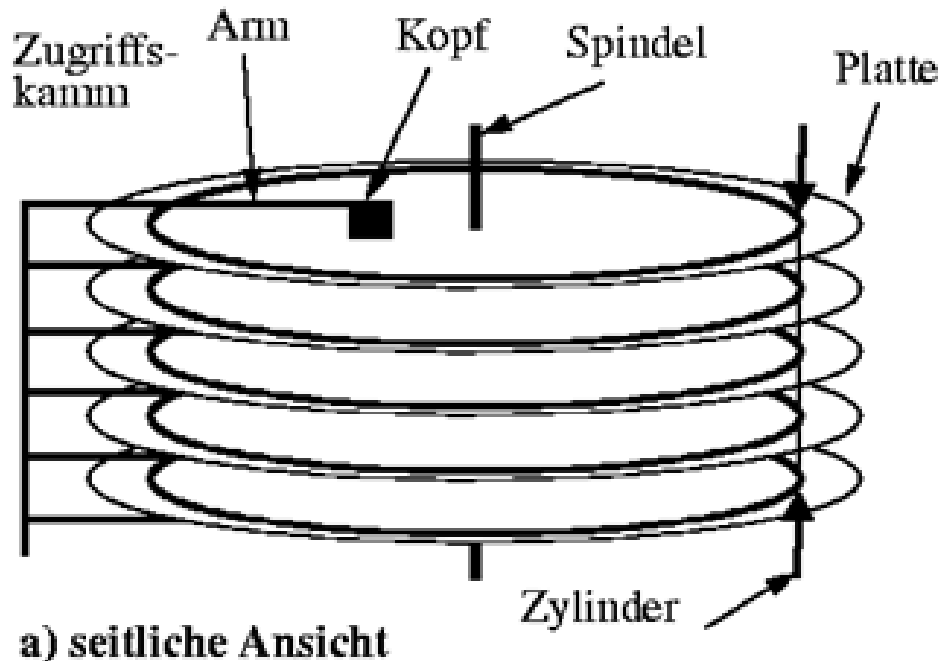


Bildquelle: cosmiq.de



# Adressierung der Daten auf der Festplatte

- (Alte) Methode der **Zylinder/Kopf/Sektor-Adressierung**
  - Adresse enthält Zylindernummer, Kopfnummer, Sektornummer
- Gängige Methode: **Logical Block Addressing (LBA)**
  - 48 Bit kodieren Blocknummern (von #0 bis # $2^{48}-1$ )
  - Festplatte rechnet LBA-Adressen in physische Positionen um



# Performance der Festplattenzugriffe

- Parameter für die Geschwindigkeit des Zugriffs
  - $t_{\text{seek}}$  : Positionierung des Zugriffsarms (seek time)
  - $t_{\text{rotate}}$  : Umdrehungswartezeit (Latenzzeit)
  - $t_{\text{transfer}}$  : Übertragungszeit von der Platte in den Hauptspeicher
  - $u$ : Übertragungsrate von der Platte in den Hauptspeicher (MB/s)


	1970	1990	2005
$t_{\text{seek}}$	30 ms	12 ms	5 ms
$t_{\text{rotate}}$	18 ms	14 ms	4 ms
$u$	1 MB/s	4 MB/s	50 MB/s

- Berechnung mittlerer Zugriffsgeschwindigkeit für Datenmenge  $m$

$$t = t_{\text{seek}} + 1/2 * t_{\text{rotate}} + t_{\text{transfer}} = t_{\text{seek}} + 1/2 * t_{\text{rotate}} + m/u$$

# Performance der Festplattenzugriffe

- Dauer **sequentieller Zugriffe** (chained IO) und Dauer **wahlfreier Zugriffe** (random IO) auf 1000 Blöcke á 4 KB (~ 4 MB):

	1970	2005	Verbesserung
sequentiell	4.039 ms	87 ms	~ 98%
wahlfrei	43.000 ms	7.080 ms	~ 84%
Verhältnis	1:10	1:80 	

- **Sequentieller Zugriff** ist bei großen Datenmengen **deutlich schneller** als wahlfreier Zugriff auf die gleichen Daten.
- Da die Übertragungsrate („Elektronik“) stärker verbessert wird, als die Positionierungs- und Umdrehungswartezeit („Mechanik“), werden sequentielle Zugriffe immer schneller im Vgl. zu wahlfreien Zugriffen.

# Entwicklung von Hintergrundspeichern

- Entwicklung von magnetischen Festplatten und **Solid State Drives (SSD)** anhand von drei Parametern:

<b>Merkmal</b>	<b>Kapazität</b>	<b>Latenz</b>	<b>Bandbreite</b>
1983	30 MB	48.3 ms	0.6 MB/s
1994	4.3 GB	12.7 ms	9 MB/s
2003	73.4 GB	5.7 ms	86 MB/s
2009	2 TB	5.1 ms	95 MB/s
2010 SSD	500 GB	read 65 $\mu$ s write 85 $\mu$ s	read 250 MB/s write 170 MB/s

- Recherchieren Sie, welche Merkmale die **heute (2018) besten** HDD und SSD jeweils aufweisen!!!

