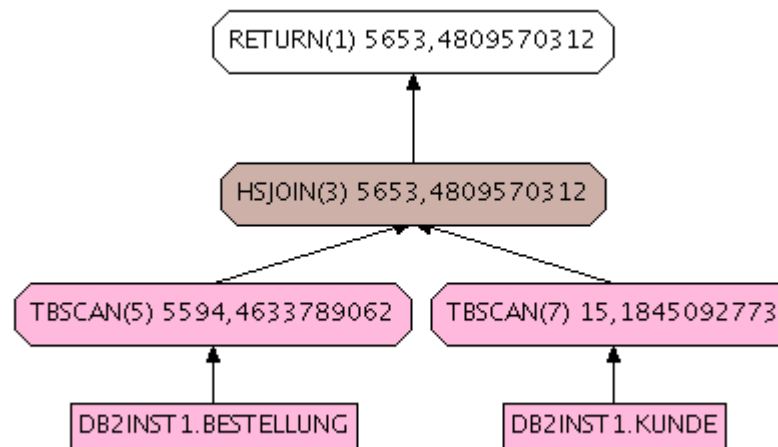




# Datenbanken 1

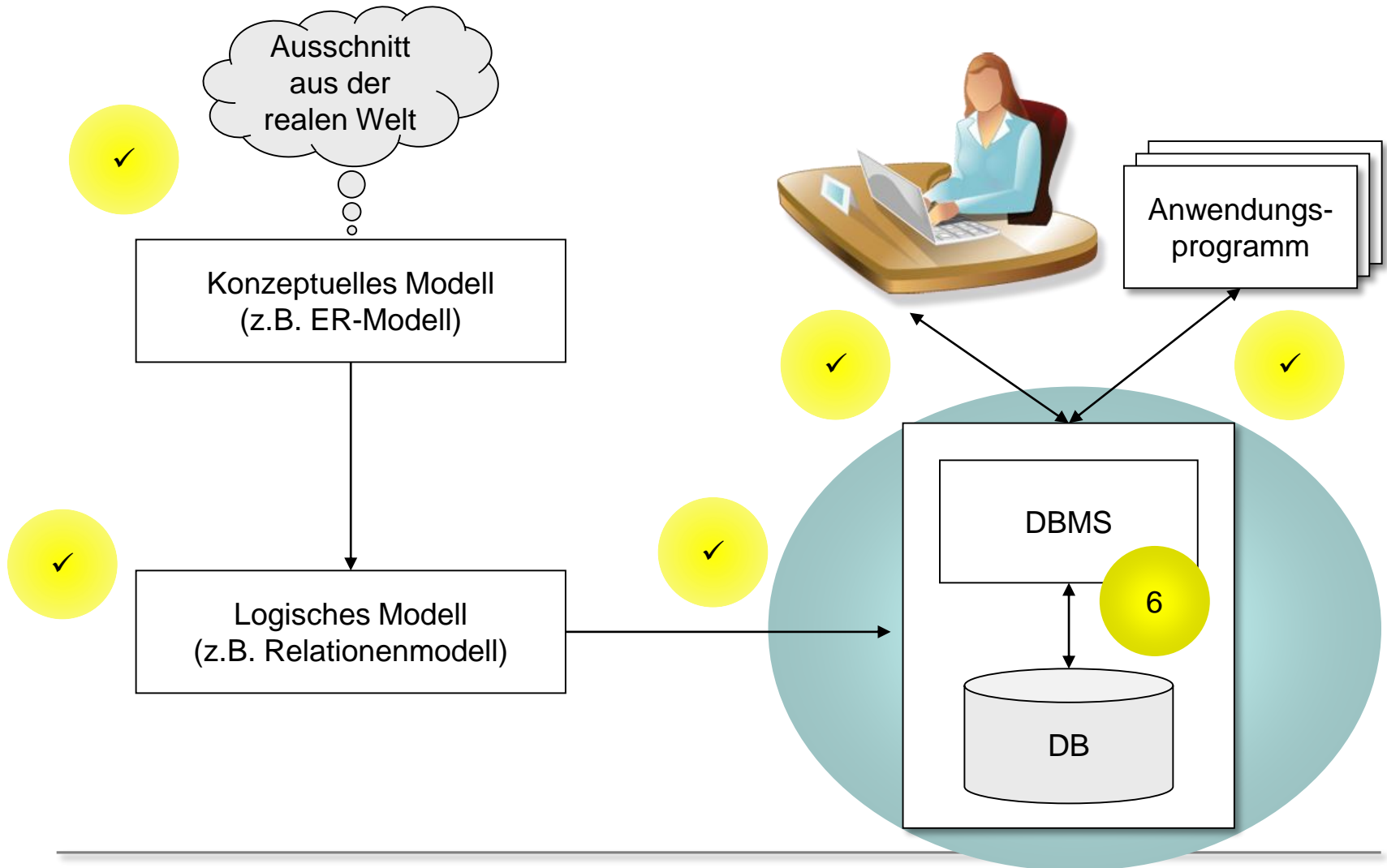
## Kapitel 6:

### – Architektur von Datenbanksystemen –





# Vorlesung Datenbanken 1





# Architektur von Datenbanksystemen

---

## Inhalt des Kapitels

- 3-Ebenen-Architektur von Datenbanken
  - Externe Ebene
  - Konzeptionelle Ebene
  - Interne Ebene
- DBMS-Systemarchitektur
  - Schichtenarchitektur
  - Transaktionsverwaltung und Recovery

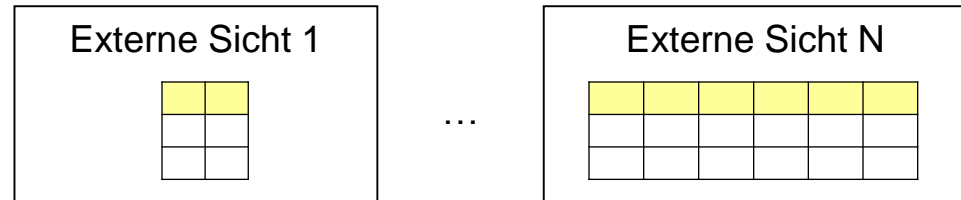
## Lernziele

- Kennen der 3-Ebenen-Architektur von Datenbanken
- Verstehen des Aufbaus von Indexen und deren Bedeutung für die Performance-Optimierung
- Kennen der wichtigsten Schichten eines DBMS und deren Funktion
- Verstehen des Transaktionsbegriffs und Kennen der Auswirkungen unterschiedlicher Isolationslevel
- Kennen und Verstehen verschiedener Fehlerarten und ihrer Behandlung

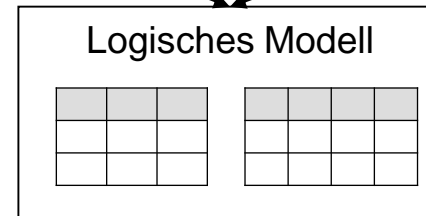


# 3-Ebenen-Architektur

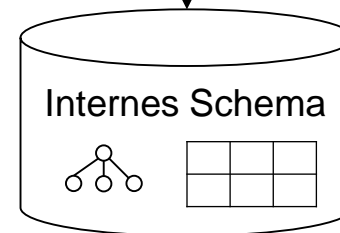
Externe Ebene



Konzeptuelle Ebene



Interne Ebene



- ⇒ **Physische Datenunabhängigkeit:** Änderungen im internen Schema (Dateiorganisation und Zugriffspfade) haben keinen Einfluss auf das logische Modell oder externe Sichten.



# Sichten (Views)

---

## Was sind Sichten?

- „Virtuelle Relationen“, die einen Ausschnitt oder eine Neukombination des Datenbankschemas zeigen
- Keine neuen Tabellen, d.h. Daten sind nicht materialisiert, sondern werden bei jeder Verwendung neu berechnet!

## Wofür werden Sichten eingesetzt?

- Vereinfachung von Anfragen („Makro“)
- Datenschutz (z.B. Ausblenden bestimmter Attribute oder Beschränkung auf bestimmte Datensätze)
- zur (besseren) Abbildung von Generalisierungen

## Wie werden Sichten definiert?

```
CREATE VIEW sicht-name [ schema-deklaration ]  
AS SQL-anfrage
```



# Sichten zur Vereinfachung von Anfragen

PRODUKT:

<u>PRODID</u>	BEZEICHNUNG	PREIS	BESTAND	HERSTID
201	Skyscraper	99.0	12	901
202	Himmelsstürmer	129.0	4	901
203	Rainbow Hopper	45.0	20	902

HERSTELLER:

<u>HERSTID</u>	NAME
901	Flattermann GmbH
902	Dragon.com

Beispiel: Häufig benötigte Aufstellung: Produktbezeichnung + Hersteller

⇒ Sicht (View) definieren

```
CREATE VIEW ProdHerstView AS  
    SELECT p.Bezeichnung, h.Name  
    FROM Produkt p, Hersteller h  
    WHERE p.HerstID = h.HerstID
```

PRODHERSTVIEW:

BEZEICHNUNG	NAME
-------------	------

⇒ View abfragen

```
SELECT *  
FROM ProdHerstView
```

BEZEICHNUNG	NAME
Skyscraper	Flattermann GmbH
Himmelsstürmer	Flattermann GmbH
Rainbow Hopper	Dragon.com



# Sichten für den Datenschutz

- Beispiel: *Jeder Mitarbeiter soll die Namen aller anderen Mitarbeiter, aber nicht deren Personalnr., Geburtsdatum und Gehalt lesen können.*

MITARBEITER:

<u>PNR</u>	NAME	VORNAME	GEBDATUM	GEHALT
111	Lufter	Jens	01.03.1980	3800
112	Schaarschmidt	Ralf	27.05.1975	4500
113	Nowitzky	Jan	03.07.1978	3500

⇒ Erstellung der entsprechenden Sicht

```
CREATE VIEW Personal AS  
    SELECT m.Name, m.Vorname  
    FROM Mitarbeiter m
```

PERSONAL:

NAME	VORNAME
------	---------

⇒ Vergabe der Rechte

```
GRANT SELECT  
ON Personal  
TO PUBLIC
```



# Rechtevergabe in Datenbanksystemen

---

- Konzept: *Zugriffsrechte* (AutorisierungsID, DB-Objekte, Operation)
  - AutorisierungsID ist interne Kennung eines „Datenbankbenutzers“
  - DB-Objekte: Relationen und Sichten
  - DB-Operationen: Lesen, Einfügen, Ändern, Löschen

- **Rechtevergabe in SQL**

```
GRANT <Rechte>  
ON <Objekt>  
TO <BenutzerListe>  
[ WITH GRANT OPTION ]
```

- In <Rechte>-Liste: **ALL** bzw. Langform **ALL PRIVILEGES** oder Liste aus **SELECT, INSERT, UPDATE, DELETE**
- Hinter **TO**: Autorisierungsidentifikatoren (auch **PUBLIC, GROUP**)
- **WITH GRANT OPTION**: Recht auf die Weitergabe von Rechten





# Sichten für den Datenschutz (Forts.)

---

- Beispiel: *Jeder Mitarbeiter soll **seine** Arbeitszeitangaben sehen und neue Arbeitszeitangaben einfügen können (aber nicht löschen und ändern)!*

⇒ Erstellung der entsprechenden Sicht

```
CREATE VIEW MyWorkSchedule AS  
SELECT *  
FROM WorkSchedule  
WHERE Employee = USER
```

⇒ Vergabe der Rechte

```
GRANT SELECT, INSERT  
ON MyWorkSchedule  
TO PUBLIC
```



# Rechtevergabe in Datenbanksystemen (Forts.)

---

- **Rücknahme von Rechten in SQL**

```
REVOKE [ GRANT OPTION FOR ] <Rechte>  
ON <Objekt>  
FROM <BenutzerListe>  
[ RESTRICT | CASCADE ]
```

- **GRANT OPTION FOR:** entzieht das Recht auf Weitergabe der Rechte
- **RESTRICT:** Falls Recht bereits an Dritte weitergegeben: Abbruch von REVOKE
- **CASCADE:** Rücknahme des Rechts mittels REVOKE wird an alle Benutzer propagiert, die es von diesem Benutzer mit GRANT erhalten haben



# Sichten

---

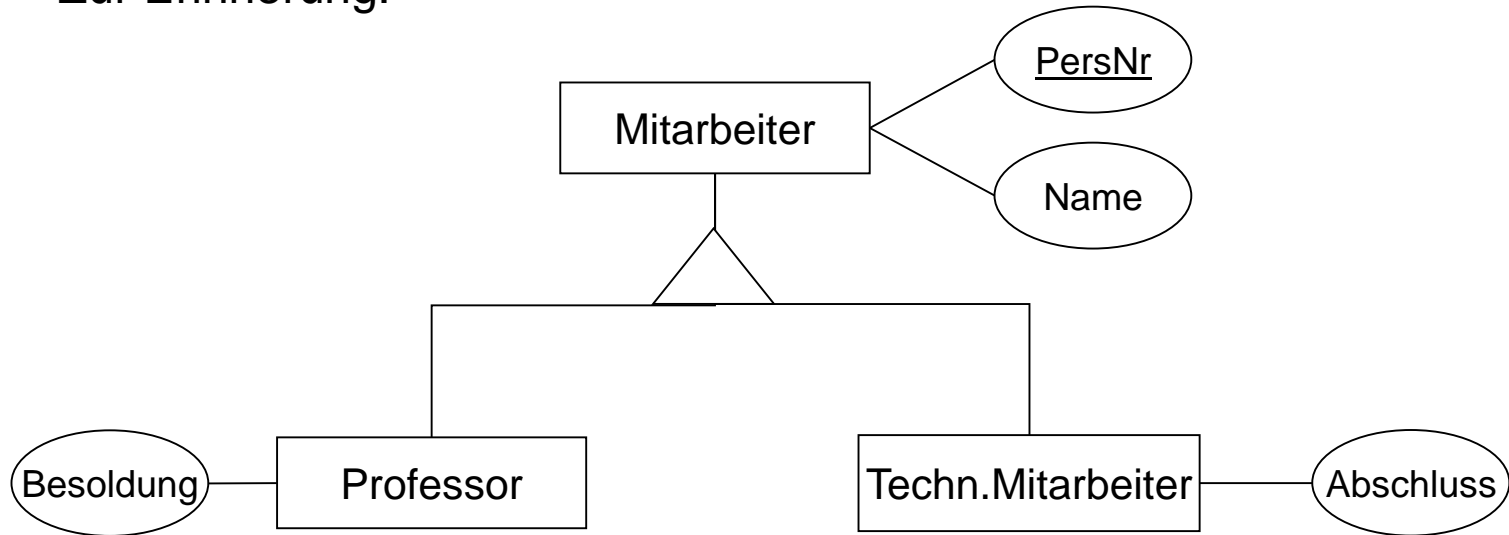
## **Wofür werden Sichten eingesetzt?**

- ✓ Vereinfachung von Anfragen („Makro“)
- ✓ Datenschutz (z.B. Ausblenden bestimmter Attribute oder Beschränkung auf bestimmte Datensätze)
- zur (besseren) Abbildung von Generalisierungen



# Sichten zur Abbildung der Generalisierung

- Zur Erinnerung:



- Vier Varianten der Abbildung auf das relationale Modelle mit verschiedenen Vor- und Nachteilen ...



# Sichten zur Abbildung der Generalisierung

## Variante 1: „Hausklassenmodell“

- Nur für die Spezialisierungen werden Relationenschemata ausgeprägt:

PROFESSOR		
<u>PERSNR</u>	integer	<pk>
BESOLDUNG	character(2)	
NAME	variable character(20)	

TECHNMITARBEITER		
<u>PERSNR</u>	integer	<pk>
ABSCHLUSS	variable character(10)	
NAME	variable character(20)	

⇒ **Generalisierung (Supertyp) als Sicht**

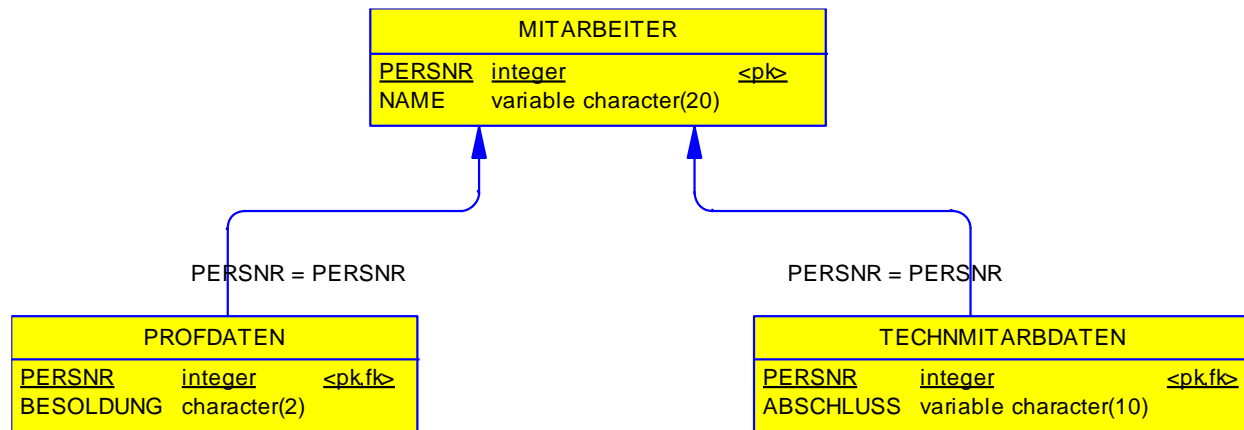
```
CREATE VIEW Mitarbeiter AS  
  (SELECT PersNr, Name  
   FROM Professor)  
  UNION  
  (SELECT PersNr, Name  
   FROM TechnMitarbeiter)
```



# Sichten zur Abbildung der Generalisierung

## Variante 2: „Partitionierungsmodell“

- Sowohl für die Spezialisierungen als auch die Generalisierung werden Relationenschemata ausgeprägt und (nur) der Primärschlüssel der Generalisierung wird übernommen:



## ⇒ Spezialisierung (Subtypen) als Sicht

```
CREATE VIEW Professoren AS
SELECT *
FROM Mitarbeiter m, ProfDaten d
WHERE m.PersNr=d.PersNr
```

```
CREATE VIEW TechnMitarbeiter AS
SELECT *
FROM Mitarbeiter m, TechnMitarbDaten d
WHERE m.PersNr=d.PersNr
```



# Sichten

---

- Sichten können die Benutzung der Datenbank vereinfachen, ohne dabei Redundanz in Kauf nehmen zu müssen!
- Arten von Sichten
  - Projektionssichten
  - Selektionssichten
  - Verbundsichten
  - Vereinigungssichten, Schnittssichten
  - Aggregationssichten
- Nicht vergessen: Sichten verbessern die Performance nicht! (Anfrageergebnis wird bei jeder Anforderung neu berechnet)
- Teilweise in Datenbanksystemen auch unterstützt: Materialisierte Sichten (*materialised views*). Änderungen werden von den Basisrelationen zu den Sichten propagiert.