# Die Festplatte als Flaschenhals im Gesamtsystem „Zugriffslücke“

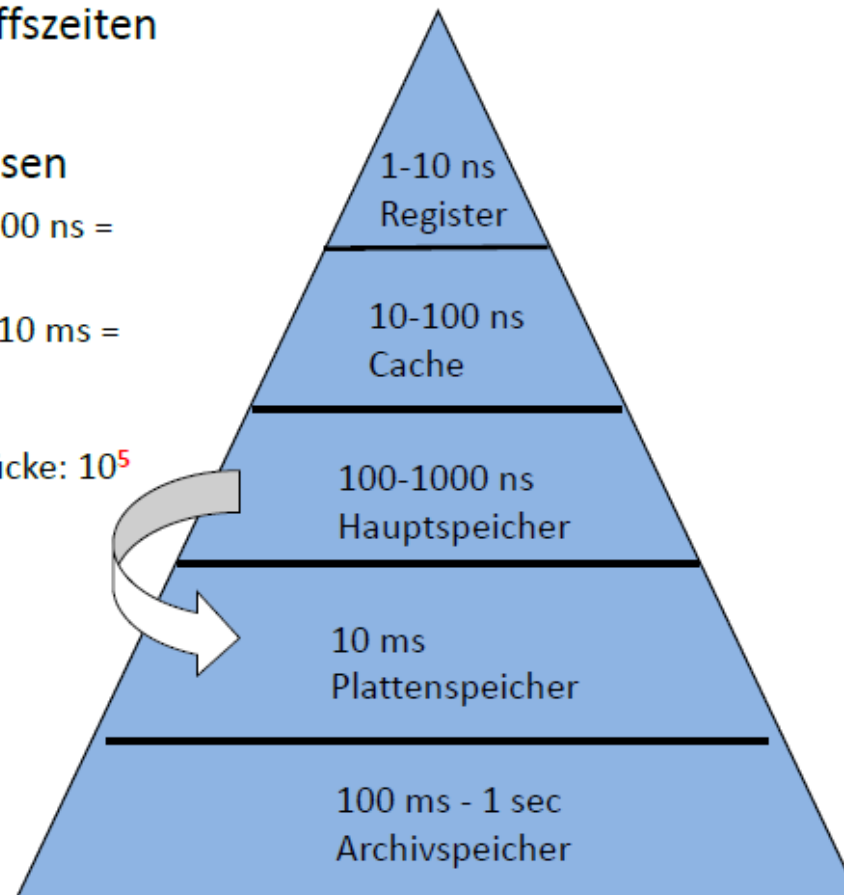
- Die hier illustrierte sog. **Zugriffslücke** motiviert die **Minimierung** der Anzahl von I/O-Operationen (HDD- oder SSD-Zugriffen) als ein wesentliches Design- und Erfolgskriterium für DBMS.

## Größenordnung Zugriffszeiten

### Beispiel: 100 Seiten lesen

- Hauptspeicher:  $100 \times 100 \text{ ns} = 10.000 \text{ ns} = 0,01 \text{ ms}$
- Plattenspeicher:  $100 \times 10 \text{ ms} = 1.000 \text{ ms} = 1 \text{ s}$