# Änderungen in Sichten – 1(2)

---

- Bisher: select-Anfragen auf Sichten
- Was ist mit Änderungen (insert, update, delete)?

## Probleme bei Änderungen auf Sichten

- Mögliche Integritätsverletzung in der Basisrelation (z.B. not null bei Einfügen in Projektionssicht)
- Bei Aggregationssichten keine sinnvolle Abbildung auf Basisrelation möglich
- Bei Verbundsichten oft keine Transformation auf Basisrelationen möglich
- ...





# Änderungen in Sichten – 2(2)

---

Sichten sind in **SQL-92** änderbar (*updatable*), wenn sie

- nur genau eine Tabelle (also Basisrelation oder Sicht) verwenden, die ebenfalls änderbar sein muss
- keine Vereinigung oder Schnittbildung enthalten
- kein distinct enthalten (d.h. eine 1:1 Abbildung von Sichttupeln und Basistupeln möglich ist)
- im select-Teil keine Aggregationsfunktionen oder Arithmetikfunktionen enthalten sind
- keine Gruppierung enthalten (group by, having)

In **SQL-99** etwas weniger restriktiv. Einschränkungen aufgehoben für:

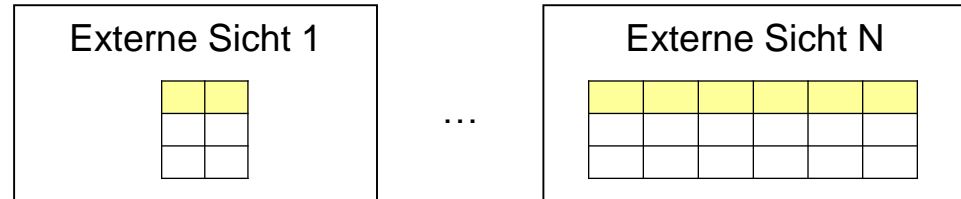
- bestimmter Vereinigungsbildungen und
- insert in Verbundsichten über Primärschlüssel und Fremdschlüssel

*Sichten vs. theoretisch änderbare Sichten vs. in SQL änderbare Sichten*

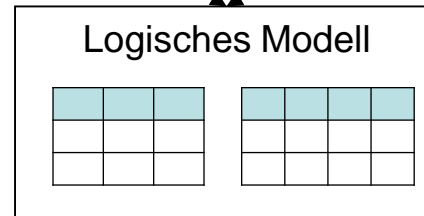


# 3-Ebenen-Architektur

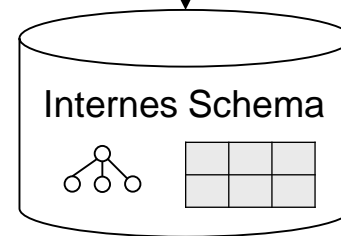
✓ Externe Ebene



✓ Konzeptuelle Ebene



Interne Ebene





# Interne Ebene

## Aspekte des physischen Entwurfs

- Verteilung der Tabellen auf physische Datenträger
- Speicherungsform der Daten  $\Rightarrow$  **Dateiorganisation**
- Clustering von Daten
- Definition von  $\Rightarrow$  **Indexen** auf den Tabellen
- ...

- Beispiel:

<u>KontoNr</u>	Kunde	Typ	Saldo
3002700	K711	GI	2.728,00
3002720	K711	GM	10.000,00
3003200	K589	GI	-27,53
3002721	K711	GM	3.000,00
3003220	K589	GM	100,00
...	...	...	...



# Dateiorganisation – 1(2)

## Heap-Organisation (Stapeldatei)

Seite 47

3002700	K711	GI	2.728,00
3002720	K711	GM	10.000,00
3003200	K589	GI	-27,53

Seite 48

3002721	K711	GM	3.000,00
3003220	K589	GM	100,00

- **Vorteil:** Einfügen sehr schnell
- **Nachteil:** Suchen und Löschen sehr(!) aufwendig
- Alternative?



# Dateiorganisation – 2(2)

## Sortierte Speicherung

Seite 47

3002700	K711	GI	2.728,00
3002720	K711	GM	10.000,00
3002721	K711	GM	3.000,00



Seite 48

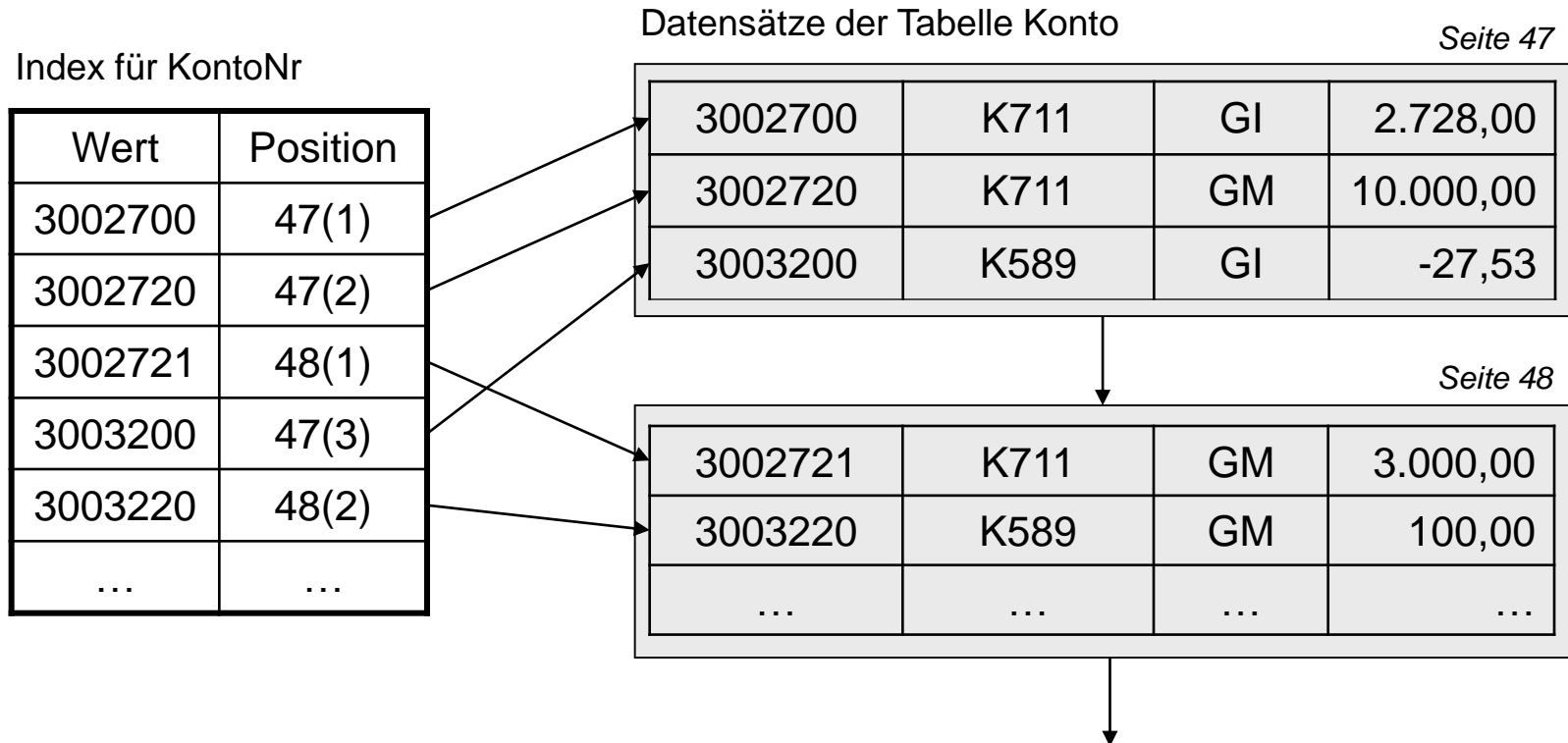
3003200	K589	GI	-27,53
3003220	K589	GM	100,00



- **Vorteil:** Für Werte der sortierten Spalte: Suche und Löschen schnell
  - **Nachteil:** Einfügen sehr(!) aufwendig (Einträge müssen verschoben werden)
- ⇒ Nur für kleine oder relativ statische Datenbestände geeignet!



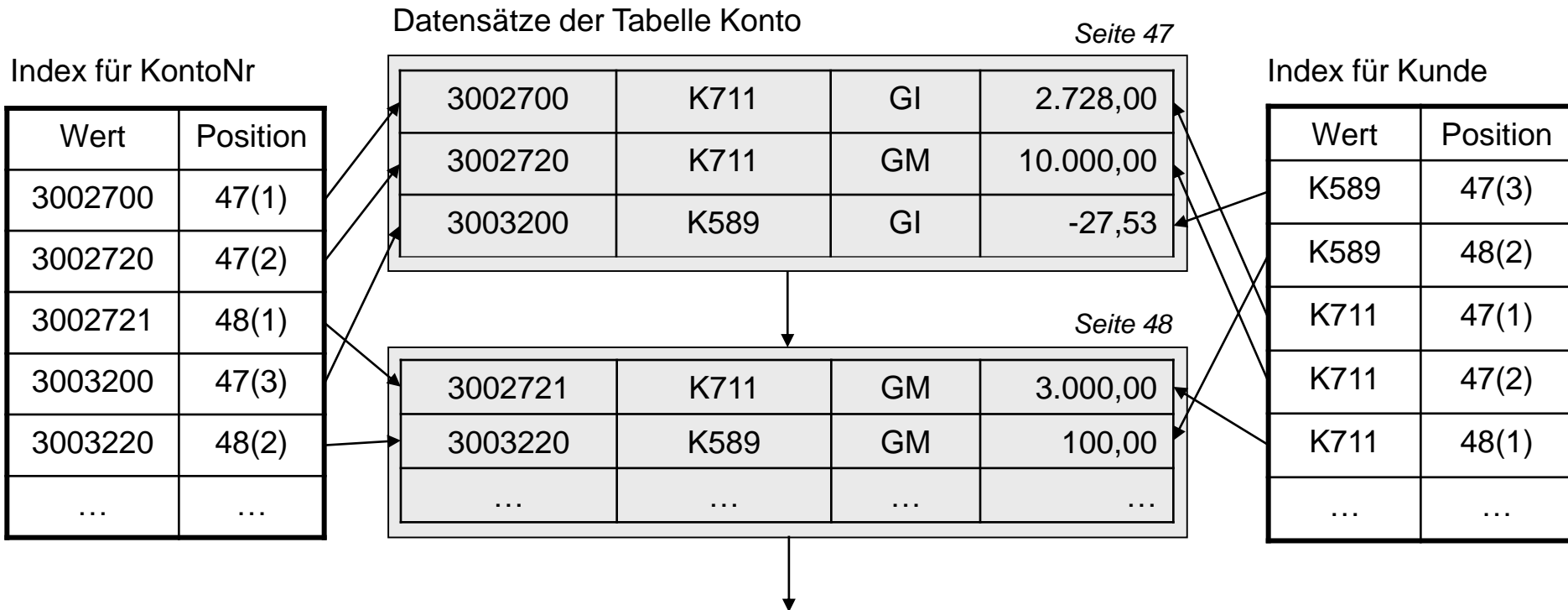
# Indexe – 1(3)



- **Vorteil:** schnellere Suche für indexierte Werte
- **Nachteil:** beim Einfügen, Änderung und Löschen muss auch der Index aktualisiert werden!



# Indexe – 2(3)

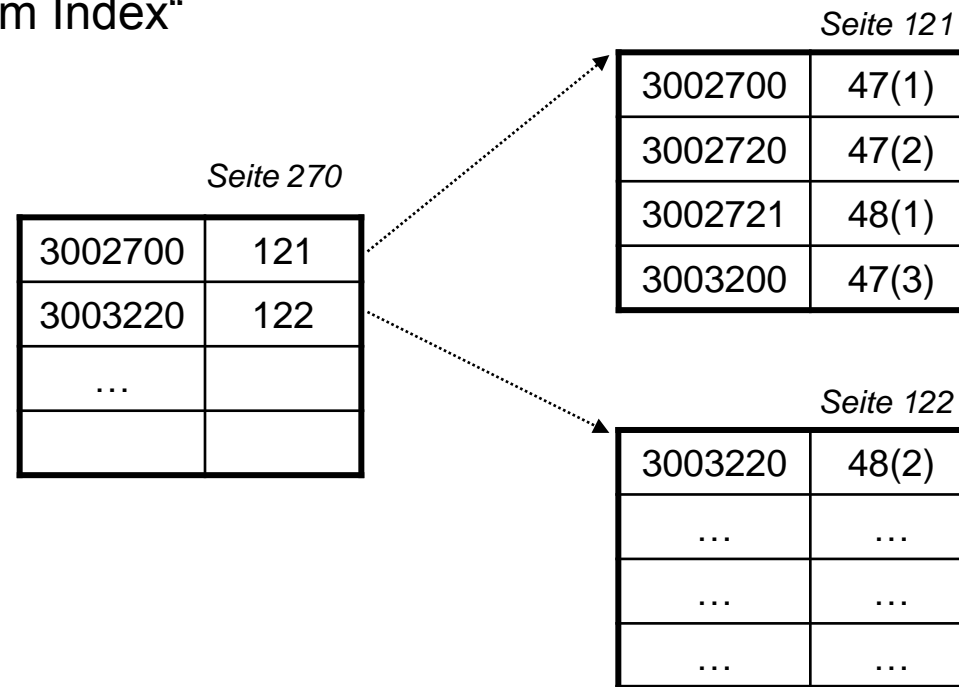


- Immer beachten: beim Einfügen, Änderung und Löschen muss (durch das DBMS) auch der Index aktualisiert werden!
- ⇒ Index verbessert Suchgeschwindigkeit, aber ist „teuer“ (Performance)



# Indexe – 3(3)

- Ist „linearer“ Index sinnvoll?
  - 100.000 Datensätze  $\Rightarrow$  100.000 Index-Einträge
  - Bei Suchanfrage müssen auch diese 100.000 Index-Einträge erstmal sequentiell gelesen werden ...
- $\Rightarrow$  „Index auf dem Index“



Index ...

Index 2. Stufe

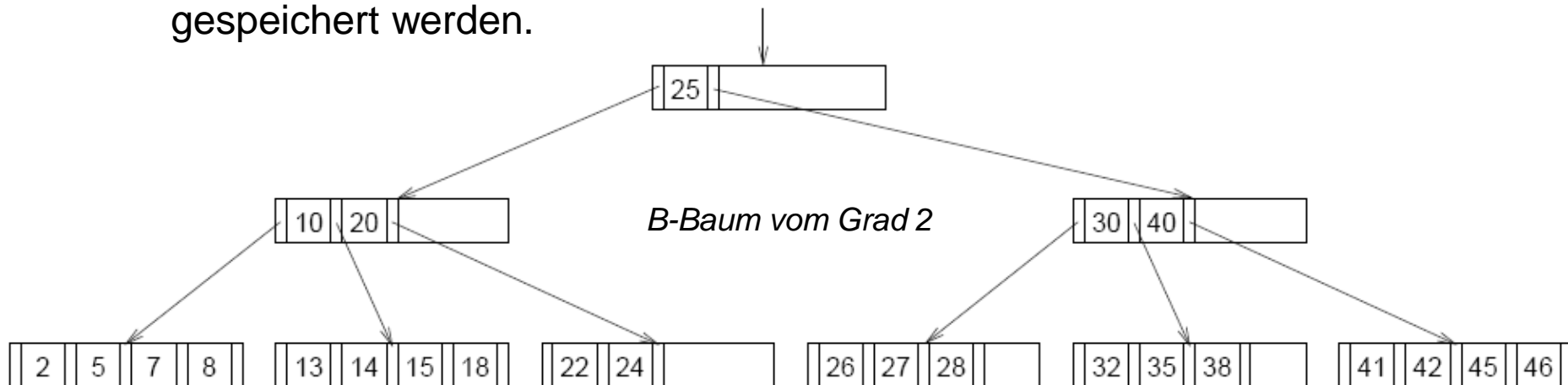
Index 1. Stufe



# B-Bäume

## B-Baum vom Grad $m$

1. Jede Seite, außer der Wurzelseite, enthält mindestens  $m$  Elemente.
2. Jede Seite enthält höchstens  $2m$  Elemente.
3. Jeder Weg von der Wurzelseite zu einer Blattseite hat die gleiche Länge. ( $\Rightarrow$  *höhenbalancierter Baum*)
4. Die Elemente werden in allen Seiten sortiert gespeichert. Jede Seite ist entweder eine Blattseite ohne Nachfolger oder hat  $i + 1$  Nachfolger, falls  $i$  die Anzahl ihrer Elemente ist.
5. Für einen Element  $E_i$  gilt, dass die Werte zwischen  $E_{i-1}$  und  $E_i$  im linken Teilbaum und die Werte zwischen  $E_i$  und  $E_{i+1}$  im rechten Teilbaum gespeichert werden.