Zugriffslücke:  $10^5$

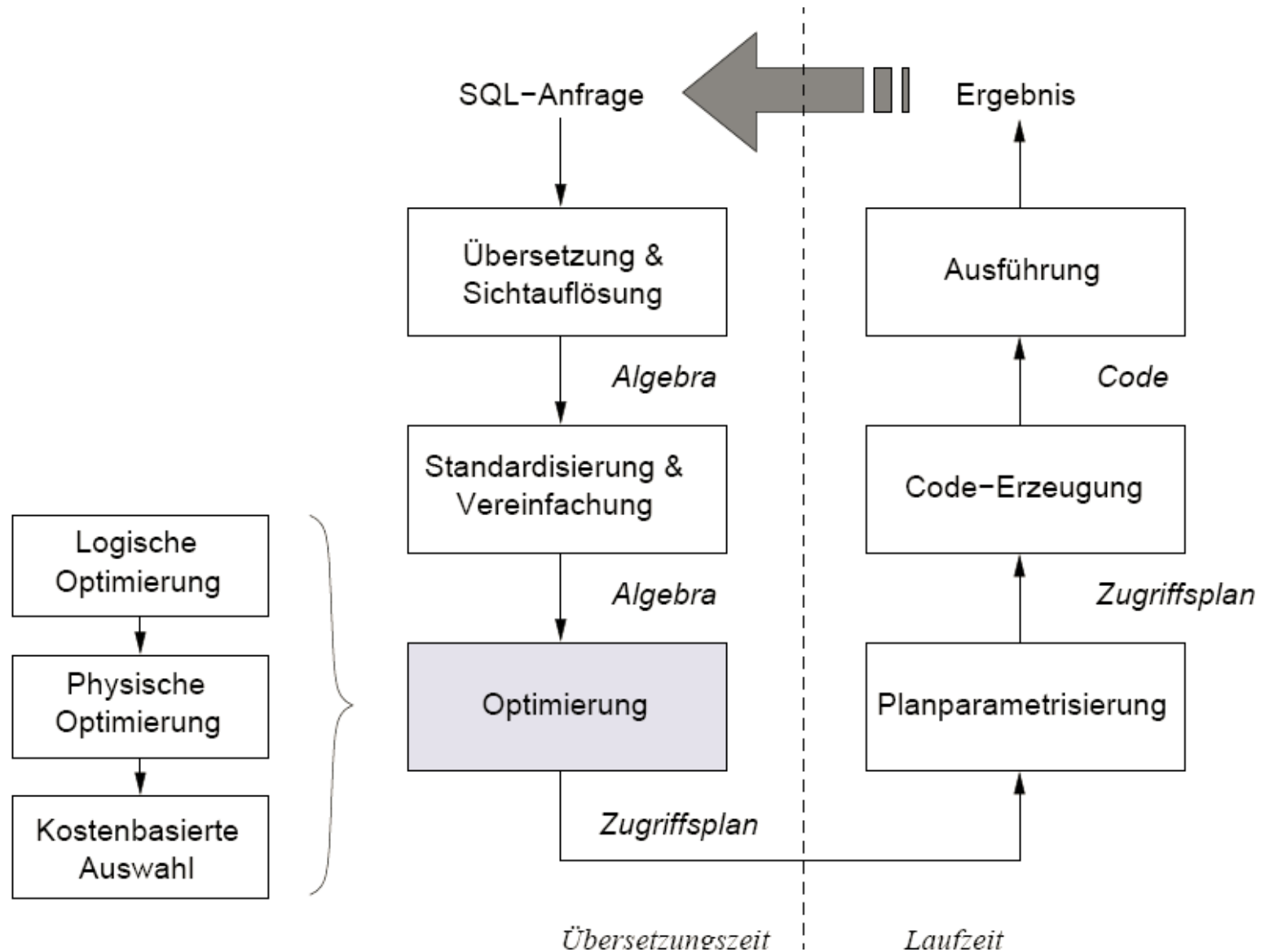


---

## 5. Anfragenverarbeitung in RDBMS

# Prozess der optimierten Anfragenverarbeitung

## Übersicht



# Anfragenverarbeitung


## Schritte zur Übersetzungszeit

---

- **Syntaxprüfung** der Anfrage mit Hilfe eines SQL-Parsers
- **Zugriffskontrolle:** Ist der angemeldete Benutzer berechtigt, die Anfrage durchzuführen, d.h. betroffene DB-Objekte erzeugen/lesen/verändern?
- **Prüfung und Durchsetzung von Konsistenzbedingungen** gemäß statischer Datenstruktur (Tabellen-, bzw. Sichtdefinitionen)
- **Vereinfachung** durch Auflösung von Sichten, Synonymen, usw.
- **Übersetzung** der SQL-Anfrage in Folgen von algebraischen Operatoren
- **Optimierung** dieser Operationsfolgen
- Erzeugung eines **Ausführungsplans** bzw. Zugriffsplans bestehend aus physischen Operatoren, die mit internen, d.h. physischen Objekten arbeiten.



## ■ Logische Optimierung

- Ziel: **Minimierung des Zeit- und Speicheraufwandes** bei der Suche, Sortierung und Bewegung der von der Anfrage betroffenen Daten (insb. Reduzierung der Anzahl Festplattenzugriffe) 
- Optimierer nutzt algebraische Äquivalenz-Regeln aus und erstellt einen **Ausführungsplan** zu der Anfrage.
- Eine **Faustregel der Optimierung**:
  - **Selektionen und Projektionen möglichst früh,**
  - **Verbund-Operationen (JOIN) möglichst spät durchführen**

Warum ist diese Faustregel sinnvoll?

## ■ Beispiel anhand Tabellen von ACME

```
CREATE VIEW best_deals_view AS  
SELECT p.PName AS Produktname, p.ANr AS Artikelnummer, a.Anzahl  
AS Menge, a.StPreis AS Stueckpreis, a.Datum AS Datum  
FROM produkt p INNER JOIN absatz a ON p.ANr = a.ANr  
WHERE a.Anzahl > 1000;  
...
```

```
SELECT Produktname, Artikelnummer, Menge  
FROM best_deals_view;
```

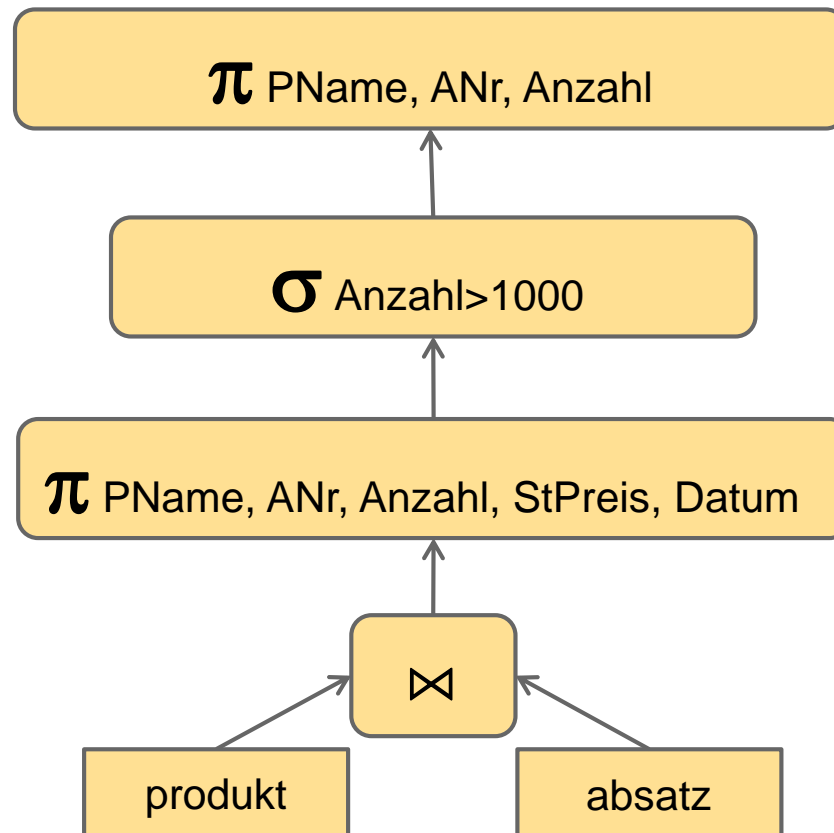


Produktname	Artikelnummer	Menge
Laptop	5000	3421
Tablet	6000	4532

# Logische Optimierung von Ausführungsplänen

Ein möglicher **Ausführungsplan** (Operatorbaum) zu der Anfrage:

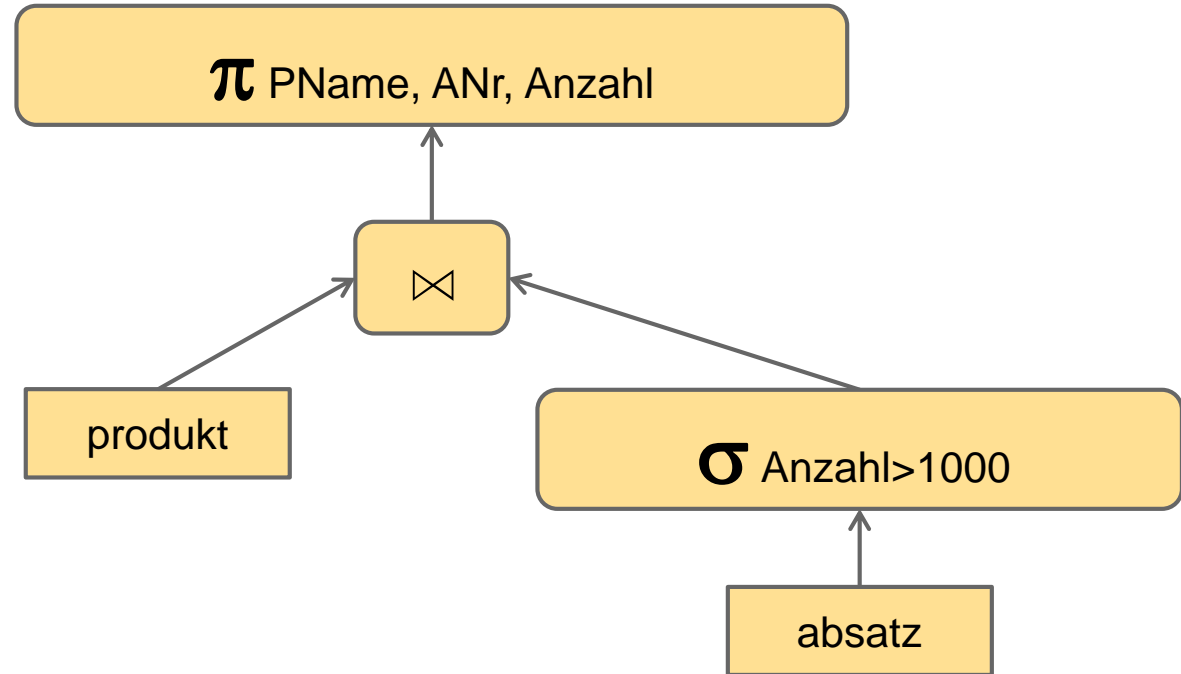
- Die Ausführung beginnt mit dem **Join** unten und endet mit der Ausgabe der Attributwerte gemäß der **Projektion** oben 



# Logische Optimierung von Ausführungsplänen

Ein anderer, kostengünstigerer Plan mit demselben Ergebnis

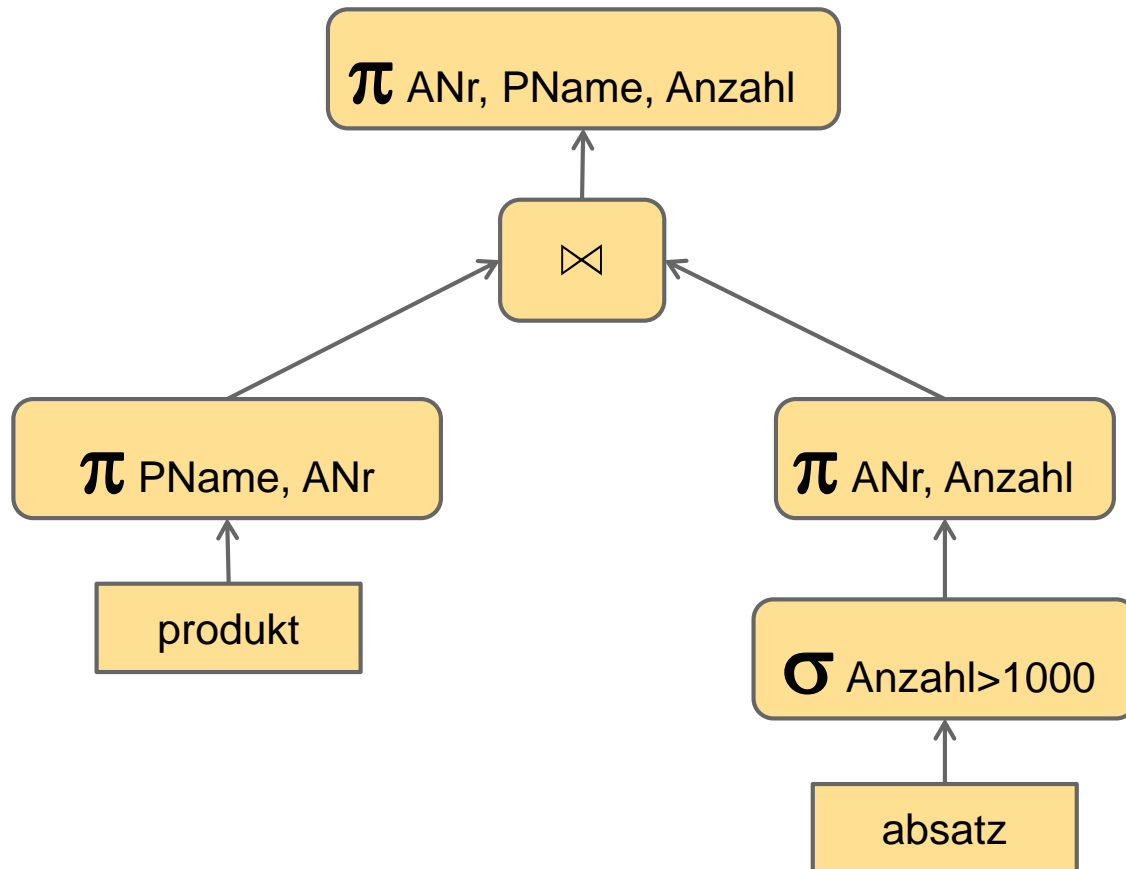
- Join wird über **produkt** und nur zwei Zeilen von **absatz** berechnet
- Kann zur **Reduzierung des Speicherplatzes und der Anzahl Plattenzugriffe** insgesamt führen



# Logische Optimierung von Ausführungsplänen

Ein noch besserer Plan (liefert dasselbe Ergebnis wie die vorherigen):

- Nur die **relevanten Spalten von absatz und produkt** werden als Input der Verbundbildung berücksichtigt
- Führt bspw. zur besseren Ausnutzung von CPU-Cache und RAM



## Physische Optimierung

- **Ziel:** Kostenoptimierte Auswahl von Verfahren um den bereits logisch optimierten abstrakten Ausführungsplan am physischen Datenbestand effizient auszuführen.
- DBMS bietet verschiedene Algorithmen und Strukturen an, um
  - im physischen Datenbestand zu **navigieren**,
  - physische Daten zu **adressieren**, oder
  - **algebraische Operatoren** (Join, Selektion, Projektion, usw.) zu realisieren.
- Je nach **Datenbankzustand, Größe der Tabellen, Zugriffstatistiken** wählt das System das am besten erscheinende Verfahren aus
  - Quelle obiger Informationen ist das **Katalogsystem**

# Physische Optimierung von Ausführungsplänen

---

Optionen bei der physischen Optimierung einiger Operatoren:

- **Selektion** (gemäß Prädikate ... WHERE ... )
  - Relationen-Scan: Datensätze werden **einzeln** auf Erfüllung der Prädikate hin untersucht („Brute-Force“)
  - Index-Scan: Suche erfolgt mittels Primär-/Sekundär-**Index**
  
- **Projektion** (SELECT a, b, c... FROM ...)
  - Relationen-Scan und Extrahierung relevanter Attributwerte
  - Index-Scan falls alle Attributwerte indiziert sind
  
- **Verbund** (... JOIN ... ON ...)
  - Beispiele: **Nested Loops, Hash Join, Merge Join**, etc.

# Anfragenverarbeitung

## Schritte zur Laufzeit

---

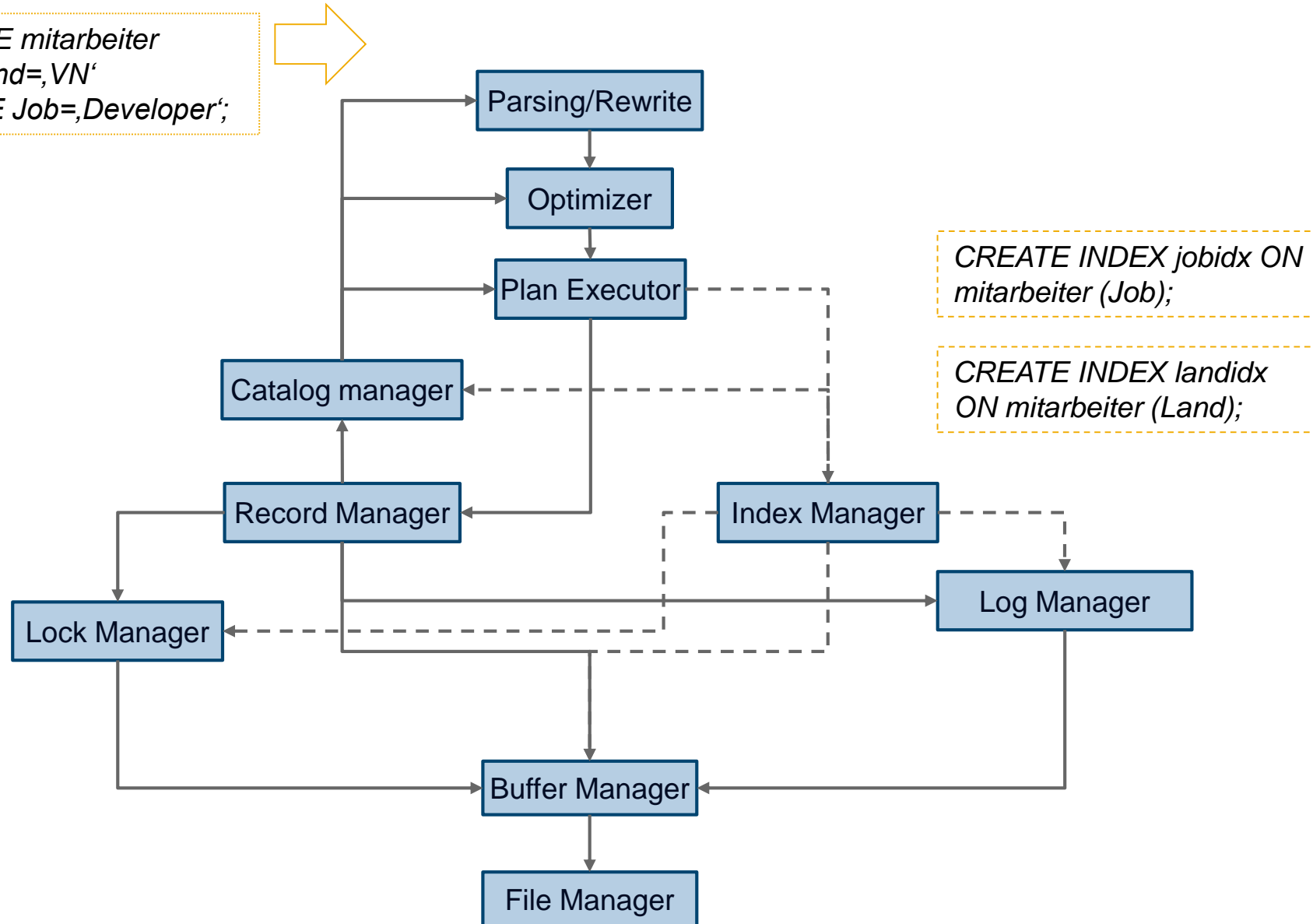
- **Codeerzeugung:** Der optimierte Plan wird durch interne physische Datenobjekte und Aufrufe der ausgewählten Operator-Implementierungen instanziiert.
- **Ausführung** des erzeugten Codes (engl. *plan execution*):
  - Plan Executor ruft ausgehend von Operator-Instanz in der Wurzel des Plans *rekursiv* linke/rechte Sub-Pläne auf, solange bis einzelne Datensätze/ Attributwerte zurückgeliefert werden.
- **Planparametrisierung:** Relevant für **vorkompilierte** Anfragen (engl. *prepared statement*), welche einen Plan definieren. Es werden die jeweiligen Werte eingesetzt, gemäß aktuellem Aufruf der Anwendung.
- Manche Systeme **übersetzen** SQL-Statements direkt in eine „nativ“ ausführbare Sprache wie C (Low-Level Virtual Machine, LLVM) und können dadurch Laufzeit-Performanz verbessern.