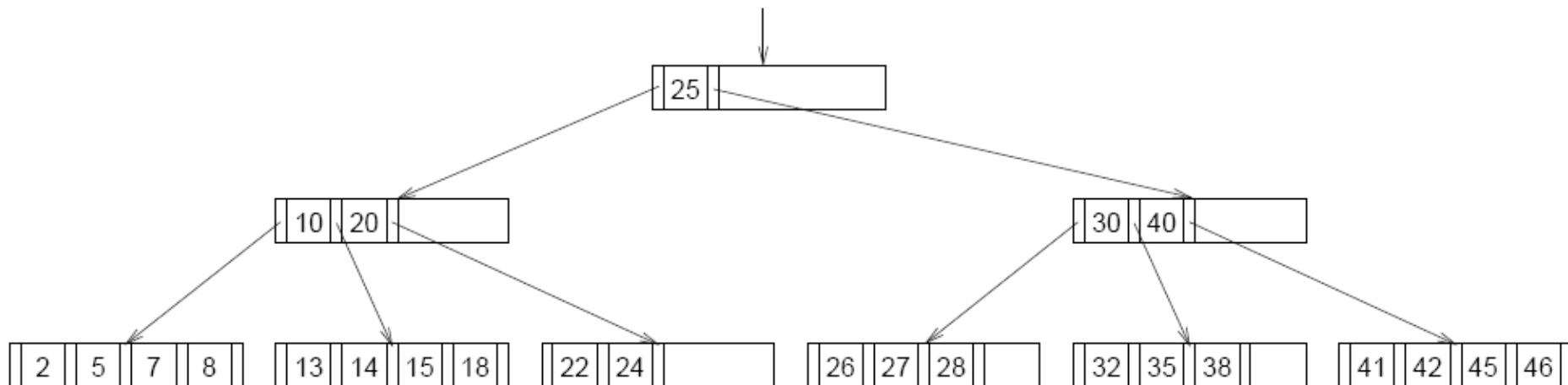




# Suchen in B-Bäumen

## lookup

- startend auf Wurzelseite Eintrag im B-Baum ermitteln, der den gesuchten Wert enthält  $\Rightarrow$  Zeiger verfolgen, Seite nächster Stufe laden
- Beispiel: Suchen: 38, 20, 6





# Einfügen in B-Bäumen

---

## **insert:** Einfügen eines Wertes

- mit **lookup** entsprechende **Blattseite** suchen
  - Falls passende Seite  $n < 2m$  Elemente (d.h. noch nicht voll)  
 $\Rightarrow w$  einsortieren
  - Falls passende Seite  $n = 2m$  Elemente (d.h. bereits voll)  
 $\Rightarrow$  neue Seite erzeugen und
    - ersten  $m$  Werte auf Originalseite belassen
    - letzten  $m$  Werte auf neue Seite speichern
    - mittleres Element auf entsprechende Indexseite nach oben verschieben
  - ggf. diesen Prozess rekursiv bis zur Wurzel wiederholen

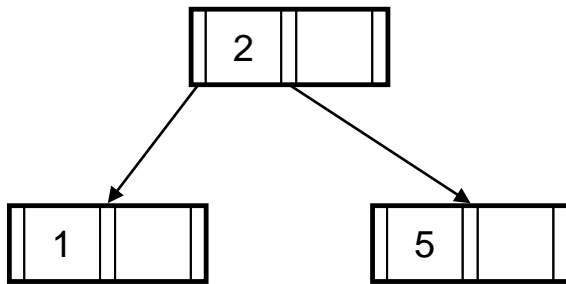
## **delete:** Löschen eines Wertes

- siehe z.B. *Kemper/Eickler: Datenbanksysteme*



# B-Bäume: Beispiel

- Baum mit  $m=1$
- Die Zahlen 1, 5, und 2 wurden schon eingefügt ...



- ... die Zahlen 6, 7, 4, 8, 3 sollen noch eingefügt werden



# B-Bäume: Eigenschaften und Anwendung

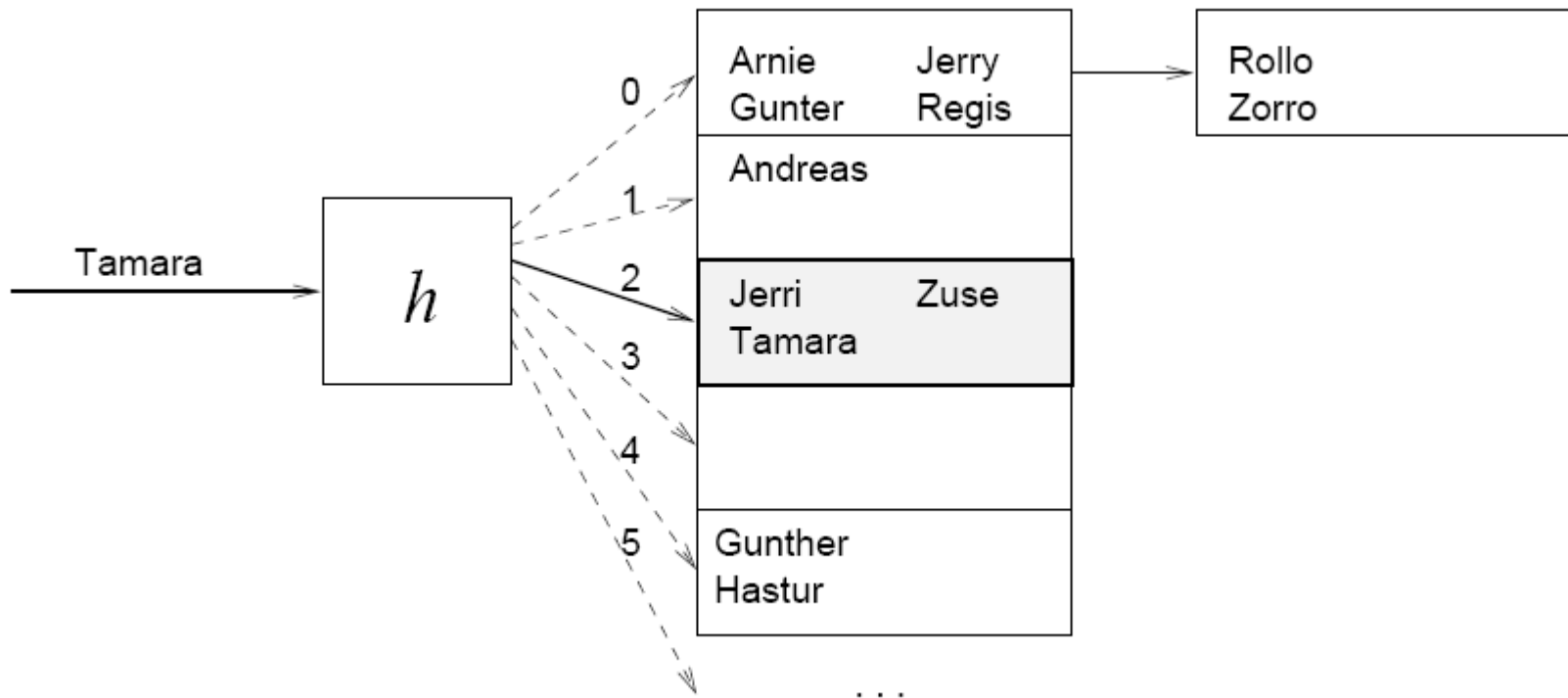
---

- Höhe eines Baums vom Grad  $m$  mit  $n$  Einträgen:  $\leq \log_m(n)$ 
    - ⇒ Aufwand beim Einfügen, Suchen und Löschen im B-Baum beträgt immer  $O(\log_m(n))$  – zum Vergleich: lineare Suche:  $O(n)$
  - B-Bäume als Datenstruktur für Datenbank-Indexe:
    - Realistische B-Bäume haben Größenordnungen von  $m=100$  (abhängig von der Größe der Datensätze und dem Fassungsvermögen der Seite).
    - ⇒ Um einen Datensatz unter  $10^7$  Einträgen in einem B-Baum mit  $m=100$  zu finden, braucht man maximal:  $\log_m(n) = \log_{100}(10^7) = 3,5$  d.h. maximal **4** Seitenzugriffe  
zum Vergleich: bei linearer Suche (und 10 Datensätzen pro Seite) wären dies  $10^7/10 = 10^6 = \mathbf{1.000.000}$  Zugriffe
- ➔ B-Bäume sind wichtige und weit verbreitete Datenstruktur zur Implementierung von Datenbank-Indexen!



# Hash-Index

- Hash-Verfahren:



- Aufwand beim Einfügen, Suchen und Löschen?
- Vor- und Nachteile gegenüber Baumverfahren?

Quelle: Saake/Heuer/Sattler:2005



# Definition von Indexen – 1(2)

---

## Auf welchen Attributen sollten Indexe definiert werden?

- auf Primärschlüsseln! (wird von vielen DBMS automatisch erzeugt)
- auf Fremdschlüsseln oft sinnvoll
- auf weiteren, häufig in Suchanfragen verwendeten Attributen

## Wichtige Eigenschaften

- ein Index kann aus mehreren Attributen bestehen
- auf einer Relation (Tabelle) können mehrere Indexe definiert sein

**Nicht vergessen:** Indexe müssen (vom DBMS) bei Änderungsoperationen aktualisiert werden (Performance!)



# Definition von Indexen – 2(2)

- Leider seit SQL-92 (nicht mehr) im SQL-Standard definiert ☹
- In fast allen Systemen wird die folgende Notation unterstützt:

```
CREATE [ UNIQUE ] INDEX index-name  
ON relationen-name ( spaltenname1 [ ordnung1 ],  
                    ...,  
                    spaltennamen [ ordnungn ] )  
mit ordnungk := ASC | DESC
```

- Abhängig vom konkreten DBMS können oft noch weitere Optionen (z.B. Auswahl des Index-Typs) angegeben werden
- Beispiel:

```
CREATE INDEX StudName  
ON Studenten (Name)
```

- Löschen eines Index:

```
DROP INDEX index-name
```





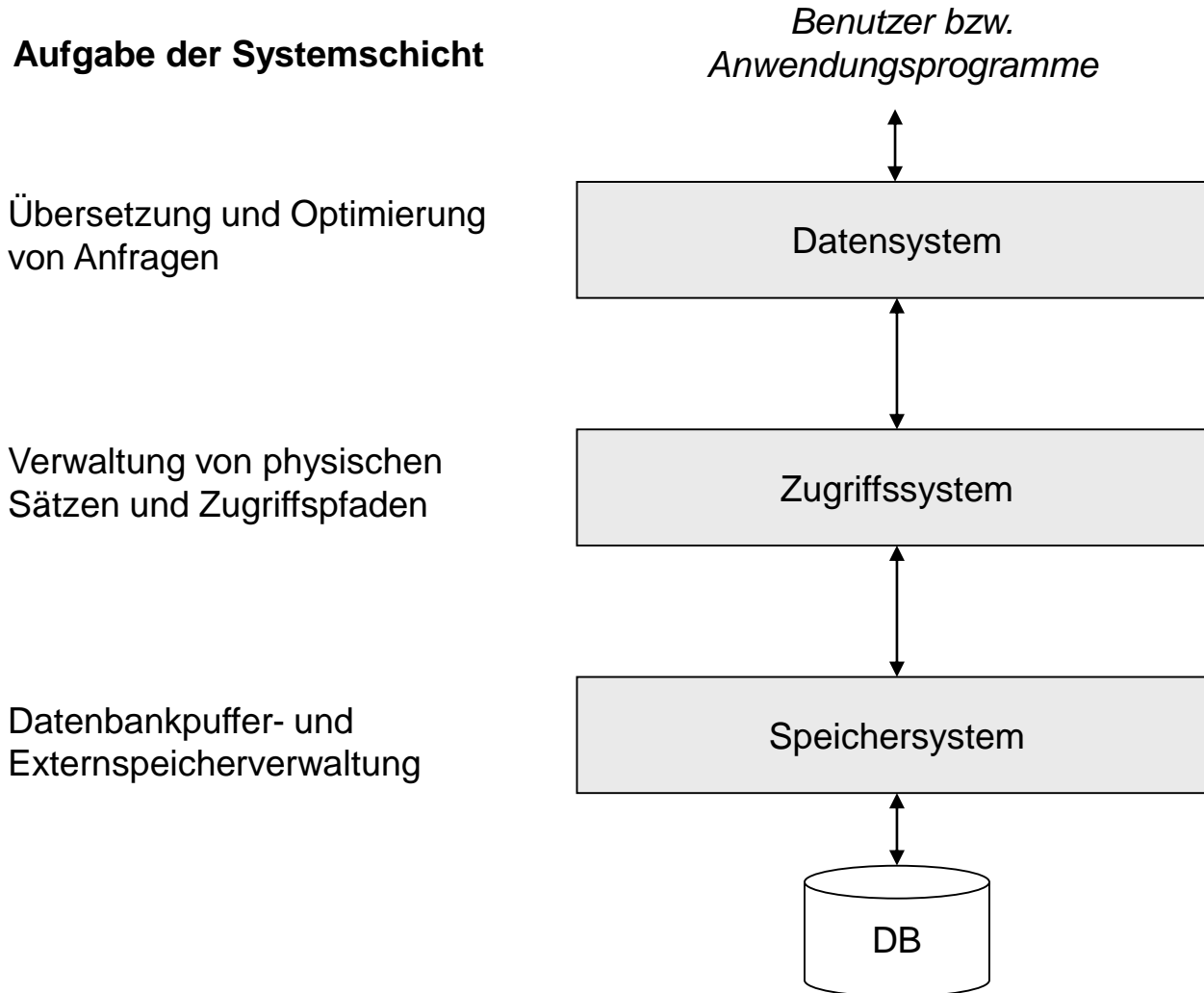
# Architektur von Datenbanksystemen

---

- ✓ 3-Ebenen-Architektur von Datenbanken
  - ✓ Externe Ebene
  - ✓ Konzeptionelle Ebene
  - ✓ Interne Ebene
- DBMS-Systemarchitektur
  - Schichtenarchitektur
  - Transaktionsverwaltung und Recovery



# Schichtenarchitektur von DBMS



Quelle: Haerder/Rahm:1999



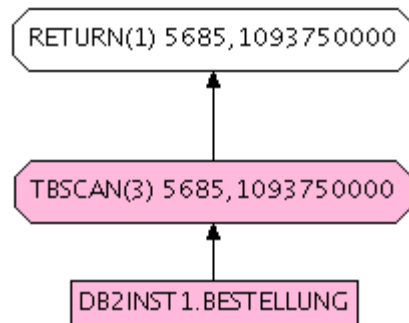
# Datenystem: Anfragepläne – 1(3)

## Datenystem: Übersetzung und Optimierung der Anfragen

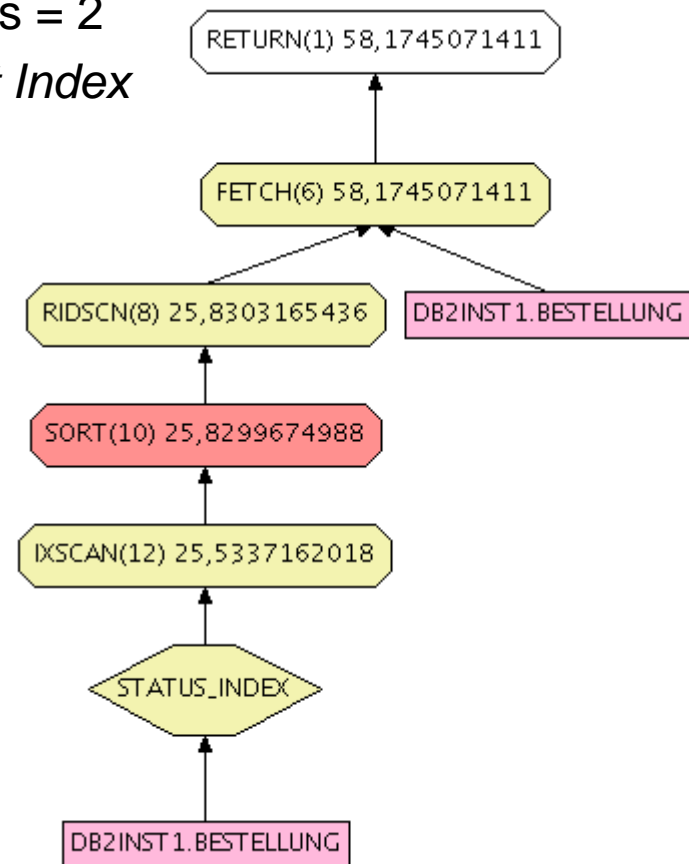
*Beispiel: Nutzung eines ggf. vorhandenen Index*

`select * from bestellung where status = 2`

*ohne Index*



*mit Index*



*IBM DB2: Visual Explain*

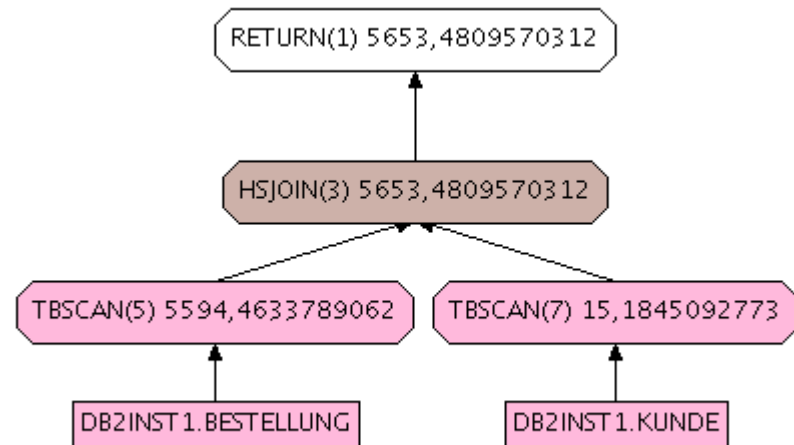


# Datensystem: Anfragepläne – 2(3)

... unterschiedlich formulierte SQL-Anfragen werden auf ihre Grundoperationen zurückgeführt:

```
select kunde.knr, kname  
from kunde, bestellung  
where kunde.knr = bestellung.knr
```

```
select kunde.knr, kname  
from kunde join bestellung on kunde.knr = bestellung.knr
```



IBM DB2: Visual Explain



# Datensystem: Anfragepläne – 3(3)

... Optimierung erfolgt basierend auf der Umformung in die Relationenalgebra:

The screenshot shows the Oracle SQL Developer interface. The top toolbar has a red circle around the 'Explain Plan' icon. The 'Arbeitsblatt' (Worksheet) tab is active, displaying the following SQL query:

```
1 select kunde.id, nachname from kunde, bestellung where kunde.id = bestellung.kunde_id;
2 |
```

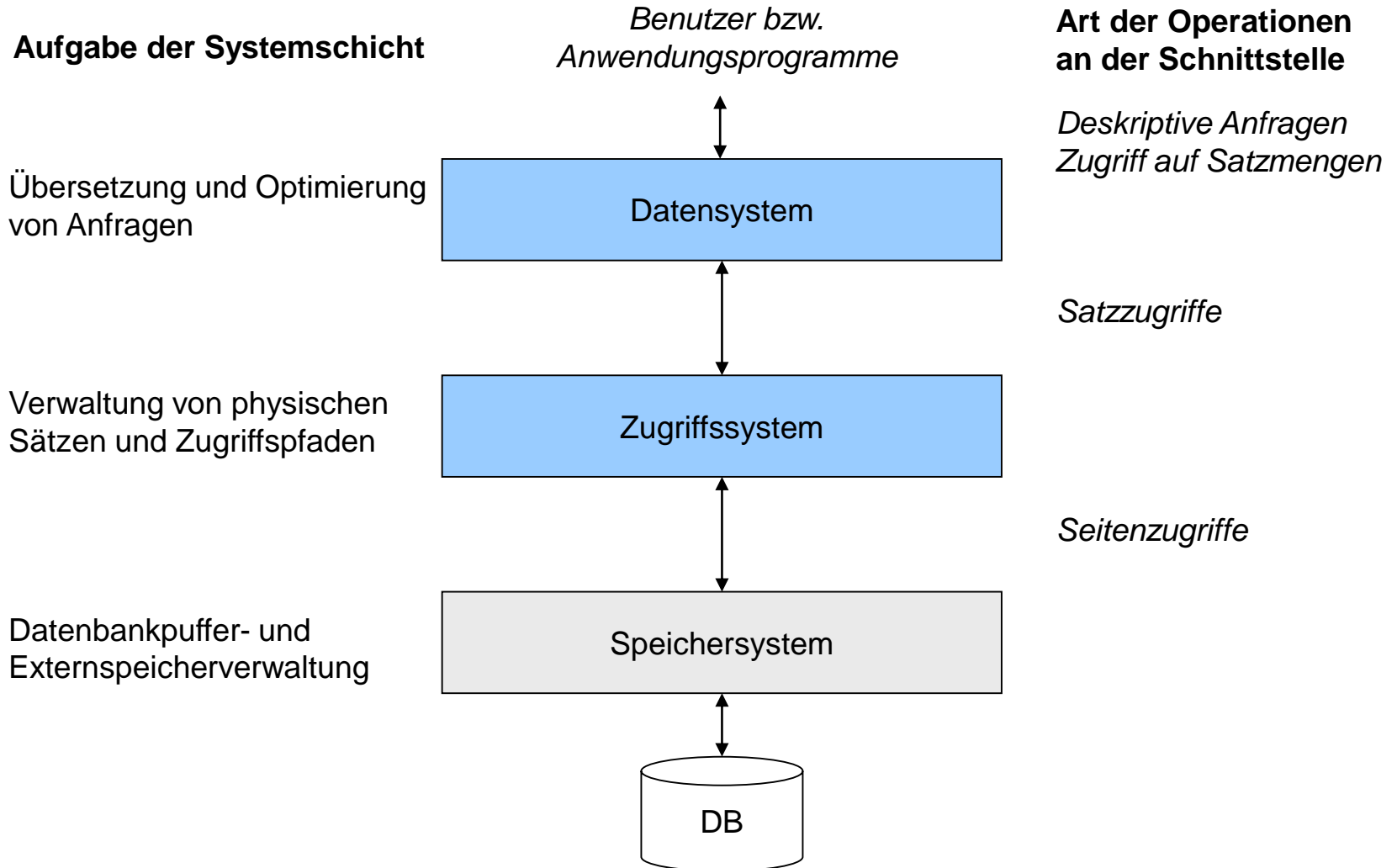
The 'Explain-Plan' window is open, showing the execution plan for the query. The plan is as follows:

OPERATION	OBJECT_NAME	OPTIONS	COST
SELECT STATEMENT			3
NESTED LOOPS			3
TABLE ACCESS	BESTELLUNG	FULL	3
TABLE ACCESS	KUNDE	BY INDEX ROWID	0
INDEX	SYS_C008535	UNIQUE SCAN	0
Access Predicates			
KUNDE.ID=BESTELLUNG.KUNDE_ID			

Oracle: SQL Developer



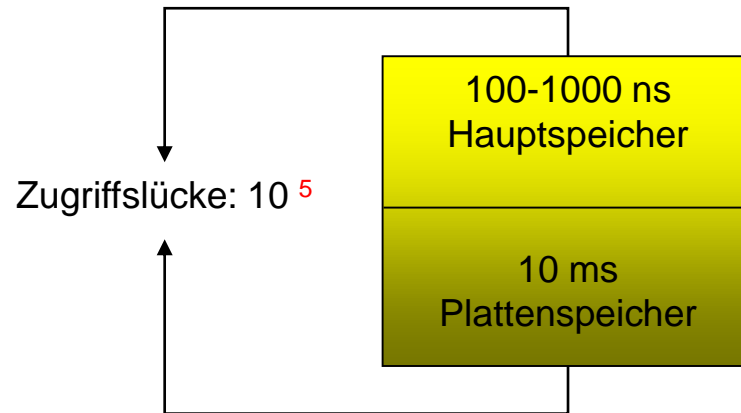
# Schichtenarchitektur von DBMS





# Speichersystem: Speicherhierarchie

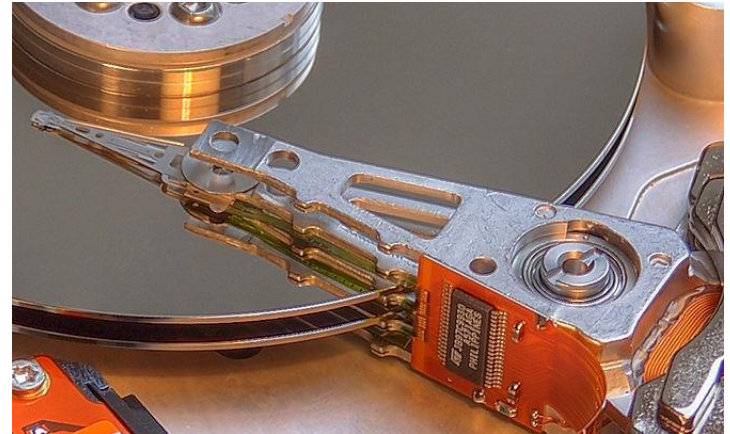
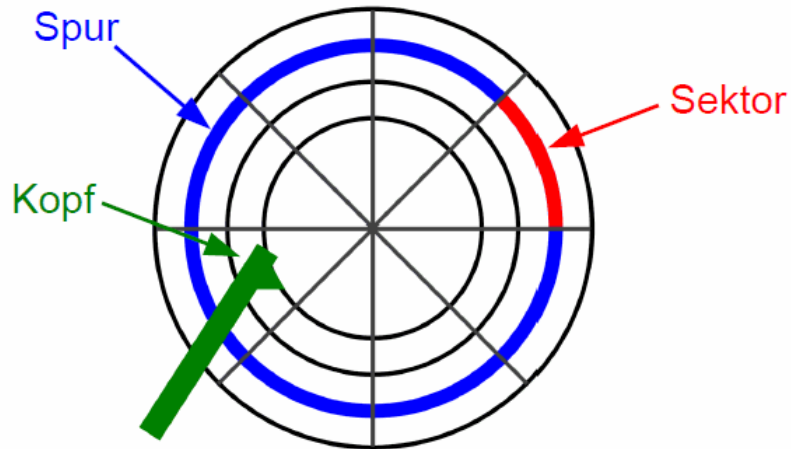
Geschwindigkeit Plattenzugriff gegenüber Hauptspeicher-Zugriff:



100 Seiten lesen:

$$100 \times 100 \text{ ns} = 10.000 \text{ ns} = 0,01 \text{ ms}$$

$$100 \times 10 \text{ ms} = 1.000 \text{ ms} = 1 \text{ s}$$

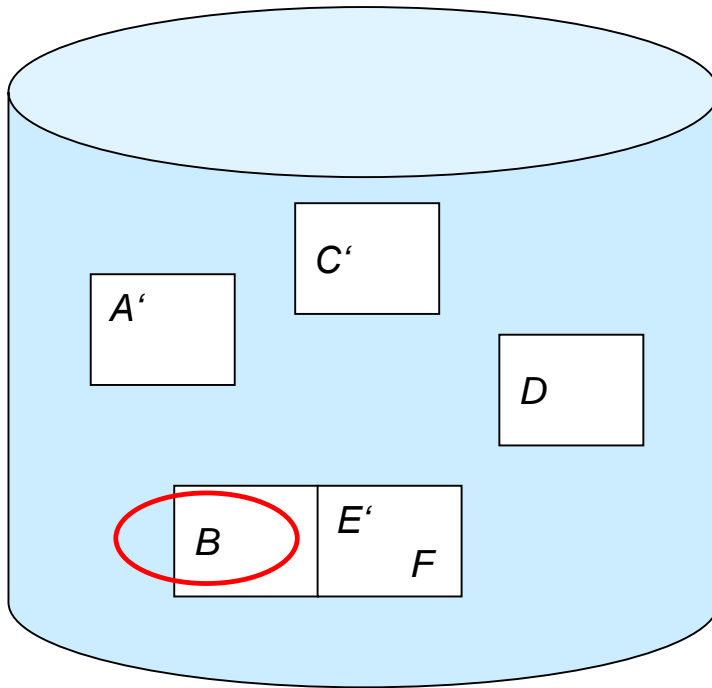


Quelle: Wikimedia

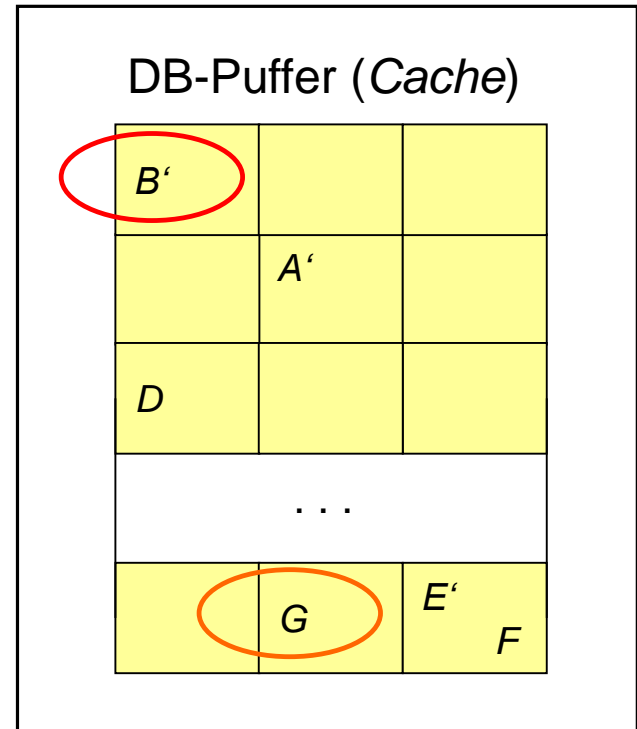


# Speichersystem: DB-Pufferverwaltung

Datenbank auf dem Externspeicher



Hauptspeicher



Einlagerung

Auslagerung

- Typische Seitengröße: zwischen 4 und 32 KB (fest)
  - Ziel: die nächste benötigte (zu lesende) Seite, soll möglichst bereits im Puffer sein
- ⇒ verschiedene Strategien zur Pufferersetzung





# Architektur von Datenbanksystemen

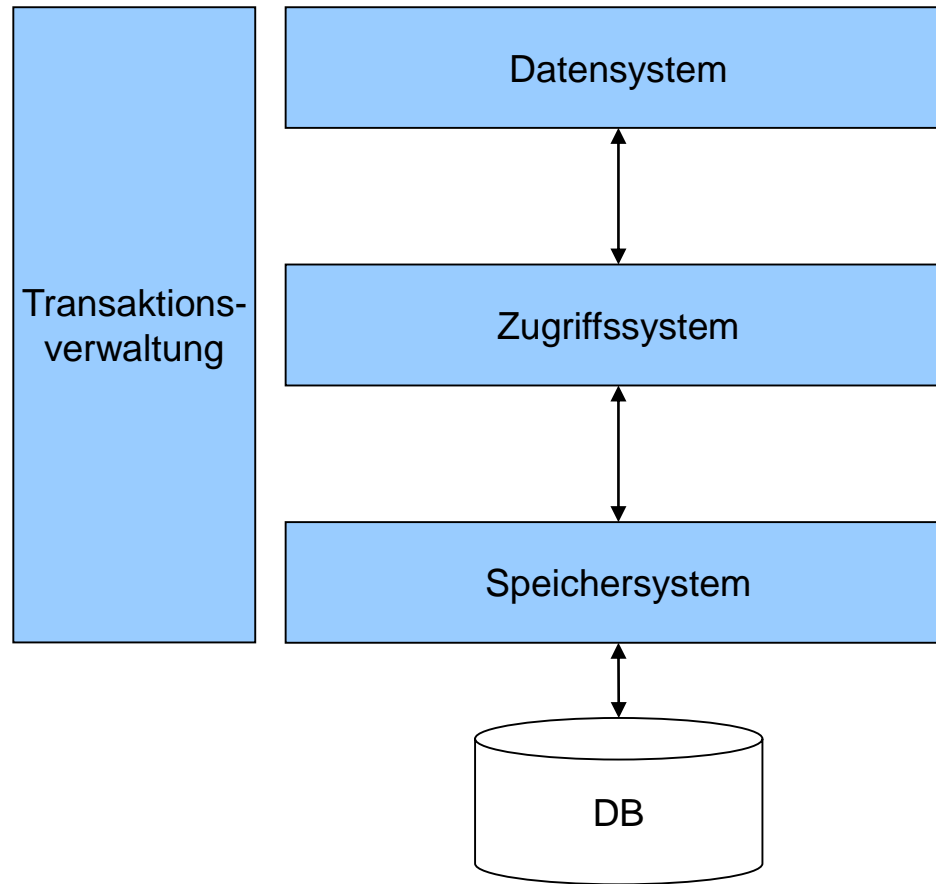
---

- ✓ 3-Ebenen-Architektur von Datenbanken
  - ✓ Externe Ebene
  - ✓ Konzeptionelle Ebene
  - ✓ Interne Ebene
- DBMS-Systemarchitektur
  - ✓ Schichtenarchitektur
    - Transaktionsverwaltung und Recovery



# Schichtenarchitektur von DBMS: weitere Komponenten

---





# Transaktionen – Warum?

## Beispiel

- Was passiert, wenn während der Ausführung des unten stehenden PL/SQL-Programms das DBMS abstürzt / oder das Betriebssystem abstürzt / oder der Strom ausfällt ...?

```
...  
begin  
  for ProdRecord in CurProd loop  
    if ProdRecord.MarktPreis < 100 then  
      update Toepferprodukt_Markt  
      set MarktPreis = MarktPreis + 10  
      where current of CurProd;  
    else  
      update Toepferprodukt_Markt  
      set MarktPreis = MarktPreis + 20  
      where current of CurProd;  
    end if;  
  end loop;  
end;  
/
```



# Transaktionsverwaltung und Recovery

---

- Transaktionsbegriff
- Mehrbenutzersynchronisation
- Fehlerbehandlung



# Transaktion – Definition

---

- Eine **Transaktion** ist eine Folge von Datenbankoperationen, die eine Datenbank von einem konsistenten Zustand in einen neuen konsistenten Zustand überführt und entweder ganz oder gar nicht ausgeführt wird.
- „Gar nicht“ bedeutet, dass beim Abbruch einer Transaktion (egal ob explizit oder implizit) , diese vollständig zurückgesetzt, d.h. alle ausgeführten Änderungen in der Datenbank rückgängig gemacht werden müssen.
  - ⇒ Entsprechende Fehlerbehandlungsmechanismen notwendig
- ⇒ Eine Transaktion wird oft auch als logisch atomare Einheit oder *logical unit of work* bezeichnet.



# Operationen zur Transaktionsverarbeitung

---

Allgemein (unabhängig vom verwendeten Datenmodell oder DBMS):

- **begin of transaction (BOT)**
  - Markiert den Beginn einer Transaktion.
- **commit**
  - Markiert das Ende einer Transaktion. Alle Änderungen werden in der Datenbank festgeschrieben, d.h. dauerhaft in der Datenbank abgelegt.
- **abort**
  - Abbruch der Transaktion (z.B. explizit durch Nutzer oder Programm oder implizit – z.B. bei Connection-Verlust). Das DBMS muss sicherstellen, dass die Datenbank wieder in den Zustand zurückgesetzt wird, der vor Beginn der Transaktionsausführung existierte.



# Anwendung der Operationen

Ausgangswerte: A = 100; B = 200

**BOT**

```
read(A,a);  
a:=a-50;  
write(A,a);  
read(B,b);  
b:=b+50;  
write(B,b);  
commit
```

A = ... B = ...

**BOT**

```
read(A,a);  
a:=a-50;  
write(A,a);  
abort
```

oder

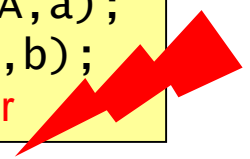
**BOT**

```
read(A,a);  
a:=a-50;  
write(A,a);  
read(B,b);  
b:=b+50;  
write(B,b);  
abort
```

A = ... B = ...

**BOT**

```
read(A,a);  
a:=a-50;  
write(A,a);  
read(B,b);  
Fehler
```



A = ... B = ...



- Kein separater Befehl für „begin of transaction“ – Transaktion beginnt implizit mit der ersten Anweisung
- `commit [ work ]`
- `rollback [ work ]`

```
-- Transaktion T1 wird implizit geöffnet
update Konto set balance = balance-50 where KontoID = 'A';
update Konto set balance = balance+50 where KontoID = 'B';

-- T1 wird beendet und die Ergebnisse in der DB festgeschrieben
commit work;

-- neue Transaktion T2 wird implizit geöffnet
insert into Konto (KontoID, Name, balance)
                        values ('C', 'Meyer', 0);
...
```





# Transaktionen in JDBC

- Transaktionskontrolle durch Methodenaufrufe der Klasse Connection **setAutoCommit**
  - **true**: jedes Statement ist eine eigene Transaktion
  - **false**: Transaktion wird bei erstem Statement eröffnet und mit **commit** beendet bzw. **rollback** abgebrochen

```
Connection con = DriverManager.getConnection(url, user, pwd);
...
try {
    con.setAutoCommit (false);
    // ... update ...
    // ... update ...
    con.commit ();
}
catch (SQLException e2) {
    con.rollback ();
}
```



# Was passiert bei Mehrbenutzerbetrieb?

## Beispiel

- Zwei Programme („Überweisung“ und „Zinsgutschrift“) arbeiten gleichzeitig auf der Datenbank:

Zeit	Transaktion 1	Transaktion 2	Zustand von A in der Datenbank
1	read (A,a1)		1000
2	a1 := a1 - 50		1000
3		read (A,a2)	1000
4		a2 = a2 * 1.03	1000
5		write (A, a2)	1030
6		commit	1030
7	write (A,a1)		950
8	read (B,b1)		950
9	b1 := b1 + 50		950
10	write (B,b1)		950
11	commit		950

- Sog. **lost update Phänomen**



# Eigenschaften von Transaktionen – 1(3)

---

## **ACID-Paradigma** für Transaktionen [Härder/Reuter:1983]

- **A**tomicity (Atomarität)
- **C**onsistency (Konsistenz oder auch Integritätserhaltung)
- **I**solation (Isolation)
- **D**urability (Dauerhaftigkeit)



# Eigenschaften von Transaktionen – 2(3)

---

- **Atomicity (Atomarität)**
  - Transaktion wird als kleinste, nicht mehr zerlegbare Einheit behandelt, d.h.
  - entweder werden alle Änderungen der Transaktion in der Datenbank festgeschrieben oder keine.
  - ⇒ Mechanismen zur **Fehlerbehandlung** notwendig.
- **Consistency (Konsistenz oder auch Integritätserhaltung)**
  - Eine Transaktion überführt die Datenbank von einem konsistenten Zustand in einen neuen konsistenten Datenbankzustand.
  - Zwischenzustände dürfen inkonsistent sein!, aber der resultierende Endzustand (also insbesondere auch nach einem Abbruch) muss die im Schema definierten Konsistenzbedingungen (z.B. referentielle Integrität) erfüllen.
  - ⇒ Mechanismen zur **Konsistenzsicherung** und **Fehlerbehandlung** notwendig.



# Eigenschaften von Transaktionen – 3(3)

---

- **Isolation (Isolation)**
  - Nebenläufig (parallel, gleichzeitig) ausgeführte Transaktionen dürfen sich nicht gegenseitig beeinflussen.
  - Jede Transaktion muss – logisch gesehen – so ausgeführt werden, als wäre sie die einzige Transaktion die während ihrer gesamten Ausführungszeit auf der Datenbank aktiv ist.
  - ⇒ Mechanismen zur **Mehrbenutzersynchronisation** notwendig.
- **Durability (Dauerhaftigkeit)**
  - Nach dem erfolgreichen Abschluss einer Transaktion muss das Ergebnis dieser Transaktion „dauerhaft“ in der Datenbank gespeichert werden,
  - insbesondere muss eine beendete Transaktion bzw. die von ihr auf der Datenbank ausgeführten Veränderungen auch einen Systemfehler (Hard- oder Software) „überleben“.
  - ⇒ Mechanismen zur **Fehlerbehandlung** notwendig.



# Transaktionsverwaltung und Recovery

---

- ✓ Transaktionsbegriff
- Mehrbenutzersynchronisation
  - Anomalien beim Mehrbenutzerbetrieb
  - Konzept der Serialisierung
  - Sperren zur Umsetzung der Isolationseigenschaft von Transaktionen
  - Unterschiedliche Isolation Level
  - Multiversion Concurrency Control
- Fehlerbehandlung



# Mehrbenutzersynchronisation

---

Ziel: Eigenschaft der Isolation (Isolation)

- Nebenläufig (parallel, gleichzeitig) ausgeführte Transaktionen dürfen sich nicht gegenseitig beeinflussen. Jede Transaktion muss – logisch gesehen – so ausgeführt werden, als wäre sie die einzige Transaktion die während ihrer gesamten Ausführungszeit auf der Datenbank aktiv ist.

Vorgehen

- Bei der gleichzeitigen Ausführung mehrerer Transaktionen können verschiedene „Effekte“ (Anomalien) auftreten, die für den Anwender i.a. nicht akzeptabel sind
- ⇒ Klassifikation dieser Anomalien und Identifikation von Mechanismen, um diese zu vermeiden.



# Verlorengegangene Änderungen (*lost update*)

- Zwei Programme („Überweisung“ und „Zinsgutschrift“) arbeiten gleichzeitig auf der Datenbank:

Zeit	Transaktion 1	Transaktion 2	Zustand von A auf der Datenbank
1	read (A,a1)		1000
2	a1 := a1 - 50		...
3		read (A,a2)	...
4		a2 = a2 * 1.03	...
5		write (A, a2)	1030
6		commit	...
7	write (A,a1)		950
8	read (B,b1)		...
9	b1 := b1 + 50		...
10	write (B,b1)		...
11	commit		950





# Abhängigkeit von nicht freigegebenen Änderungen (*dirty read*)

- *Der neue Mitarbeiter B soll 10% mehr als Mitarbeiter A verdienen.*
- *Mitarbeiter A erhält eine Gehaltserhöhung um 100 Euro.*  
*Diese Transaktion wird abgebrochen, da am Ende der Transaktion die Verletzung einer Integritätsbedingung festgestellt wird (z.B. Maximalgrenze für Gehalt einer bestimmten Tarifgruppe verletzt).*

Zeit	Transaktion 1	Transaktion 2	A	B
1	read (A,a1)		2500	
2	a1 := a1 + 100		2500	
3	write (A,a1)		2600	
4		read (A,a2)	...	
5		b1 = a2 * 1.1	...	
6		write (B, b1)	...	2860
7		commit	2600	2860
9	abort		2500	2860

- Problem?



# Inkonsistente Analyse (*non repeatable read*)

- Die Summe der Gehälter aller Mitarbeiter wird ermittelt.
- Parallel dazu werden die Gehälter der Mitarbeiter um jeweils 1.000 Euro erhöht.

Zeit	Transaktion 1	Transaktion 2
1	read (A, a1)	
2	summe = summe + a1	
3		read (A, a2)
4		a2 = a2 + 1000
5		write (A, a2)
6		read (B, b2)
7		b2 = b2 + 1000
8		write (B, b2)
9		commit
10	read (B, b1)	
11	summe = summe + b1	
12	commit	

- Problem?



# Phantom-Problem

- *Ein Bonus von 100.000 Euro soll gleichmäßig auf alle Mitarbeiter der Firma verteilt werden.*
- *Parallel dazu wird ein neuer Mitarbeiter eingefügt.*

Zeit	Transaktion 1	Transaktion 2
1	<code>select count(*) into x</code>	
2	<code>from Mitarbeiter;</code>	<code>insert into Mitarbeiter</code>
3		<code>values (Meier, 50.000, ...);</code>
4	<code>update Mitarbeiter</code>	<code>commit;</code>
5	<code>set Gehalt = Gehalt + 100.000/x;</code>	
	<code>commit;</code>	

- Problem?



# Synchronisation von Transaktionen: Modell

---

Ziel der Synchronisation: Vermeidung aller Mehrbenutzeranomalien

## Modell

- Wenn Transaktionen seriell ausgeführt werden, dann bleibt die Konsistenz der DB erhalten.
- Transaktion: BOT, Folge von READ- und WRITE-Anweisungen, EOT
- Die Ablauffolge von Transaktionen mit ihren Operationen kann durch einen *Schedule* beschrieben werden  
(BOT ist implizit, EOT wird durch  $c_i$  (*commit*) oder  $a_i$  (*abort*) dargestellt):
  - Beispiel:  
 $r_1(x), r_2(x), r_3(y), w_1(x), w_3(y), r_1(y), c_1, r_3(x), w_2(x), a_2, w_3(x), c_3, \dots$
  - Beispiel eines *seriellen Schedules* :  
 $r_1(x), w_1(x), r_1(y), c_1, r_3(y), w_3(y), r_3(x), c_3, r_2(x), w_2(x), c_2, \dots$



# Korrektheitskriterium der Synchronisation: Serialisierbarkeit – 1(2)

---

- Ziel der Synchronisation: logischer Einbenutzerbetrieb, d.h. Vermeidung aller Mehrbenutzeranomalien
- Gleichbedeutend mit formalem Korrektheitskriterium der **Serialisierbarkeit**:
  - Die parallele Ausführung einer Menge von n Transaktionen ist serialisierbar, wenn es eine serielle Ausführung derselben Transaktionen gibt, die den gleichen DB-Zustand und die gleichen Ausgabewerte wie die ursprüngliche Ausführung erzielt.
- Hintergrund:
  - serielle Ablaufpläne sind korrekt
  - jeder Ablaufplan, der denselben Effekt wie ein serieller erzielt, ist akzeptierbar



# Korrektheitskriterium der Synchronisation: Serialisierbarkeit – 2(2)

Transaktion 1

$T_1 - 1$
$T_1 - 2$
$T_1 - 3$

Transaktion 2

$T_2 - 1$
$T_2 - 2$
$T_2 - 3$

Transaktion 3

$T_3 - 1$
$T_3 - 2$

(Quasi-) parallele Ausführung ...

$T_1 - 1$
$T_2 - 1$
$T_3 - 1$
$T_1 - 2$
$T_1 - 3$
$T_2 - 2$
$T_2 - 3$
$T_3 - 2$

Die parallele Ausführung einer Menge von  $n$  Transaktionen ist *serialisierbar*, wenn es eine serielle Ausführung derselben Transaktionen gibt, die den gleichen DB-Zustand und die gleichen Ausgabewerte wie die ursprüngliche Ausführung erzielt.

$T_2 - 1$
$T_2 - 2$
$T_2 - 3$
$T_3 - 1$
$T_3 - 2$
$T_1 - 1$
$T_1 - 2$
$T_1 - 3$

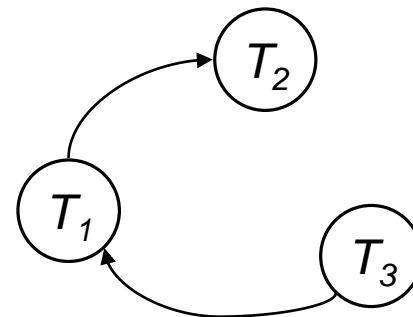
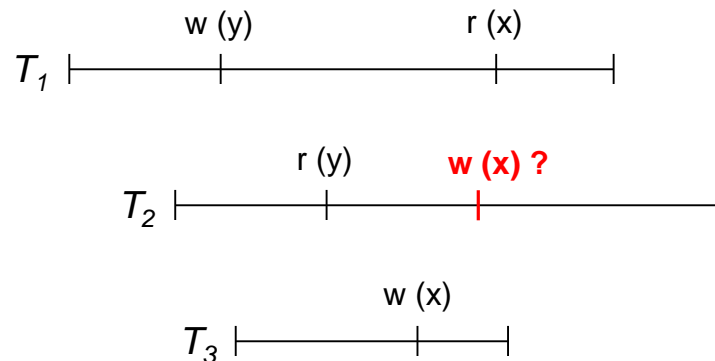


# Untersuchung der Serialisierbarkeit: Abhängigkeitsgraphen (Konfliktgraphen)

Serialisierbarkeit lässt sich durch die Untersuchung von zeitlichen Abhängigkeiten zwischen Transaktionen in einem *Abhängigkeitsgraphen* (*Konfliktgraphen*) ermitteln

- Konfliktarten:

– Schreib-/Lese-Konflikt	$w(x)$	$r(x)$
– Lese-/Schreib-Konflikt	$r(x)$	$w(x)$
– Schreib-/Schreib-Konflikt	$w(x)$	$w(x)$



- Serialisierbarkeit liegt vor, wenn der Abhängigkeitsgraph keine Zyklen enthält



# Praktische Umsetzung der Serialisierbarkeit: Sperren

- Berechnung des Abhängigkeitsgraphen ist für den Korrektheitsnachweis von Synchronisationsverfahren geeignet.
- **Aber:** Berechnung des Abhängigkeitsgraphen ist i.a. nicht geeignet, um Serialisierbarkeit im laufenden Betriebs eines DBMS zu überprüfen
  - **Warum?** die Operationen einer Transaktion sind in der Regel nicht im voraus bekannt ...Was würde passieren?
- **Praktische Umsetzung der Serialisierbarkeit in DBMS: Sperren**
  - Zwei Arten von Sperren
    - **RL**(x) Lesesperre (*read lock* bzw. *shared lock*) auf Objekt x
    - **WL**(x) Schreibsperre (*write lock* bzw. *exclusive lock*)

– Kompatibilitätsmatrix:

		angeforderte Sperre	
		RL	WL
gesetzte Sperre	RL		
	WL		





# Zwei-Phasen-Sperrprotokolle (2 Phase Locking, 2PL)

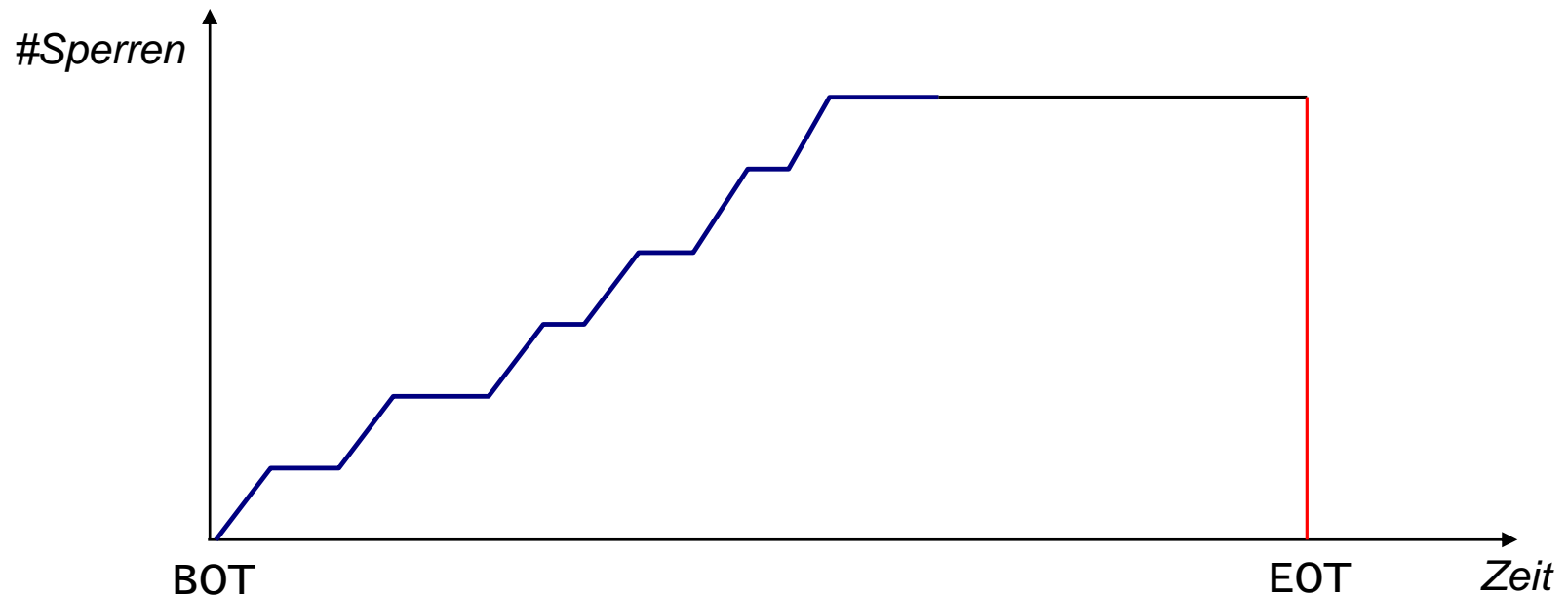
---

- Einhaltung folgender Regeln gewährleistet Serialisierbarkeit:
- Vor jedem Objektzugriff muss Sperre mit ausreichendem Modus angefordert werden
  - Schreibzugriff  $w(x)$  nur nach Setzen einer Schreibsperre  $WL(x)$  möglich
  - Lesezugriffe  $r(x)$  nur nach  $RL(x)$  oder  $WL(x)$  erlaubt
- Gesetzte Sperren anderer Transaktionen sind zu beachten
  - Nur Lesesperren sind „verträglich“
- Zweiphasigkeit:
  - Anfordern von Sperren erfolgt in einer Wachstumsphase
  - Bei EOT werden alle Sperren freigegeben



# Zwei-Phasen Sperrprotokoll: Beispiel

- Zwei-Phasen Sperrprotokoll (2PL)





# Verzahnung zweier TAs gemäß 2PL

- T1 modifiziert nacheinander die Datenobjekte A und B (z.B. eine Überweisung)
- T2 liest nacheinander dieselben Datenobjekte A und B (z.B. zur Aufsummierung der beiden Kontostände).

Zeit	$T_1$	$T_2$	Bemerkung
1	<b>BOT</b>		
2	<b>WLock(A)</b>		
3	read(A)		
4	write(A)		
5		<b>BOT</b>	
6		<b>RLock(A)</b>	$T_2$ muss warten
7	<b>WLock(B)</b>		
8	read(B)		
9	write(B)		
10	<b>commit</b>		
11			$T_2$ wecken
12		read(A)	
13		<b>RLock(B)</b>	
14		read(B)	
18		<b>commit</b>	



# Verklemmungen (*deadlocks*) – 1(2)

- $T_1$  modifiziert nacheinander die Datenobjekte  $A$  und  $B$ .
- $T_2$  liest nacheinander dieselben Datenobjekte  $B$  und  $A$ .

Zeit	$T_1$	$T_2$	Bemerkung
1	<b>BOT</b>		
2	<b>WLock(A)</b>		
3		<b>BOT</b>	
4		<b>RLock(B)</b>	
5		read(B)	
6	read(A)		
7	write(A)		
8	<b>WLock(B)</b>		$T_1$ muss auf $T_2$ warten
9		<b>RLock(A)</b>	$T_2$ muss auf $T_1$ warten
10	...	...	⇒ <b>deadlock</b>

- ⇒ DBMS muss *deadlock* erkennen und eine der beiden Transaktionen abrechen
- verschiedene Strategien, welche Transaktion abgebrochen wird (ältere, jüngere, ...).

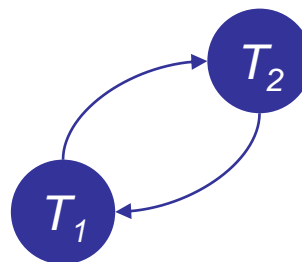


# Verklemmungen (deadlocks) – 2(3)

- Deadlocks können anhand von Wartegraphen erkannt werden
  - T1 modifiziert nacheinander die Datenobjekte A und B.
  - T2 liest nacheinander dieselben Datenobjekte B und A.

→ T1 muss auf T2 warten

→ T2 muss auf T1 warten



⇒ Durch Abbruch von T1 oder T2 kann die Verklemmung aufgelöst werden

- Generell: Zyklenerkennung durch Tiefensuche im Wartegraphen.
- Verschiedene Strategien, welche Transaktion abgebrochen wird (ältere, jüngere, ...).



# Phantom-Problem – Sperren als Lösung?

- *Ein Bonus von 100.000 Euro soll gleichmäßig auf alle Mitarbeiter der Firma verteilt werden.*
- *Parallel dazu wird ein neuer Mitarbeiter eingefügt.*

Zeit	$T_1$	$T_2$
1	<code>select count(*) into x</code>	
2	<code>from Mitarbeiter;</code>	<code>insert into Mitarbeiter</code>
3		<code>values (Meier, 50.000, ...);</code>
4	<code>update Mitarbeiter</code>	<code>commit;</code>
5	<code>set Gehalt = Gehalt + 100.000/x;</code>	
	<code>commit;</code>	

- Sperren?
- Lösung?



# Phantom-Problem – Lösung

---

- Zusätzlich zu den Tupeln muss auch der Zugriffsweg, auf dem man zu den Objekten gelangt ist, gesperrt werden.
- Beispiel:
  - `select count(*) into X from Mitarbeiter;`
    - ⇒ alle *Mitarbeiter* (bzw. deren Primärschlüssel-Index) müssen mit einer *RL*-Sperrung belegt werden
    - ⇒ beim Einfügen eines neuen *Mitarbeiter* wird dies erkannt und  $T_2$  muss warten
- Sperrung kann ggf. auch selektiver sein – z.B.:
  - `select count(*) into X from Mitarbeiter  
where PNr between 1000 and 2000`
    - ⇒ nur die Mitarbeiter mit der entsprechenden PNr müssen gesperrt werden (z.B. Index-Bereich von PNr [1000, 2000])



# Transaktionsverwaltung und Recovery

---

- ✓ Transaktionsbegriff
- Mehrbenutzersynchronisation
  - ✓ Anomalien beim Mehrbenutzerbetrieb
  - ✓ Konzept der Serialisierung
  - ✓ Sperren zur Umsetzung der Isolationseigenschaft von Transaktionen
    - Unterschiedliche Isolation Level
    - Multiversion Concurrency Control
- Fehlerbehandlung





# Isolation Level in SQL92 – 1(4)

```
set transaction
```

```
[read only, | read write,]
```

```
[isolation level
```

```
read uncommitted |
```

```
read committed |
```

```
repeatable read |
```

```
serializable ]
```

Default-  
Werte

*Beispiel (erstes Statement innerhalb der Transaktion!)*

```
set transaction read only, isolation level read committed;  
select ...;  
commit;
```



# Isolation Level in SQL92 – 2(4)

## ***read uncommitted***

- Schwächste Konsistenzstufe. Darf nur für read only-Transaktionen spezifiziert werden! (sonst wäre *lost update* möglich!).
- Eine derartige Transaktion hat Zugriff auf noch nicht festgeschriebene Daten, z.B.:

$T_1$	$T_2$
	read(A)
	...
	write(A)
read(A)	
...	
	<b>rollback</b>

⇒ *dirty read, non repeatable read* und *Phantome* möglich.



# Isolation Level in SQL92 – 3(4)

## ***read committed***

- Solche Transaktionen lesen nur festgeschriebene Werte.
- ⇒ kein *dirty read* möglich. Allerdings kann eine solche Transaktion u.U. unterschiedliche Zustände der Datenbank-Objekte sehen:

$T_1$	$T_2$
read(A)	write(A)
	write(B)
	<b>commit</b>
read(B)	
read(A)	
...	

⇒ *non repeatable read* und *Phantome* möglich.



# Isolation Level in SQL92 – 4(4)

## ***repeatable read***

- *non repeatable read* wird durch diese Konsistenzstufe ausgeschlossen.
- *Phantome* sind in dieser Konsistenzstufe immer noch möglich.

## ***serializable***

- Diese Konsistenzstufe garantiert die Serialisierbarkeit = default.

## ***Zusammenfassung***

Konsistenzebene	Anomalie		
	Dirty Read	Non Repeatable Read	Phantome
Read Uncommitted	+	+	+
Read Committed	-	+	+
Repeatable Read	-	-	+
Serializable	-	-	-

- *lost update* nie möglich!



# Isolation Level in Oracle

## SQL92

```
set transaction
  [read only, | read write,]
  [isolation level
    read uncommitted |
    read committed   |
    repeatable read   |
    serializable]
```

## Oracle

```
set transaction
  [read only, | read write,]
  [isolation level
    read committed |
    serializable]
```

- Isolationsebenen für eine Menge von Transaktionen

```
alter session set isolation_level <isolation_level>
```



# Isolation Level in JDBC

- Isolation Level können durch die Methode `setTransactionIsolation()` der Klasse `Connection` gesetzt werden:
  - `TRANSACTION_NONE`
  - `TRANSACTION_READ_UNCOMMITTED`
  - `TRANSACTION_READ_COMMITTED`
  - `TRANSACTION_REPEATABLE_READ`
  - `TRANSACTION_SERIALIZABLE`

```
Connection con = DriverManager.getConnection(url, user, pwd);
...try {
    con.setTransactionIsolation (TRANSACTION_SERIALIZABLE);
    // ... update ...
}
```

- WICHTIG: Natürlich muss das angesprochene DBMS die entsprechenden Isolation Level unterstützen bzw. geeignet abbilden!
- JDBC bietet Methoden der `DatabaseMetaData` Klasse zur Ermittlung der Eigenschaften des DBMS, u.a.
  - `getDefaultTransactionIsolation( )`
  - `supportsTransactions( )`
  - `supportsTransactionIsolationLevel( )`



# Transaktionsverwaltung und Recovery

---

- ✓ Transaktionsbegriff
- Mehrbenutzersynchronisation
  - ✓ Anomalien beim Mehrbenutzerbetrieb
  - ✓ Konzept der Serialisierung
  - ✓ Sperren zur Umsetzung der Isolationseigenschaft von Transaktionen
  - ✓ Unterschiedliche Isolation Level
    - Multiversion Concurrency Control
- Fehlerbehandlung



# MVCC: Motivation

- Vorbetrachtung
  - $T_1$  liest  $x$  ( $\Rightarrow r_1(x) c_1$ )
  - $T_2$  schreibt  $x$  ( $\Rightarrow w_2(x) c_2$ )
  - $T_1$  und  $T_2$  werden „gleichzeitig“ gestartet
  - Was kann passieren?
- Komplexeres Beispiel:  $r_1(x) w_1(x) r_2(x) w_2(y) r_1(y) w_1(z) c_1 c_2$ 
  - $\Rightarrow$  nicht konfliktserialisierbar ...
  - ... aber: wenn  $r_1(y)$  eine alte Version von  $y$  lesen könnte ...
  - $\Rightarrow$  äquivalent zu  $r_1(x) w_1(x) r_1(y) r_2(x) w_2(y) w_1(z) c_1 c_2$
  - $\Rightarrow$  konfliktserialisierbar!





# MVCC: Idee

---

- ⇒ Multiversionen-Synchronisation (*multiversion concurrency control, MVCC*)
- Prinzip: jede Änderungsoperation *w* erzeugt eine neue Version des geänderten Datenbankobjekts
  - Lesoperationen können nun auf passender („alter“) Version lesen
  - realisiert z.B. in Oracle, PostgreSQL, MS SQL Server (seit V 2005), IBM DB2 (seit V 9.7 2009), diversen analytischen DBMS und verschiedenen NoSQL-DBMS



# MVCC: Vorteile

- Vorteile
  - MVCC führt zur Entkopplung von Lese- und Änderungsoperationen
  - Eine Lesetransaktion hat eine Sicht auf die Datenbank, als ob alle Daten am Beginn der Transaktion atomar gelesen werden
  - Keine Synchronisation gegen Lesetransaktionen notwendig  $\Rightarrow$  Reduktion der Konfliktwahrscheinlichkeit

- Beispiel:

$T_1$	$T_2$	$T_R$
BOT $w(x_0 \rightarrow x_1)$ $w(y_0 \rightarrow y_1)$ commit	BOT $w(x_1 \rightarrow x_2)$ commit	BOT $r(y_0)$         $r(x_0)$

$\equiv T_R T_1 T_2$

Realisierung?



# Transaktionsverwaltung und Recovery

---

- ✓ Transaktionsbegriff
- ✓ Mehrbenutzersynchronisation
  - ✓ Anomalien beim Mehrbenutzerbetrieb
  - ✓ Konzept der Serialisierung
  - ✓ Sperren zur Umsetzung der Isolationseigenschaft von Transaktionen
  - ✓ Unterschiedliche Isolation Level
  - ✓ Multiversion Concurrency Control
- Fehlerbehandlung
  - Fehlerklassen
  - Recovery-Strategien
  - Backup-Varianten



# Fehlerklassifikation

---

## Fehlerklassifikation

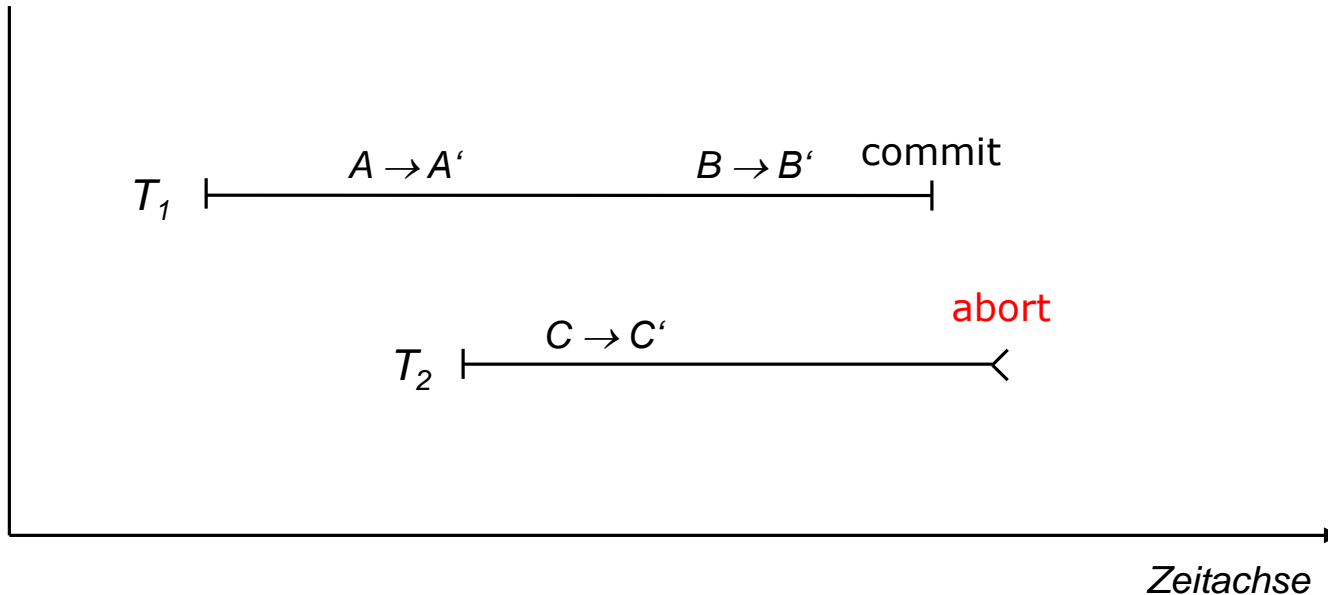
- 1. Transaktionsfehler**
- 2. Systemfehler**
- 3. Externspeicherfehler**

⇒ unterschiedliche Recovery-Maßnahmen je nach Fehlerart



# Transaction-Recovery: Beispiel

- Szenario:



- Transaction-Recovery
  - $C'$  muss wieder auf  $C$  zurückgesetzt werden (*UNDO*).



# Transaktionsfehler ( $\Rightarrow$ Transaction-Recovery)

---

- **Typische Transaktionsfehler**
  - Fehler im Anwendungsprogramm
  - Transaktionsabbruch explizit durch den Benutzer
  - Transaktionsabbruch durch das System
- **Charakteristika**
  - Abbruch einer einzelnen(!) Transaktion
  - kein Einfluss auf den Rest des Systems  $\Rightarrow$  auch: lokaler Fehler
- **Behandlung (Transaction-Recovery)**
  - „Isoliertes“ Zurücksetzen aller Änderungen der abgebrochenen Transaktionen = *lokales UNDO* bzw. *R1-Recovery*
  - Wichtig: von dieser Seite veränderte Blöcke können sich sowohl im Datenbank-Puffer als auch bereits in der materialisierten Datenbank (also auf dem Externspeicher) befinden!



# Protokollierung von Änderungsinformation

---

- Durchgeführte Änderungen werden von den meisten DBMS in einem Log bzw. Log-Dateien protokolliert.
- Ein Log besteht aus Einträgen der Form:  
{ LSN, TA, PageID, Undo, Redo PrevLSN }
  - LSN: Log-Sequence Number = eindeutige und aufsteigende Nummerierung der Log-Einträge
  - TA: Transaktionskennung (Nummer)
  - PageID: Seitennummer
  - Undo: UNDO-Information
  - Redo: REDO-Information
  - PrevLSN: LSN des letzten Eintrags der selben Transaktion
- Außerdem wird Beginn und Ende einer Transaktion vermerkt (BOT, commit, abort)



# Log-Einträge: Beispiel

LSN	TA	PageID	Undo	Redo	PrevLSN
#1	T <sub>1</sub>	BOT			0
#2	T <sub>1</sub>	27	[. A .]	[. A' .]	#1
#3	T <sub>2</sub>	BOT			0
#4	T <sub>2</sub>	40	[. C .]	[. C' .]	#3
#5	T <sub>1</sub>	70	[. B .]	[. B' .]	#2
#6	T <sub>1</sub>	commit			#5





# Transaction-Recovery: Beispiel (Forts.)

## Transaction-Recovery

- alle Log-Einträge von  $T_2$  werden in umgekehrter Reihenfolge ihrer ursprünglichen Ausführung gelesen und rückgängig gemacht, d.h. die Undo-Information (alter Zustand der Seite) in die Datenbank eingebracht.

LSN	TA	PageID	Undo	Redo	PrevLSN
#1	$T_1$	BOT			0
#2	$T_1$	27	[. A .]	[. A' .]	#1
#3	$T_2$	BOT			0
#4	$T_2$	40	[. C .]	[. C' .]	#3
#5	$T_1$	70	[. B .]	[. B' .]	#2
#6	$T_1$	commit			#5
#7	$T_2$	abort			#4



# Systemfehler ( $\Rightarrow$ Crash Recovery)

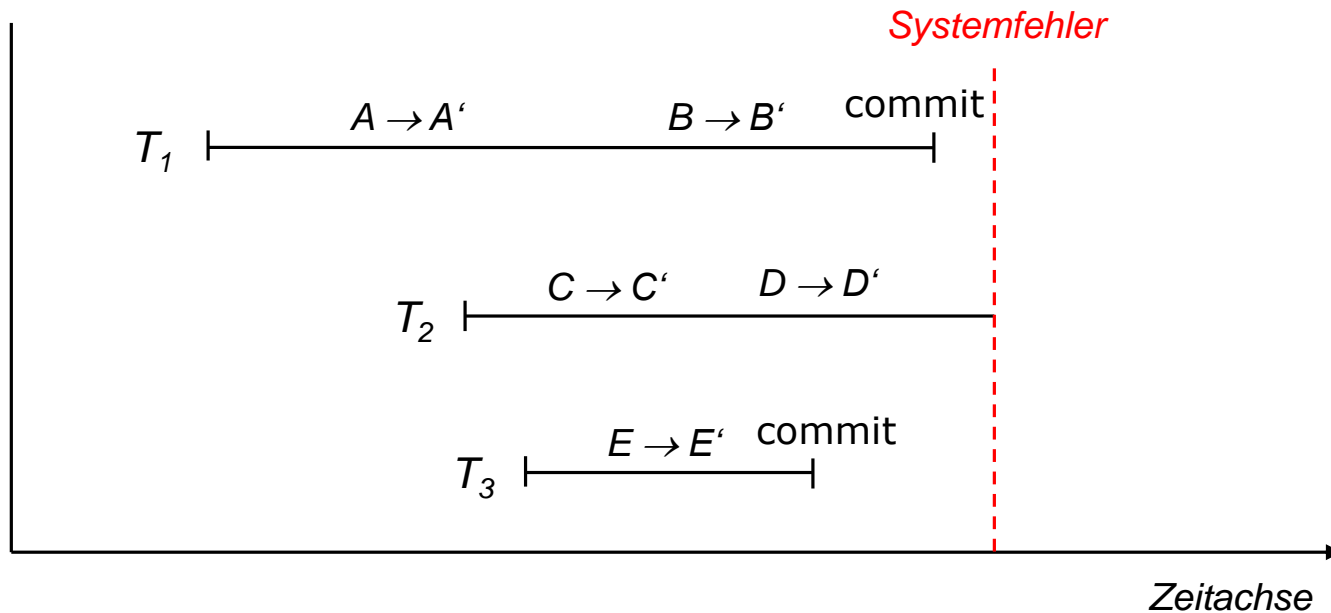
---

- **Typische Systemfehler**
  - DBMS-Fehler
  - Betriebssystemfehler
  - Hardware-Fehler
- **Charakteristika**
  - die im DB-Puffer befindlichen Daten sind zerstört
  - die auf dem Externspeicher befindlichen Daten (also die „materialisierte“ Datenbank) ist jedoch unversehrt!
- **Behandlung (Crash Recovery)**
  - Nachvollziehen der von abgeschlossenen Transaktionen **nicht in die DB eingebrachten Änderungen** = *partielles REDO* bzw. *R2-Recovery*
  - Zurücksetzen der von **nicht beendeten** Transaktionen in die DB eingebrachten Änderungen = *globales UNDO* bzw. *R3-Recovery*



# Crash-Recovery: Beispiel – 1(2)

Szenario:

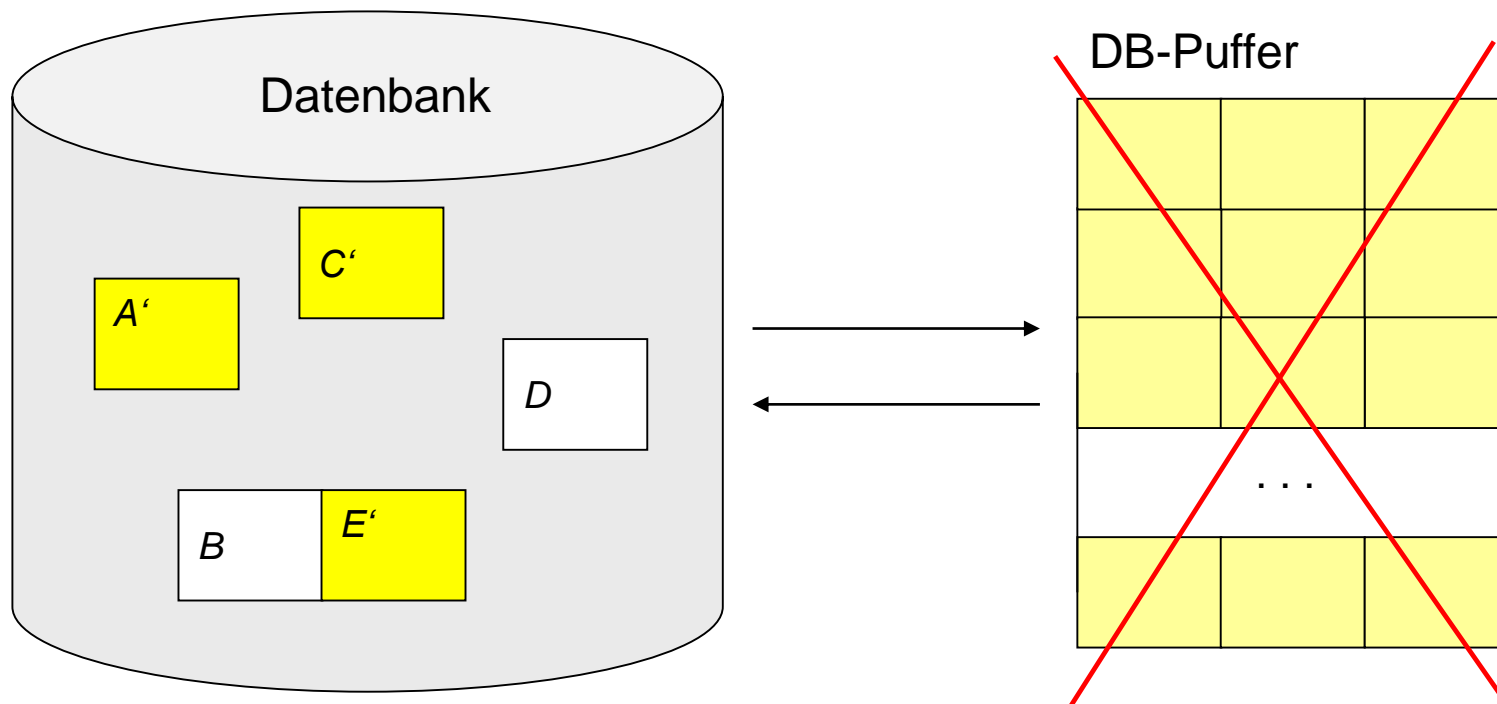




# Crash-Recovery: Beispiel – 2(2)

Situation im Puffer bzw. der Datenbank

- $T_1$  (*committed*) : A' wurde bereits zurück geschrieben; B' nicht(!)
- $T_2$  (*open*) : C' wurde bereits zurück geschrieben(!); D' nicht
- $T_3$  (*committed*) : E' wurde bereits zurück geschrieben





# Crash-Recovery

---

## 3 Phasen

### 1. Analyse

Das Log wird von Anfang bis zum Ende gelesen und ermittelt, welche Transaktionen erfolgreich beendet (committed) wurden und welche zum Fehlerzeitpunkt offen waren.



### 2. Wiederholung der Historie (REDO)

Es werden alle(!) protokollierten Änderungen in der Reihenfolge ihrer Ausführung in die Datenbank eingebracht.



### 3. UNDO

Die Änderungsoperationen der zum Fehlerzeitpunkt offenen Transaktionen werden rückgängig gemacht.





# Crash-Recovery – Phase 1

- Ermittlung aller abgeschlossenen Transaktionen ( $T_1$  und  $T_3$ ) und offenen Transaktionen ( $T_2$ )

Analyse ↓

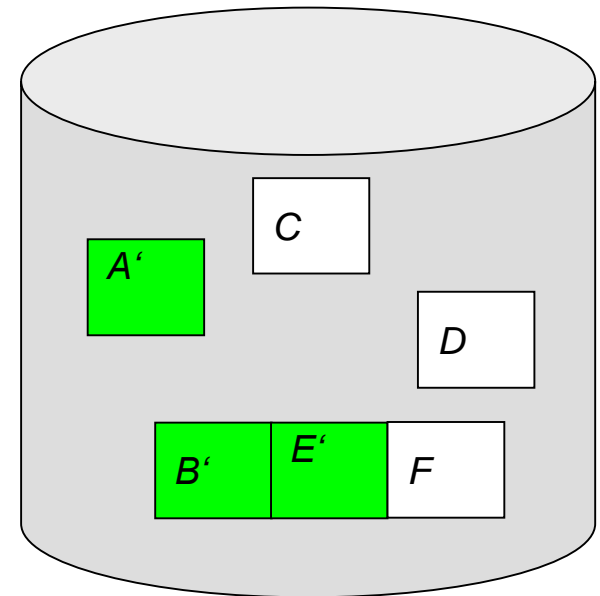
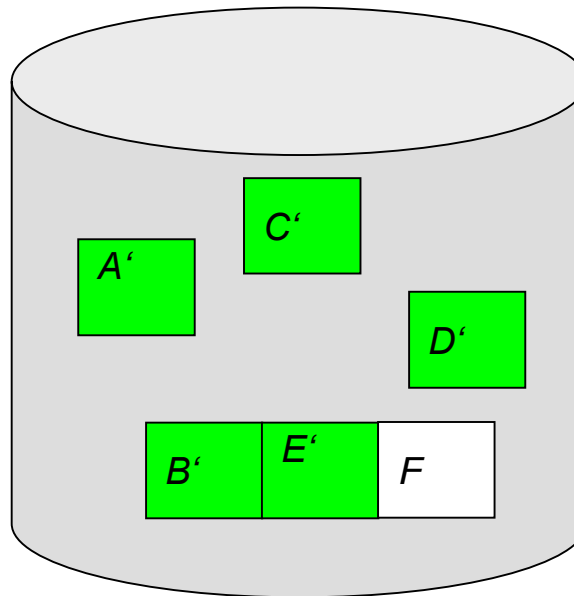
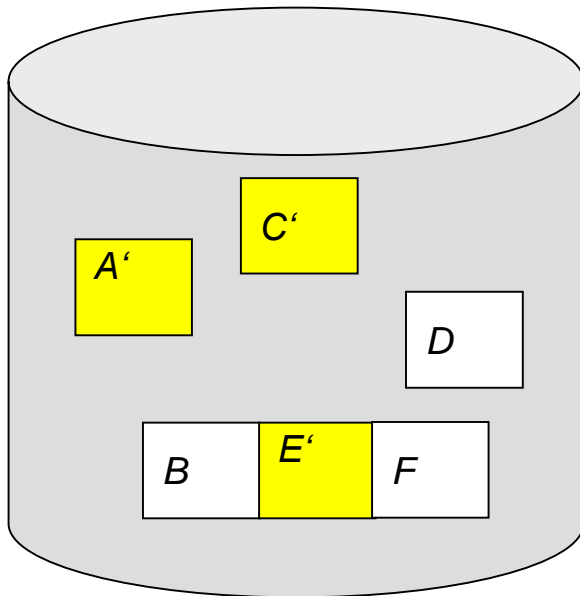
LSN	TA	PageID	Undo	Redo	PrevLSN
#1	$T_1$	BOT			0
#2	$T_1$	27	[. A .]	[. A' .]	#1
#3	$T_2$	BOT			0
#4	$T_2$	40	[. C .]	[. C' .]	#3
#5	$T_3$	44	[. E .]	[. E' .]	0
#6	$T_3$	commit			#5
#7	$T_2$	43	[. D .]	[. D' .]	#4
#8	$T_1$	70	[. B .]	[. B' .]	#2
#9	$T_1$	commit			#8



# Crash-Recovery – Phase 2 und 3

**Phase 2**  
Wiederholung der  
Historie (REDO)

**Phase 3**  
UNDO der offenen  
Transaktionen



Datenbankzustand nach  
dem Systemfehler



# Write-Ahead-Log-Prinzip (WAL)

---

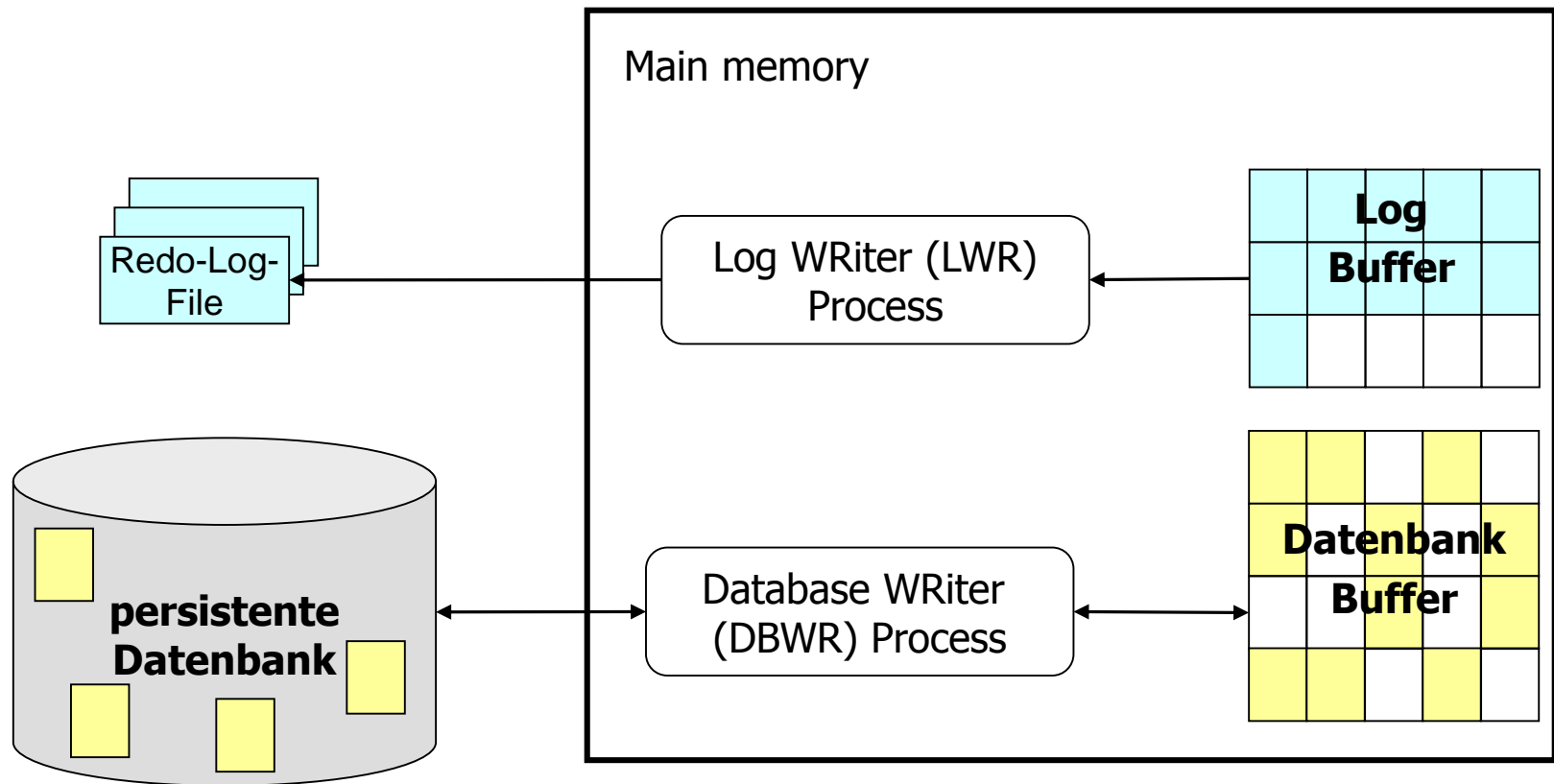
- Wir haben in den Beispielen vorausgesetzt, dass die für die Recovery benötigte Information im Log steht (trotz Verlust der Hauptspeicherinformation) – ist das gewährleistet?
- ⇒ Bevor eine Transaktion festgeschrieben (*committed*) wird, müssen alle zu ihr gehörenden Log-Einträge ausgeschrieben werden (für das REDO im Fehlerfall).
- ⇒ Bevor eine veränderte Seite in die Datenbank eingebracht wird, müssen alle diese Seite betreffenden Log-Einträge ausgeschrieben werden (für das UNDO im Fehlerfall).
- Diese beiden Forderungen werden als *Write-Ahead-Log-Prinzip (WAL)* bezeichnet.
- Anmerkung: Natürlich werden dabei nicht einzelne Log-Einträge, sondern alle Log-Einträg bis zum betroffenen sequentiell ausgeschrieben.





# Prozessarchitektur (Ausschnitt)

- Typische Prozessarchitektur für das Ausschreiben des Datenbank- und des Log-Puffers:





# Externspeicherfehler ( $\Rightarrow$ Media-Recovery)

---

- **Typische Externspeicherfehler**
  - Hardware-Fehler: „Head-Crashes“, Controller-Fehler etc.
  - Naturgewalten wie Feuer oder Erdbeben
  - Viren
- **Charakteristika**
  - Die Daten der materialisierten Datenbank sind zerstört oder unbrauchbar
- **Behandlung (Media-Recovery)**
  - Die Datenbank muss mit Hilfe einer Sicherungskopie (Backup) wiederhergestellt werden.
  - Danach muss der letzte transaktionskonsistente Zustand wiederhergestellt werden, d.h. es alle seit der Erstellung des Backup erfolgreich beendeten Transaktionen nachvollzogen werden.
  - $\Rightarrow$  Konsequenz: Die Log-Dateien müssen auf einem anderen Medium gesichert werden (z.B. anderer Rechner, Magnetband o.ä.)!



# Media-Recovery

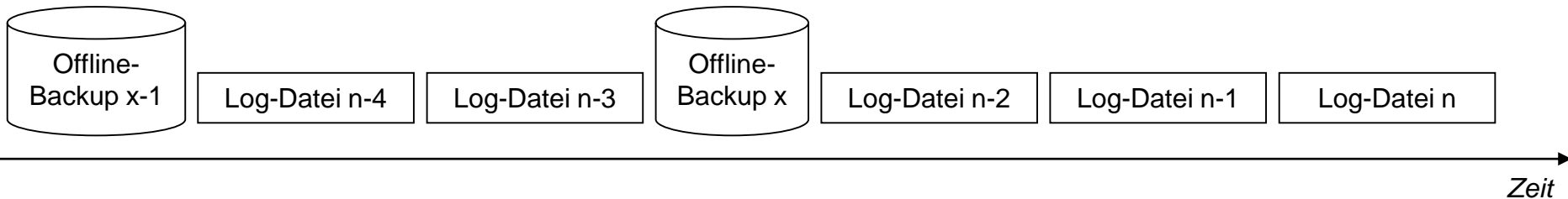
---

- Wichtige Unterscheidung: In welchem Zustand ist die Datenbank bei der Sicherung?
- **Variante 1:** Es sind keine Transaktionen auf der Datenbank während der Sicherung aktiv = **Offline-Backup** (*consistent backup*)
  - Vorteil: Datenbankkopie ist in transaktionskonsistentem Zustand!
  - Nachteil: Während der Sicherung darf keine (Schreib-)Transaktion auf der Datenbank aktiv sein! (vielfach nicht akzeptabel, z.B. 24h Betrieb im Internet bzw. bei weltweit agierenden Unternehmen)
- **Variante 2:** Es können Transaktionen auf der Datenbank während der Sicherung aktiv sein = **Online-Backup** (*inconsistent backup*)
  - Vorteil: Datenbankbetrieb wird nicht (oder kaum) beeinträchtigt
  - Nachteil: Datenbankkopie ist nicht in transaktionskonsistentem Zustand – Wiederherstellung (Recovery) aufwändiger
- Kommerzielle DBMS unterstützen heute meist(!) beide Varianten.



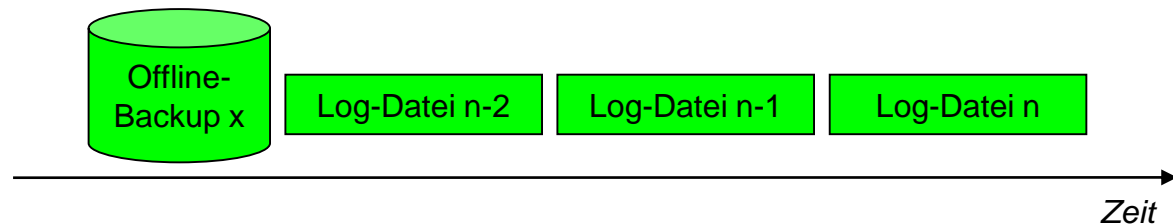
# Media-Recovery: Grundprinzip – 1(2)

- Gesicherte Daten (Offline-Backup):



- **Media-Recovery**

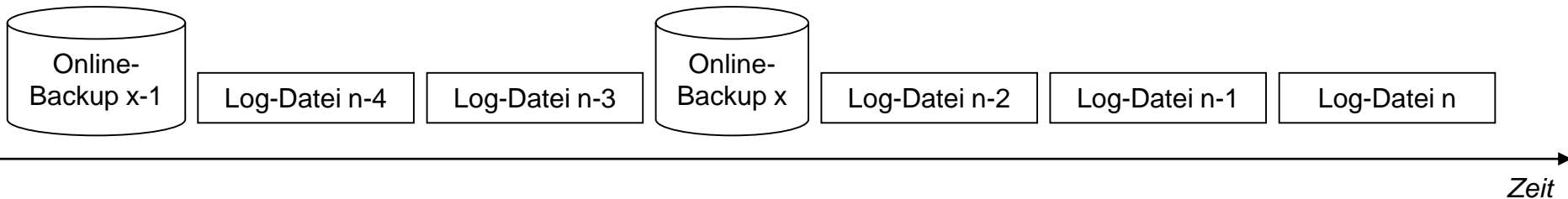
- Letzte Sicherungskopie der Datenbank wird eingespielt
- Nach der letzten Sicherung erstellte Log-Dateien werden analysiert und die Änderungen erfolgreich beendeter Transaktionen nachvollzogen (REDO)  
(Bemerkung: aus Performance-Gründen kann auch eine Vorgehensweise analog zur Crash-Recovery gewählt werden: komplette Historie wiederholen und danach Undo der Änderungen offener Transaktionen.)





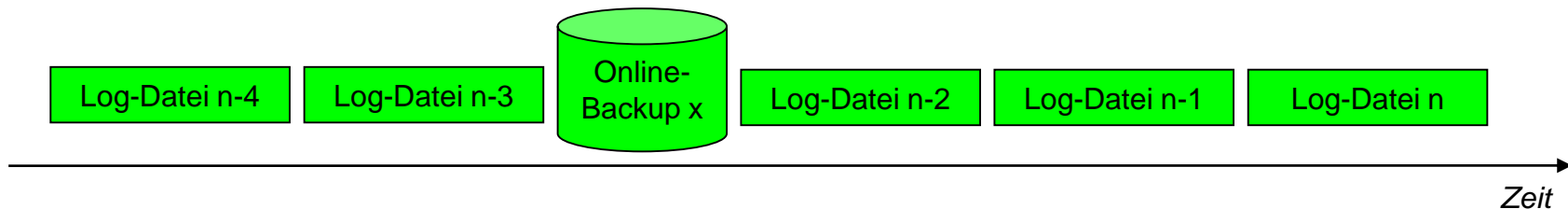
# Media-Recovery: Grundprinzip – 2(2)

- Gesicherte Daten (Online-Backup):



- **Media-Recovery**

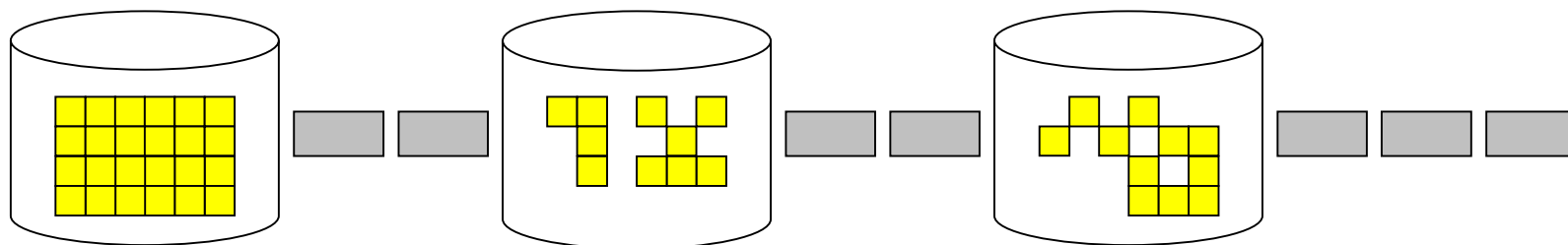
- Letzte Sicherungskopie der Datenbank wird eingespielt
  - Es müssen auch Log-Dateien vor dem letzten Backup betrachtet werden, da die Sicherungskopie u.U. Änderungen von später nicht erfolgreich beendeten Transaktionen enthält. Die Undo-Information dieser Transaktionen wird benötigt.
- ⇒ Recovery zeitaufwändiger und Administration (Aufbewahren der Log-Dateien) aufwändiger



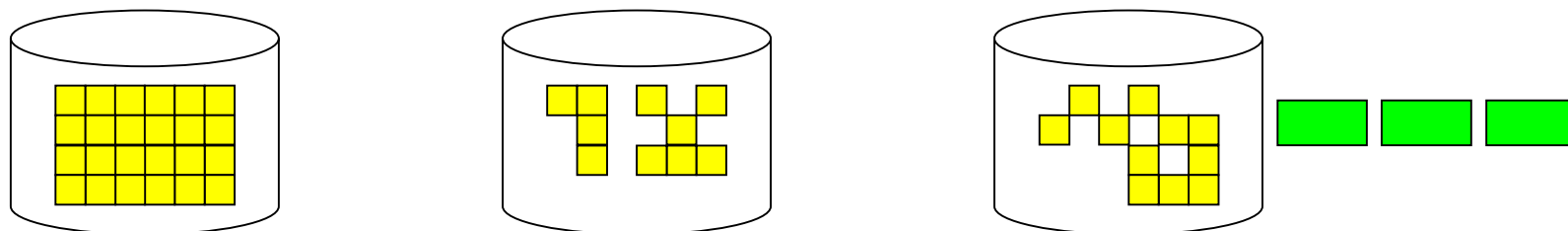


# Inkrementelles Backup

- Bei großen Datenbanken ist eine Komplettsicherung sehr(!) zeitaufwändig
- ⇒ Sicherung der veränderten Datenbankseiten = inkrementelles Backup



- Bei der Wiederherstellung wird
  - das letzte Full Backup und alle danach erstellten inkrementellen Backups eingespielt und danach
  - nur die Log-Dateien (Annahme: offline Backup) nach dem letzten inkrementellen Backup angewandt:





# Weitere Backup-Varianten

---

Die bereits aufgeführten Backup-Varianten

- Online vs. Offline Backup
- Komplettes Backup vs. inkrementelles Backup

können orthogonal mit weiteren Backup-Varianten kombiniert werden:

- Partielles Backup (Backup von Teilen der Datenbank z.B. Tablespaces)
- Paralleles Backup



# Backup und Recovery

---

- Backup- und Recovery-Kommandos in Datenbanksystemen sind nicht standardisiert.
- Es gibt eine Vielzahl von Parameter (Ausgabekanäle, Parameter für Größe von Log-Dateien und Häufigkeit der Sicherung, Checkpoint-Parameter etc.)
- Die Wahl dieser Parameter und der Sicherungsstrategie hat erhebliche(!) Auswirkungen sowohl auf die Performance im laufenden Betrieb als auch auf die Wiederherstellungszeit im Fehlerfall.
- Wichtig: Spiegelung bzw. RAID-Systeme sind kein ausreichender Ersatz für regelmäßige Backups! Warum?





# Zusammenfassung

---

- Anomalien bei Mehrbenutzerbetrieb
- ACID-Eigenschaften von Transaktionen
- Sperren zur Umsetzung der Isolationseigenschaft von Transaktionen
- Unterschiedliche Isolation Level
- Fehlerbehandlung



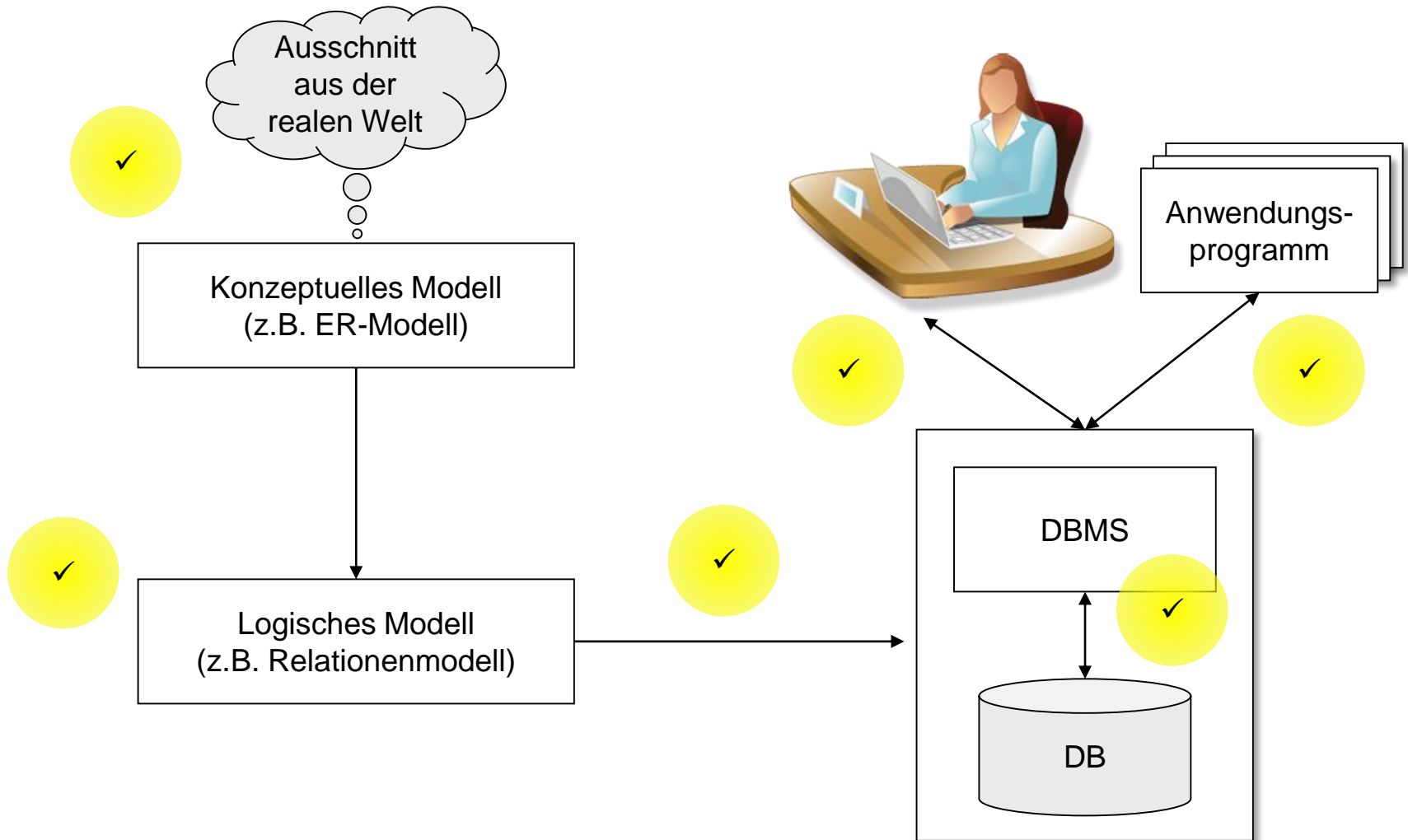
# Architektur von Datenbanksystemen

---

- ✓ 3-Ebenen-Architektur von Datenbanken
  - ✓ Externe Ebene
  - ✓ Konzeptionelle Ebene
  - ✓ Interne Ebene
  
- DBMS-Systemarchitektur
  - ✓ Schichtenarchitektur
  - ✓ Transaktionsverwaltung und Recovery

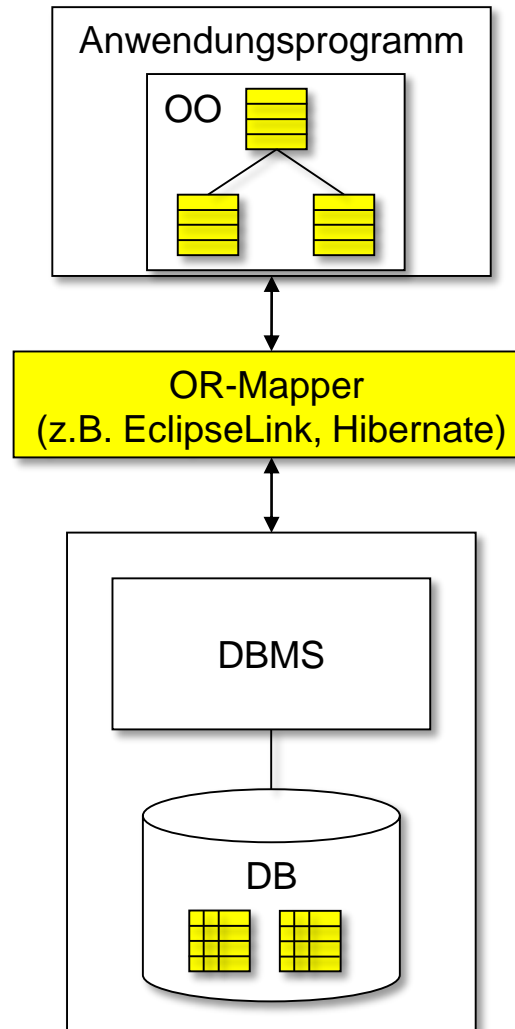


# Vorlesung Datenbanken 1





# Ausblick Datenbanken 2





# Ausblick

- Wahlpflichtfächer mit Datenbank-Schwerpunkt:

Mobile Datenbanken  
(Erbs)

Objektorientierte und  
OR DB (Erbs)

Data Warehouse und  
OLAP (Karczewski)

Pflichtbereich

Datenbanken 2

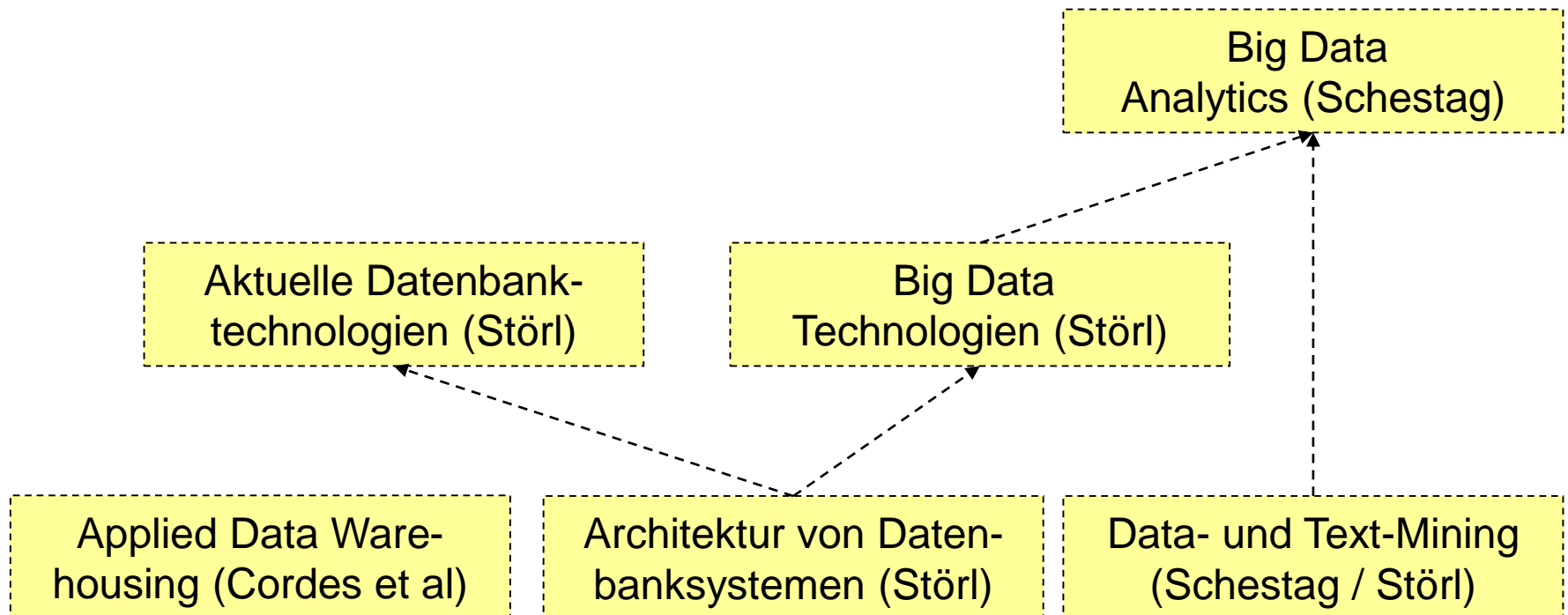


Datenbanken 1



# Ausblick Masterstudiengang

- Auch im Masterstudiengang gibt es eine Vielzahl von Angeboten mit Datenbank-Schwerpunkt:





# Masterstudiengang Data Science

- Neu: Ab Wintersemester 2016/17
- **Gemeinsamer Studiengang** des Fachbereichs Mathematik und Naturwissenschaften und des Fachbereichs Informatik
- **Schwerpunkt:** Bearbeitung mathematisch, statistisch und informatisch anspruchsvoller Probleme sowohl aus der Praxis als auch aus der anwendungsorientierten Forschung einzusetzen – mit Fokus auf Data Science und Big Data.
- **Zielgruppe:** Bachelor-Absolventen aus den Bereichen Mathematik bzw. Informatik, die eine ausgeprägte Affinität zur Informatik, speziell Computing und Datenbanken, bzw. zur Statistik und Angewandten Mathematik mitbringen.

## **Infoveranstaltungen:**

Donnerstag, 23.06. um 16.30 Uhr, C10, Raum 03.33 bzw.  
Freitag, 24.06. um 15.00 Uhr, D14/013

- Weitere Informationen:

<http://fbmn.h-da.de/DataScience>

Ansprechpartner am Fachbereich Informatik: Prof. Dr. Arnim Malcherek

<https://www.fbi.h-da.de/organisation/personen/malcherek-arnim.html>