# Anfragenverarbeitung

## Interaktionen der Komponenten eines RDBMS

*UPDATE mitarbeiter  
SET Land=,VN'  
WHERE Job=,Developer';*



---

## 6. Pufferverwaltung in RDBMS

# Die Festplatte als Flaschenhals im Gesamtsystem „Zugriffslücke“

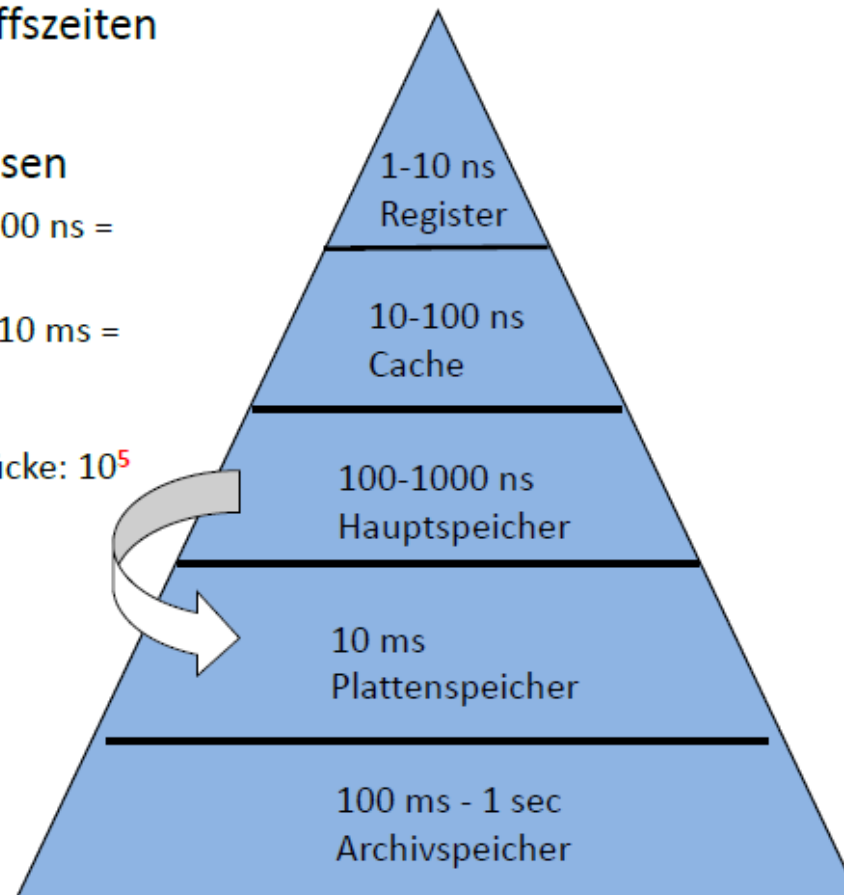
- Die hier illustrierte sog. **Zugriffslücke** motiviert die **Minimierung** der Anzahl von I/O-Operationen (HDD oder SSD Zugriffen) als ein wesentliches Design- und Erfolgskriterium für DBMS.

## Größenordnung Zugriffszeiten

### Beispiel: 100 Seiten lesen

- Hauptspeicher:  $100 \times 100 \text{ ns} = 10.000 \text{ ns} = 0,01 \text{ ms}$
- Plattenspeicher:  $100 \times 10 \text{ ms} = 1.000 \text{ ms} = 1 \text{ s}$

Zugriffslücke:  $10^5$



# Pufferverwaltung

## Aufgaben

---

Wesentliche Aufgaben der **Pufferverwaltung** (engl. buffer manager):

- Anstoß und Überwachung von **Block/Seiten-Transfer** zwischen Hintergrund- und Hauptspeicher
  - **Problem:** *Welche Blöcke sollten wann in RAM geladen werden, welche ggfs. verdrängt, d.h. überschrieben?*
- **Minimierung von Transfer-Wartezeiten** durch geeignete Strategien gemäß observierten Zugriffsmustern auf die Daten
- Gute Wahl der Strategie ist entscheidend mit Blick auf die Performance des Datenbanksystems (als Folge der „**Zugriffslücke**“).
- DBMS-Pufferverwaltung muss **unabhängig vom OS** realisiert werden, um die „Doppel-Pufferung“ von Seiten in beiden Speicherbereichen und evtl. daraus resultierende DB-Inkonsistenzen zu vermeiden.

# Pufferverwaltung

## Organisation des Puffers

---

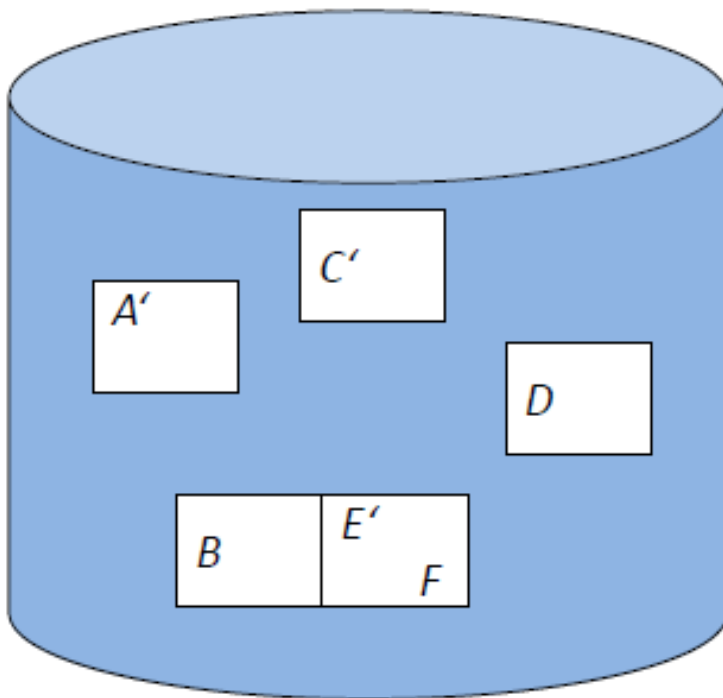
- DBMS verfügt über einen **Puffer** im eigenen **Hauptspeicherbereich** um die für aktuell laufende Datenbankoperationen benötigten Daten schneller zum Prozessor zu bringen.
  - Der Puffer speichert deutlich weniger Daten als bspw. die Festplatte.
- **Alle Lese- und Schreib-Vorgänge** von oder auf Seiten erfolgen ausschließlich via Puffer.
  - Warum nicht Daten direkt auf Platte schreiben?
- Systempuffer ist in **Seitenrahmen** gleicher Größe aufgeteilt
  - Ein Rahmen nimmt einen Block der Festplatte (DB-Datei) auf
  - Pro Seite werden Metadaten, wie **Speicherort** im Puffer, **Zeitstempel** des letzten Zugriffs, Angabe ob Seite **noch in Benutzung** ist („pinned“), etc. gespeichert.
- Wenn der Puffer voll ist, werden „überzählige“ Seiten auf die Platte ausgelagert und Seitenrahmen anschließend überschrieben.

# Pufferverwaltung

## Beispiel

- Seiten im Puffer und Blöcke der Festplatte
- A, B ... F sind einzelne Datensätze, bspw. B' ist die neue Version von B
  - Beachte die unterschiedlichen Versionen auf Festplatte und im Hauptspeicher!

Datenbank auf dem Hintergrundspeicher



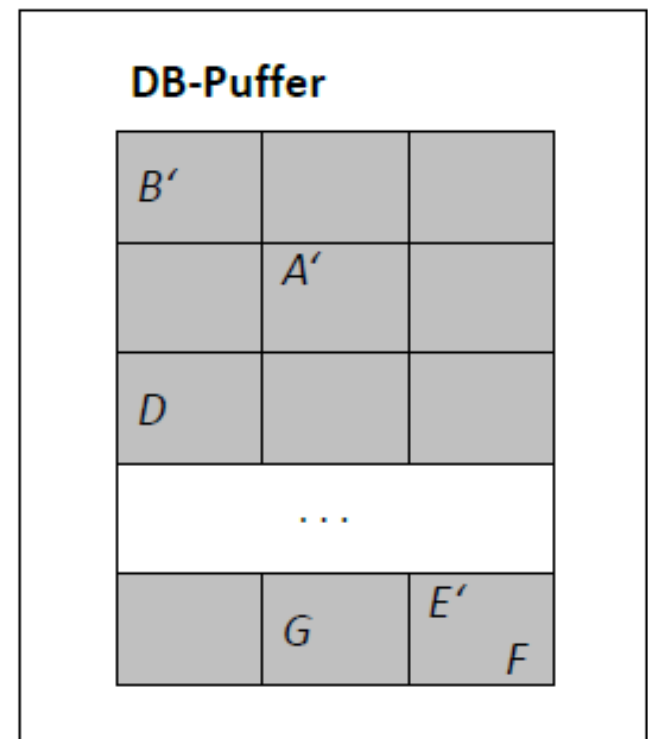
Einlagerung



Auslagerung



Hauptspeicher



Der **Ablauf eines Zugriffs** auf eine Seite z.B. während einer Transaktion:

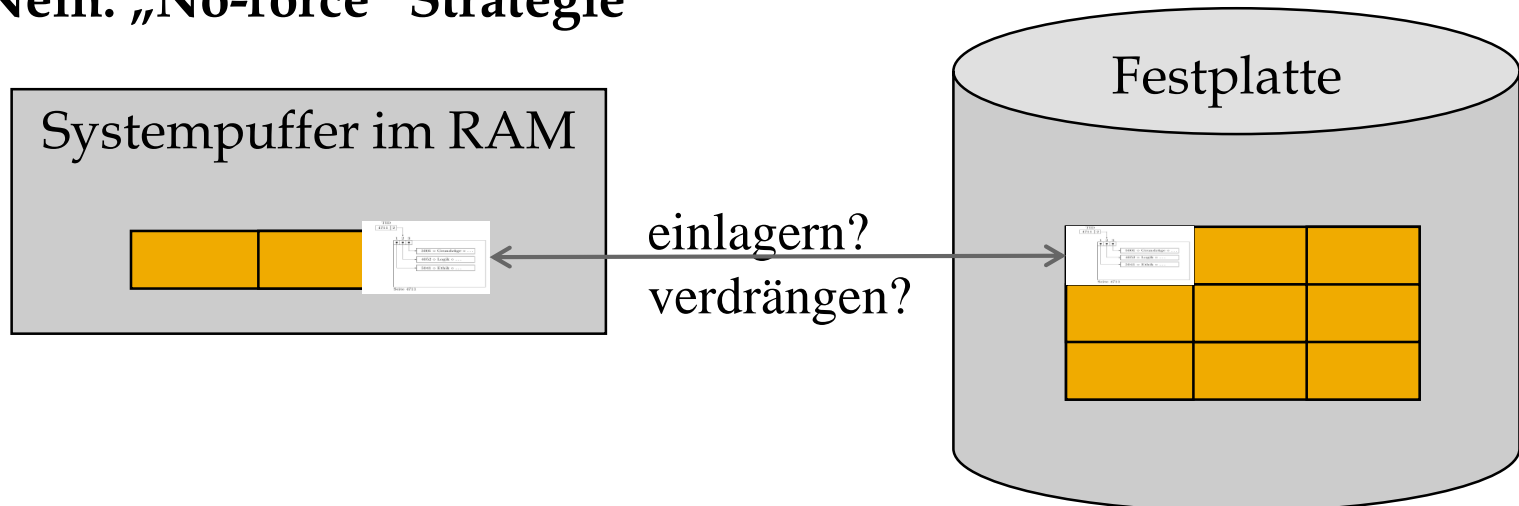
- Record Manager fordert bei Pufferverwaltung eine referenzierte Seite an (via Seitennummer)
- **Angeforderte Seite ist im Puffer:** Puffermanager teilt Adresse der Seite im Puffer mit, und der Inhalt kann gelesen (gepinnt) werden.
- **Angeforderte Seite ist nicht im Puffer (page fault):**
  - Physische Seitenreferenz durch Pufferverwaltung an Betriebssystem
  - Bei vollem Puffer wird eine Seite aus dem Puffer entfernt, bspw:
    - Die seit längstem unbenutzte Seite (*Least Recently Used, LRU*)
    - Die am seltensten benutzte Seite (*Not Frequently Used, NFU*)
    - Falls die zu verdrängende Seite seit dem Einlagern geändert wurde, wird die Seite *vorm* Überschreiben des Pufferbereichs auf die Festplatte geschrieben.

# Pufferverwaltung

## Strategiemerkmale

### Strategien für Pufferverwaltung:

- Können Seiten **nicht abgeschlossener Transaktionen verdrängt** (und dadurch zumindest temporär persistiert) werden?
  - Ja: „Steal“ Strategie
  - Nein: „No-steal“ Strategie
- Werden Seiten **erfolgreicher Transaktionen vor Transaktionsende** (Commit) **garantiert persistiert**?
  - Ja: „Force“ Strategie
  - Nein: „No-force“ Strategie





# Pufferverwaltung

## Auswirkungen auf Logging und Recovery

- **Strategien** werden **kombiniert** eingesetzt, s. Tabelle

	Force	No-Force
No-Steal	<ul style="list-style-type: none"><li>● kein Redo</li><li>● kein Undo</li></ul>	<ul style="list-style-type: none"><li>● Redo</li><li>● kein Undo</li></ul>
Steal	<ul style="list-style-type: none"><li>● kein Redo</li><li>● Undo</li></ul>	<ul style="list-style-type: none"><li>● <b>Redo</b></li><li>● <b>Undo</b></li></ul>

- Beim Logging und Recovery der Daten von Transaktionen (TA) muss die gegebene Kombination berücksichtigt werden:
  - **Beispiel:** Bei **Force und No-Steal**: kein Redo-Log und kein Undo-Log notwendig, da Daten einer „committed“ TA immer auf Platte bzw. Daten laufender TA nur im Hauptspeicher vorhanden sind.
- Verbreitet in DBMS ist die Kombination **Steal und No-Force**

# Womit ist ein DBMS ansonsten beschäftigt?

## Aus Anwendersicht nützliche vs. administrative Tasks

---

- Die Verwaltung der Prozesse und der Daten gemäß strengen ACID-Regeln ist rechenzeit- und speicherplatzintensiv:

