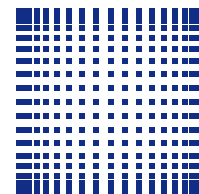


Datenmanagement

Sven Klaus
s.klaus@hs-mannheim.de



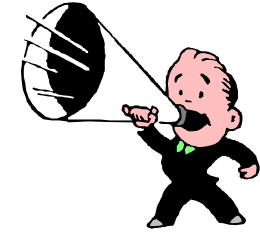
Informatik
3. Semester



hochschule mannheim



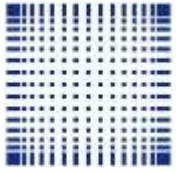
HEUTE



- Vorstellung
- Organisatorisches
- Vorlesungs-Übersicht

Was ist eigentlich ...

- ... Information?
- ... Redundanz, Inkonsistenz?



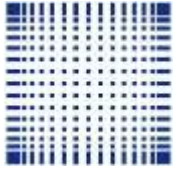
Zu meiner Person:

***Prof. Dr. Sven Klaus
Fakultät für Informatik
Gebäude A, Raum A007a***

E-Mail: s.klaus@hs-mannheim.de

Twitter: <https://twitter.com/SvenKlaus>

Facebook: <https://www.facebook.com/sven.klaus>



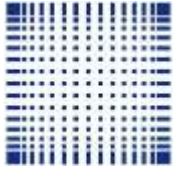
Interaktion?

UNBEDINGT erwünscht!



***Wer hatte schon mit
Datenbanken „zu tun“?***

Mit welchen?



***Wer hat schon (einmal)
Datenbanken eingesetzt?***

Wofür?

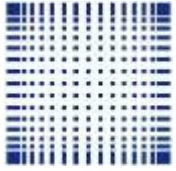


Wer hat schon mit Datenbanken programmiert?



Wer kennt SQL?

***Wer kennt MS Access
oder MySQL?***



Langjährige Erfahrung beweist ...

Es gibt nur diese drei ABSOLUTEN Wahrheiten:

- (1) Alles, was in der Prüfung dran kommt, steht auf den Folien.
- (2) Der Professor hat den Auftrag uns etwas beizubringen.
- (3) Die Erde ist eine Scheibe.

***Daher immer die folgenden BEST PRACTICE
Ratschläge beachten ...***

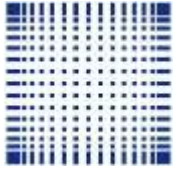


Best Academic Practice

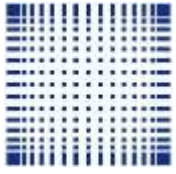
- Niemals Notizen anfertigen! Die liegen später eh nur rum.
- Bloß nicht zuhören! Es geht doch nur darum, vom Professor gesehen zu werden.
- Zu spät kommen! Aufmerksamkeit erhöht den Bekanntheitsgrad und die Note beim Professor.
- Wenn ein Professor empfiehlt sich etwas anzusehen, warten bis er es dann doch selbst erzählt!
- Mehr als eine Nacht vor der Prüfung sich den Lernstoff anzusehen, bringt rein gar nichts!
- Eine Stunde nach der Vorlesung die Folien einklagen!
- Fristen sind dazu da, überschritten zu werden!
- In den letzten 1-2 Monaten wird sowieso nur wiederholt!



- Sie erwarten von mir Pünktlichkeit.
- Sie erwarten von mir während der Vorlesung, dass ich meine gesamte Aufmerksamkeit Ihnen zuwende.
- **Dann darf ich das von Ihnen ebenfalls erwarten.**
- **Bei zu spät kommen:** Leise und zügig hinsetzen. Bei mehr als 15 Minuten zu spät hat sich diese Vorlesung definitiv erledigt.
- **Gespräche:** werden von mir mit einem nervenden „Haben Sie Fragen?“ torpediert.
- **Der Bereich rund um Pult und Tafel gehört mir.**



***Haben Sie bitte
Respekt vor Ihren
Kommilitonen, die hier
lernen wollen!***



Wir haben Datenbanken (DBA):

■ 2 Doppelstunden Vorlesung:

◆ **Bis zum 2017-11-14**

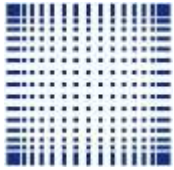
**Dienstags, 12:00-13:30 Uhr, Gebäude A,
Raum A008**

◆ **Donnerstags, 15:20-16:50 Uhr, Gebäude A,
Raum A206**

■ Übungen bzw. Laborstunden:

◆ **Ab dem 2017-11-21: Dienstags, 12:00-13:30 Uhr,
Gebäude A, Raum A008**

◆ Praxisarbeit mit Microsoft Access 2013, MySQL,
JDBC und Mongo DB



Material zur Vorlesung:

- Gibt es in Folienform – also nicht als Skript, immer nach der Vorlesung auf der Lernplattform:
<http://moodle.hs-mannheim.de/>
Alle Kurse – Fakultät für Informatik – Prof. Sven Klaus
Kurs: [DM, 3IB] Datenmanagement
- Dort finden Sie noch weitere Materialien.
- **Folienskript ≠ Buch:**
 - ◆ Erarbeite Vorlesungsbeispiele an der Tafel **fehlen!**
 - ◆ Viele Sachverhalte ergeben nur im Zusammenhang mit den verbalen Erklärungen Sinn
 - ◆ Also: **Notizen anfertigen!**



Ein Wort zum Aufwand

- Das Modul DM wird im RGS mit 4 SWS und 5 ECTS gelistet.
- $5 \text{ ECTS} \equiv 5 * 30\text{h} = 150\text{h}$ Aufwand (im Schnitt) im Semester
- $14 \text{ Wochen} * 3\text{h Vorlesung pro Woche} = 42\text{h Vorlesung}$
- Der Rest (108h im Semester bzw. **8h jede Woche!**) verteilt sich auf wöchentliche Nachbearbeitung, Lernen für die Klausur und ...
... **Eigenstudium!**
- ... oder neudeutsch: **kompetenzorientiertes Lernen**
- Wo finde ich **kompetenzorientiertes Lernen** in der Lehrveranstaltung Datenbanken?
 - ◆ Kapitel MS Access
 - ◆ drei Übungen zu MS Access
 - ◆ Kapitel SQL
 - ◆ ggf. weitere Themen nach Ansage in der Vorlesung



- Für die Zulassung zur Modulprüfung ist die erfolgreiche Erbringung der Studienleistung DM erforderlich.
- Eine in einem früheren Semester erbrachte Studienleistung DM behält ihre Gültigkeit. **Sie müssen in diesem Falle nicht aktiv werden**, die Studienleistungen werden von der Fakultät erfasst.
- Die Studienleistung wird in Form schriftlich zu beantwortender Aufgaben durch jeweils zwei Studierende nach dem Kapitel 7 – Normalformen in Moodle online gestellt.
- Der Bearbeitungszeitraum umfasst zwei Wochen und die Abgabe hat pro Gruppe mit einem PDF zu erfolgen.



- **Problem:** Es gibt entweder Informatik-Hardcore oder seichte Datenbankmanagementsystem Bedienungsanleitungen
- A. Heuer, G. Saake: Datenbanken Konzepte und Sprachen, mitp, etwa 40 €
- R. A. Elmasri, S. B. Navathe: Grundlagen von Datenbanksystemen, Pearson Studium Verlag, etwa 35 €



Roter Faden Vorlesungsübersicht – 1

Kapitel 1: Allgemeine Begriffe

- ◆ Ganz viele Fremdwörter und deren Bedeutung, damit wir die gleiche Sprache sprechen

Kapitel 2: MS Access, Tabellen und Formulare

- ◆ Der erste Kontakt, die erste Übung
- ◆ Wir versuchen Access wie Excel zu benutzen

Kapitel 3: Persistente Datenhaltung

- ◆ Worin unterscheiden sich Datenbanken von Excel?
- ◆ Wann Datenbanken, wann Excel einsetzen?

Kapitel 4: Entity-Relationship-Modell

- ◆ Wie plant man eine Datenbank vorher?
- ◆ Wie bildet man die Realität in Daten ab?



Roter Faden Vorlesungsübersicht – 2

Kapitel 5: Übertragung in das Relationale Modell

- ◆ Wie kann ich den im letzten Kapitel erstellten Plan auf Papier in Access umsetzen?

Kapitel 6: MS Access, Beziehungen und Abfragen

- ◆ Einfache Abfragen auf nur einer Tabelle
- ◆ Komplexe Abfragen über mehrere Tabellen

Kapitel 7: Normalformen

- ◆ Ein weiterer Weg, wie man zu einer „guten“ Datenbank kommen kann
- ◆ Parallelen und Unterschiede zu Kapitel 4

Kapitel 8: Structured Query Language

- ◆ Dieses Kapitel ist im Eigenstudium zu bearbeiten



Roter Faden Vorlesungsübersicht – 3

Kapitel 9: Grundlagen der Abfragesprachen

- ◆ Welche Mengenoperationen stecken hinter SQL?

Kapitel 10: Transaktionen

- ◆ Wie funktionieren Geldüberweisungen zwischen unterschiedlichen Banken

Kapitel 11: Java Database Connectivity

- ◆ Einbettung der Datenbankprogrammierung in „normale“ Java Applikationen

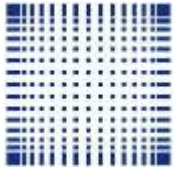
Kapitel 12: Big Data und Mongo DB

- ◆ Zum Abschluss noch der Blick in die Zukunft: Was steckt hinter dem Hype?

Addendum: Zusammenfassung des Relationalen Modells



Welche Fragen gibt es?

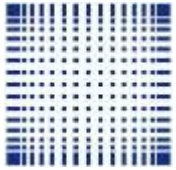


Damit Sie gewarnt wurden 😊

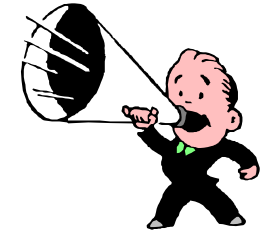
JA – Die Antworten auf die Fragen am Ende jedes Kapitels fehlen mit Absicht.

JA – In diesem Skript tauchen Fachbegriffe auf.

JA – Bei Datenmanagement lernen Sie sehr viel Theorie.



JETZT



Kapitel 1

Allgemeine Begriffe



- Keine eindeutige Definition
- **Information**
 - ◆ stellt Wissensgewinn dar
 - ◆ ermöglicht Handlungen
 - ◆ erklärt Sachverhalte
 - ◆ **umfasst eine Nachricht mit Ihrer Bedeutung für den Empfänger**
- Intuitive Vorstellung: das „Ding“ in der Realität



Was will man mit Daten alles „anstellen“?

- **C – Create** – Daten anlegen (erfassen)
 - **R – Read** – Daten wieder(auf)finden
 - **U – Update** – Daten bearbeiten, erweitern
 - **D – Delete** – (nicht benötigte) Daten löschen
-
- Verhindern von Unstimmigkeiten, z.B. durch mehrfach gespeicherte Daten
 - Mehrere Benutzer



- **Redundanz** bezeichnet allgemein einen Zustand von Überschneidung oder Überfluss.
- **Dubletten** sind Datensätze in einer Datenbank, die unabsichtlich redundant sind, deren Redundanz aber aufgrund von kleinen Abweichungen nicht geprüft werden kann.
- **Inkonsistenz** ist ein Zustand, in dem zwei Dinge, die beide als gültig angesehen werden sollen, nicht miteinander vereinbar sind.



- Zusammenfassung aller vorhandenen Daten einer (definierten) Modellwelt
- Eine Datenbank kann zusätzlich Metadaten enthalten über die gespeicherten Daten
- **Datenbank (DB)**
 - ◆ strukturierte und logisch organisierte Sammlung (formatierter) Daten,
 - ◆ die dauerhaft und weitgehend redundanzfrei gespeichert wird.



■ Datenbank-Managementsystem

- ◆ DBMS
 - ◆ ist ein universelles Softwaresystem,
 - ◆ das anwendungsunabhängiges Arbeiten mit Datenbanken ermöglicht
-
- Ein DBMS ermöglicht Daten zu speichern, wieder aufzufinden, um sie zu lesen oder zu ändern.



■ Datenbanksystem

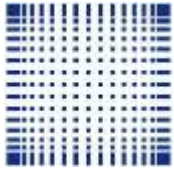
- ◆ Abkürzung: DBS
- ◆ Ist die Zusammenfassung von
- ◆ einer oder mehreren Datenbanken und
- ◆ einem Datenbankmanagementsystem



- Die in einer DB gespeicherten Daten heißen:
 - ◆ **Datensätze, Records, Tupel, Zeile, Objekte**
 - ◆ **Tabelle, Relation** mit eindeutigem **Tabellennamen, Relationsnamen**
- Die möglichen Operationen auf der Tabelle werden durch das **Daten(bank)modell** festgelegt
- Die Datensätze bilden eine Menge
- Die Zusammenfassung aller Tabellen zu einer Datenbank heißt **Datenbankschema**, eine (logisch zusammengehörende) Untermenge **Teilschema**



- Alle Datensätze einer Tabelle haben den gleichen Aufbau – eine feste Anzahl **Spalten**.
- Jede Spalte legt eine spezielle **Eigenschaft** (**Attribut** mit **Attributnamen**) fest.
- Jedes Attribut verfügt über eine Menge möglicher Werte aus einem **Wertebereich (Domaine)**. Alle Einträge einer Spalte sind vom gleichen **(Daten)Typ**.
- Ein oder mehrere (ggf. auch künstliches) Attribut, dass die Datensätze eindeutig unterscheidet bezeichnet man als **Schlüssel**.



Beispiel – 1

Spieler Thomas



| Runde | Buchstabe | Stadt | Land | Fluß | Punkte |
|-------|-----------|-----------|------------|--------|--------|
| 1 | W | Wolfsburg | | Wupper | 15 |
| 2 | K | Kempten | Kongo | | 15 |
| 3 | B | Berlin | Belgien | | 10 |
| 4 | A | Aachen | Australien | Aller | 25 |



Motivation zum Einsatz von DBS

Typische Probleme bei Informationsverarbeitung ohne DBS

- Redundanz und Inkonsistenz
- Beschränkte Zugriffsmöglichkeiten
- Probleme beim Mehrbenutzerbetrieb
- Verlust von Daten
- Integritätsverletzung
- Sicherheitsprobleme
- hohe Entwicklungskosten für Anwendungsprogramme



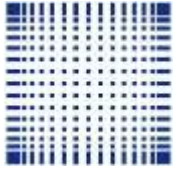
Anforderungen an ein DBS – 1

- **Effizienz:** schneller Zugriff auch bei vielen Daten
- **Kompaktheit:** gleiche Daten nur einmal speichern
- **Integrität:** keine Widersprüche z.B. bei mehrfach gehaltenen Daten
- **Persistenz:** Lebensdauer der Daten ist unabhängig vom kreierenden Prozess
- **Modellierbarkeit:** mit dem DBS kann nicht nur eine bestimmte Problemgattung modelliert werden



Anforderungen an ein DBS – 2

- **Programmierbarkeit:** Zugriff auf die Daten aus einer höheren Programmiersprache heraus; Abfragesprache
- **Parallelität:** Mehrbenutzerbetrieb ohne Inkonsistenzen
- **Transaktions-Orientiertheit:** Mehrere Operationen als zusammenhängend betrachten
- **Datenschutz:** Zugriffskontrollen
- **Fehler- und Ausfallsicherheit:** Wiederherstellung nach Problemen



Zum Nachdenken:

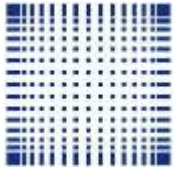
Was ist Redundanz?

Was ist Inkonsistenz?

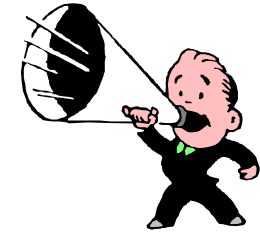
Wie hängen diese Begriffe zusammen?



Welche Fragen gibt es?



JETZT



Kapitel 2

Microsoft Access

1. Tabellen, Formulare



Objektarten in Access – 1

■ **Tabelle**

Hier (und nur hier) werden die Daten gespeichert. Keine andere Objektart kann das

■ **Abfrage**

Sucht in Tabellen nach bestimmten Daten, die entweder angezeigt oder weiterverwendet werden

■ **Formular**

Dient zur (komfortableren) Eingabe der Daten

■ **Bericht**

Dient zur Anzeige, Auswertung und zum Drucken von Daten. Eingaben sind nicht möglich, dafür kann hier übersichtlich gruppiert, aufsummiert etc. werden



Objektarten in Access – 2

■ Seite

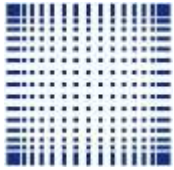
Diese Objektart stellt eine Datenzugriffsseite (dynamische HTML-Seite) dar

■ Makro

Enthält einen oder mehrere Befehle, die beim Starten des Makros ausgeführt werden

■ Modul

Enthält Programmcode (Funktionen und Prozeduren) in der Programmiersprache Visual Basic for Applications (VBA). Module werden mit bestimmten Ereignissen verknüpft (z.B. dem Drücken einer Schaltfläche in einem Formular) und dann ausgeführt



Datentypen in Access – 1

- **Zahl**
zur Speicherung numerischer Werte
je nach Feldgröße: 1,2,4 oder 8 Byte
- **Kurzer Text (früher: Text)**
für Texteingaben aller Art und Zahlen, die nicht zum Rechnen
geeignet sind
maximal 255 Zeichen
- **Langer Text (früher: Memo)**
für längere Texte, Beschreibungen, Erklärungen zum Datensatz
maximal 64 KByte
- **Datum/Zeit**
Datums- und Zeitwerte; die Datumsrechnung beginnt im Jahre 100
und endet 9999
8 Byte
- **Währung**
Geldbeträge mit maximal vier Nachkommastellen Genauigkeit
(Festkommadarstellung)
8 Byte



z.B.
PLZ



Datentypen in Access – 2

- **AutoWert**

Für automatisch hoch gezählte Nummern
4 Byte

- **Ja/Nein**

Nimmt nur den Wert Ja oder Nein an
1 Bit

- **OLE-Objekt**

Bilder und Objekte aus anderen Programmen
maximal 1 GByte

- **Hyperlink**

Texte, die als Hyperlink-Adresse verwendbar sind
maximal 64 KByte

- **Nachschlage-Assistent**

Erstellt eine Auswahlliste mit eingegebenen Werten oder Werten aus anderen Tabellen
Feldgröße der Werte bzw. der anderen Tabelle



- **Byte** (1 Byte)
Wertebereich: 0 bis 255
- **Integer** (2 Byte)
Wertebereich: -32.768 bis 32.767
- **Long Integer** (4 Byte)
Wertebereich: $-2.147.483.648$ bis $2.147.483.647$
- **Single** (4 Byte)
Wertebereich: $-3,4 \cdot 10^{38}$ bis $3,4 \cdot 10^{38}$
- **Double** (8 Byte)
Wertebereich: $-1,797 \cdot 10^{308}$ bis $1,797 \cdot 10^{308}$



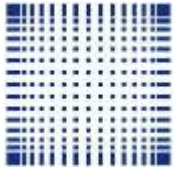
Wichtigste Feldeigenschaften – 1

- **Feldgröße:** Anzahl der Zeichen, die maximal in das Feld eingegeben werden können
- **Format:** späteres Anzeigeformat für den Feldinhalt
- **Eingabeformat:** definiert ein Muster für alle Daten, die später in das Feld eingegeben werden können
- **Dezimalstellen:** Festlegung der Dezimalstellen bei Zahlen und Währungsfeldern
- **Beschriftung:** Feldbezeichnung in Tabellen, Berichten und Formularen
- **Standardwert:** Vorbelegung eines Feldes bei neuen Datensätzen mit einem Defaultwert



Wichtigste Feldeigenschaften – 2

- **Gültigkeitsregel:** Regel zur Überprüfung einer korrekten Eingabe
- **Gültigkeitsmeldung:** auszugebende Fehlermeldung bei Verletzung der Gültigkeitsregel
- **Eingabe erforderlich:** Pflichteingabe in das Datenfeld beim Ausfüllen eines Datensatzes
- **leere Zeichenfolge:** Akzeptanz einer leeren Zeichenkette (zwei direkt aufeinander folgende doppelte Anführungszeichen)
- **indiziert:** ermöglicht die Unterbindung doppelter Datenwerte bzw. beschleunigt die Datensuche über diesen Feldwert



Wichtigste Eingabeformate

- 0 Platzhalter für eine Ziffer 0..9, Eingabe erforderlich
- 9 Platzhalter für eine Ziffer oder Leerzeichen, Eingabe optional
- # Platzhalter für eine Ziffer, ein Leerzeichen oder ein Vorzeichen (+/-), Eingabe optional
- L Platzhalter für einen Buchstaben A..Z, Eingabe erforderlich
- ? Platzhalter für einen Buchstaben A..Z, Eingabe optional
- a Platzhalter für einen Buchstaben oder eine Ziffer, Eingabe erforderlich
- A Platzhalter für einen Buchstaben oder eine Ziffer, Eingabe optional
- & Platzhalter für ein beliebiges Zeichen oder ein Leerzeichen, Eingabe erforderlich
- C Platzhalter für ein beliebiges Zeichen oder ein Leerzeichen, Eingabe optional



Gültigkeitsregeln

■ Vergleichsoperatoren:

Beispiele: > 100
< #12.01.93#
> "AN"
<> JA

=, <>, <, >, <=, >=
bei numerischen Werten
bei Datumswerten
bei Texten
bei Ja/Nein Werten

■ Zusammengesetzte Regeln: UND, ODER, NICHT, ZWISCHEN ... UND

Beispiele: >#1.1.1920# UND <= DATUM()
ZWISCHEN 100 UND 1000
NICHT ZWISCHEN 100 UND 1000

■ Vergleichsoperator WIE:

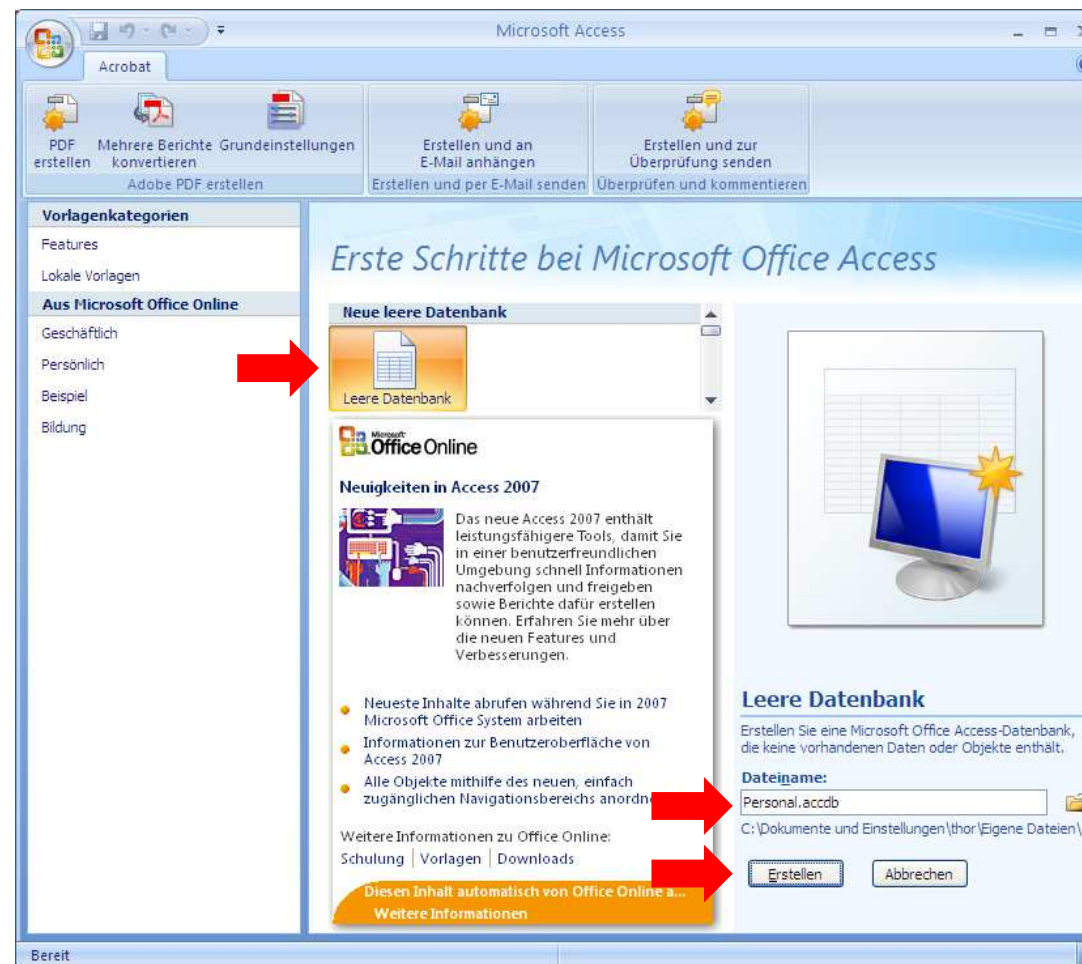
Beispiele: WIE "G*"
WIE "M??er"
WIE "#####"

Jokerzeichen * (bel. viele Zeichen)
Jokerzeichen ? (genau ein Zeichen)
Formatzeichen 0-9



Tabelle anlegen – 1

- MS Access 2010 starten
- Neue leere Datenbank anlegen
- Name der Datenbank eingeben – [Erstellen]



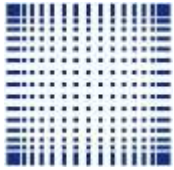


Tabelle anlegen – 2

- [Rechtsklick] auf Tabelle 1 – Entwurfsansicht
- Im „Speichern unter“ Dialog der Tabelle einen Namen geben
- Attribute und Domänen festlegen

| Feldname | Felddatentyp | Beschreibung |
|------------|--------------|----------------|
| PersonalNr | AutoWert | |
| Nachname | Text | |
| Vorname | Text | |
| Wohnort | Text | |
| PLZ | Zahl | |
| Strasse | Text | mit Hausnummer |
| Alter | Zahl | |

Feldeigenschaften

Allgemein Nachschlagen

| | |
|-----------------------|--------------|
| Feldgröße | Long Integer |
| Format | |
| Dezimalstellenanzeige | Automatisch |
| Eingabeformat | 00000 |
| Beschriftung | |
| Standardwert | |
| Gültigkeitsregel | |
| Gültigkeitsmeldung | |
| Eingabe erforderlich | Nein |
| Indiziert | Nein |
| Smarttags | |
| Textausrichtung | Standard |

Ein Feldname kann bis zu 64 Zeichen lang sein, einschließlich Leerzeichen. Drücken Sie F1, um Hilfe zu Feldnamen zu erhalten.



Tabelle anlegen – 3

- [X] – Änderungen an der Tabelle speichern – [Ja]
- [Doppel-Linksklick] auf dem Tabellennamen
- Daten eingeben – weiter: [TAB] – fertig: [X]

| Mitarbeiter | | | | | | | | × |
|-------------|-----------|---------|-----------|-------|------------------|-------|--|---|
| PersonalNr | Nachname | Vorname | Wohnort | PLZ | Strasse | Alter | | |
| 1 | Haas | Elsbeth | Offenbach | 63067 | Berliner Str. 22 | 42 | | |
| 2 | Richter | Hans | Offenbach | 63067 | Frankfurter Str | 32 | | |
| 3 | Friedrich | Irmgard | Offenbach | 63067 | Goethestr. 61 | 40 | | |
| 4 | Hartmann | Jochen | Frankfurt | 60528 | Berliner Str. 22 | 29 | | |
| 5 | Goldbach | Martin | Frankfurt | 60529 | Frankfurter Str | 35 | | |
| 6 | Naumann | Norbert | Frankfurt | 60529 | Goethestr. 61 | 38 | | |
| 7 | Klaus | Sven | Mannheim | | | | | |
| (Neu) | | | Mannheim | | | | | |

Datensatz: 7 von 7

Kein Filter

Sucher



Tabelle anlegen – 4

- Festlegung der Schlüssel:
In der Entwurfsansicht [Rechtsklick] in der Spalte vor Feldname
- [Rechtsklick] auf dem Tabellennamen ermöglicht jederzeit aber auch wieder das Öffnen der Tabelle in der Entwurfsansicht
- [Doppel-Linksklick] auf dem Tabellennamen entspricht [Rechtsklick] und dann der Auswahl Öffnen
- Sie können auch in der Dateneingabe-Maske neue Felder anlegen.
Jedoch weniger Möglichkeiten → Vermeiden!



Formular anlegen – 1

- Ribbon [Erstellen] – [Weitere Formulare] – [Formular-Assistent]
- Tabelle wählen – Alle Felder übernehmen [>>] – [Weiter]
- Layout auswählen – [Weiter]

Formular-Assistent

Welche Felder soll Ihr Formular enthalten?

Sie können aus mehr als einer Tabelle oder Abfrage auswählen.

Tabellen/Abfragen
Tabelle: Mitarbeiter

Verfügbare Felder:
PersonallNr
Nachname
Vorname
Wohnort
PLZ
Strasse
Alter

Ausgewählte Felder:

Abbrechen < Zurück Weiter > Fertig stellen

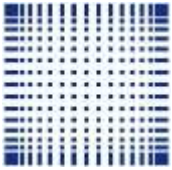


Formular-Assistent

Welches Layout soll Ihr Formular haben?

☒ Eingepflegt
☐ Tabellarisch
☐ Datenblatt
☐ In Blöcken

Abbrechen < Zurück Weiter > Fertig stellen



Formular anlegen – 2

- Format auswählen (reine Geschmackssache, Vorschau) – [Weiter]
- Namen für das Formular vergeben – [Fertig stellen]
- Danach öffnet sich das neue Formular

Formular-Assistent

Welches Format möchten Sie?

Text

Larissa
Lysithea
Metis
Modul
Nereus
Nordwind
Nyad
Okeanos
Papier
Phoebe
Rhea
Telesto
Windows Vista

Abbrechen < Zurück Weiter > Fertig stellen

Formular-Assistent

Welchen Titel soll Ihr Formular haben?

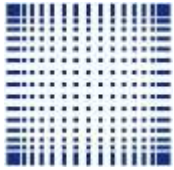
☒ Mitarbeiter

Dies sind alle Antworten, die der Assistent zur Erstellung Ihres Formulars benötigt.

Möchten Sie das Formular öffnen oder den Formularentwurf verändern?

☒ Das Formular öffnen.
☐ Den Formularentwurf verändern.

Abbrechen < Zurück Weiter > Fertig stellen



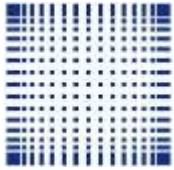
Formular anlegen – 3

- Ribbon [Erstellen] – [Formularentwurf] erlaubt eine sehr viel feinere Modellierung des Aussehens, Anordnung etc. des Formulars
- Dieser Layout Editor kann jederzeit auch nachträglich für bestehende Formulare aufgerufen werden [Rechtsklick]

The screenshot shows the Microsoft Access interface. On the left, the 'Alle Tabellen' (All Tables) pane lists 'Mitarbeiter' and 'Mitarbeiter : Tabelle'. A red arrow points to 'Mitarbeiter'. The main ribbon is 'Formularentwurf' (Form Design), with a red arrow pointing to the 'Formulare' group. The form 'Mitarbeiter' is displayed with the following fields:

| Mitarbeiter | |
|-------------|-------------------|
| PersonalNr | 1 |
| Nachname | Haas |
| Vorname | Elsbeth |
| Wohnort | Offenbach |
| PLZ | 63067 |
| Strasse | Berliner Str. 223 |
| Alter | 42 |

At the bottom, the status bar shows 'Datensatz: 1 von 7' and 'Kein Filter'.

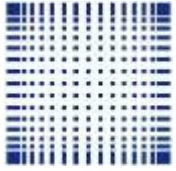


- Öffnen des Formulars für die Datenneueingabe, -modifikation durch [Doppel-Linksklick] auf dem Formularnamen
- Einen neuen, leeren Datensatz erzeugen Sie durch einen [Linksklick] auf das Sternsymbol im Datennavigator (Links unten in der Statusleiste im Formular)
- [Rechtsklick] auf dem Formularnamen öffnet ein Kontextmenü mit der Möglichkeit
 - ◆ das Formular normal zu öffnen
 - ◆ die angezeigten Felder nachträglich zu verändern (Layoutansicht) oder
 - ◆ das Formular um Schaltflächen ergänzen etc. (Entwurfsansicht)

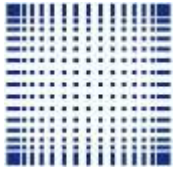


***Zu den Aufgabenzetteln der Übungsstunden
gibt es detaillierte Musterlösungen.
Durcharbeiten bringt aber mehr
Verständnis als nur herunterladen!***

***Reports, Nachschlageassistent und
einige weitere Dinge werden in den
Laborstunden vorgestellt!***



Welche Fragen gibt es?



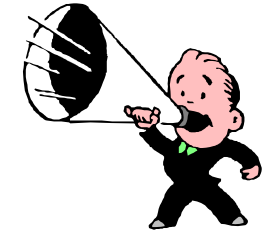
Zum Nachdenken:

***Wenn man auch in der Tabellen-Ansicht
Daten eingeben kann – wozu sind dann
die Formulare da?***

***Welche Funktion hat die Ribbon
Schaltfläche [Filtern]? – Ausprobieren!***



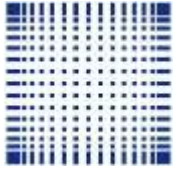
JETZT



Kapitel 3

Persistente Datenhaltung





Zum Nachdenken:

Wie speichern Sie Ihre Daten?

***Denken Sie dabei nicht nur an den
Computer, sondern auch an Medien wie
eine Musikkassette oder ein Videotape!***



HEUTE

- Speicherung auf magnetischen, optischen, solid-state Medien
- mit wahlfreiem Zugriff
- Speicherbereiche auf dem Medium werden ggf. nach Notwendigkeit verkettet
- Format stark abhängig vom Anwendungsprogramm



VERGANGENHEIT

- Lochstreifen, Magnetbänder, ...
- sequentieller Zugriff
- lineare Organisation des Speichermediums
- geringere Abhängigkeit vom Anwendungsprogramm
- aber erheblich höherer Anpassungsaufwand



- Lineare Datenorganisation wird auch heute noch verwendet
 - ◆ XML
 - ◆ Flache Dateien

```
File Edit Format Help
"01.10.03", "09:38:00", "Messpunkt 1", "23,15"
"01.10.03", "09:38:30", "Messpunkt 1", "23,28"
"01.10.03", "09:38:50", "Messpunkt 2", "18,78"
"01.10.03", "09:39:00", "Messpunkt 1", "23,32"
```



Charakteristik

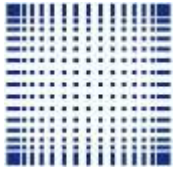
- jeder Benutzer arbeitet mit privaten Dateien (Files), teilweise unabhängig von anderen Benutzern
- Dateien werden nur von wenigen Programmen verwendet
- **physische Datenabhängigkeit:** Details der Datenformate, Sortierung, inhaltliche Beschreibung, etc. werden durch die Programme festgelegt



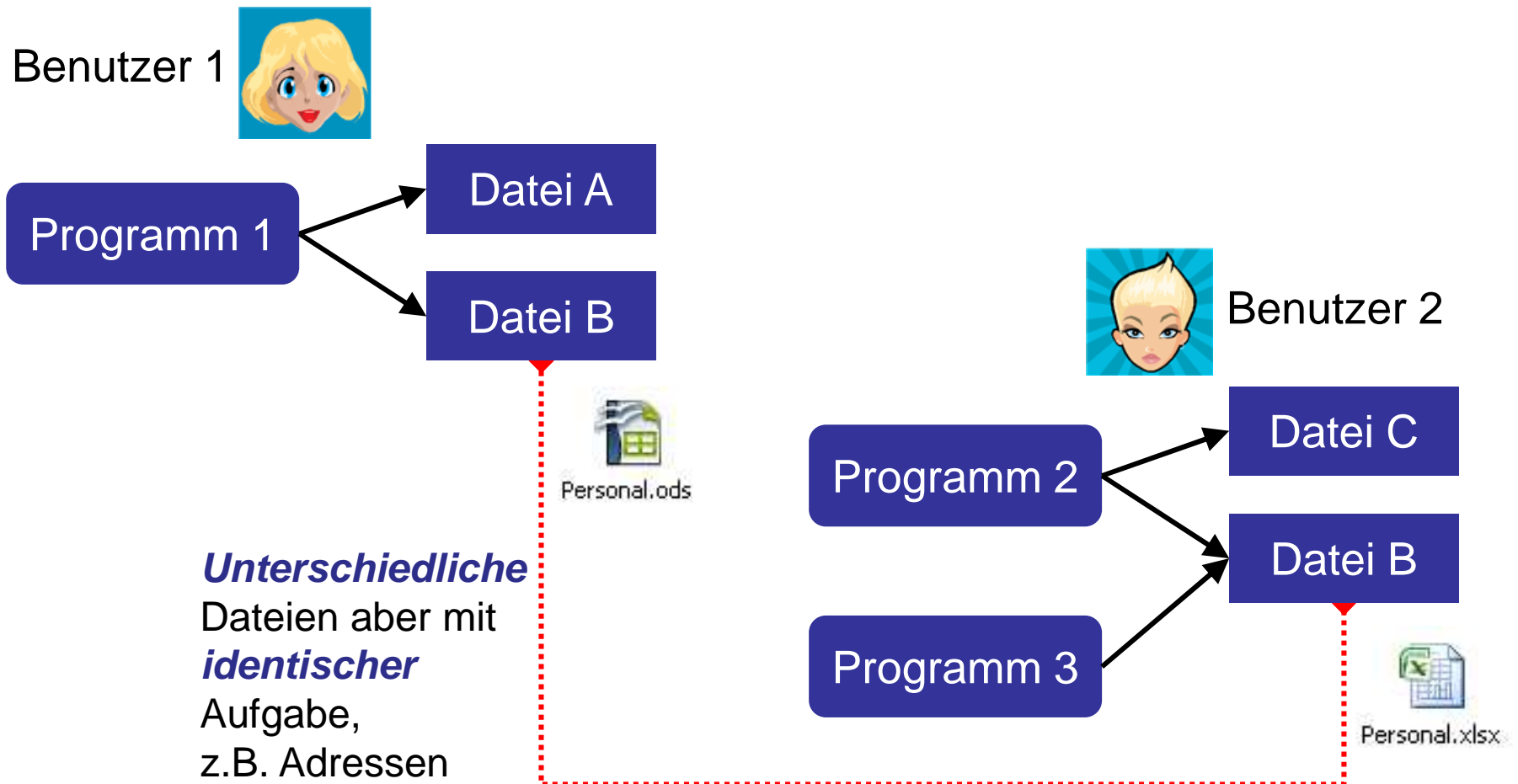
Einzeldaten ohne Integration – 2

Nachteile 👎

- Daten werden mehrfach gespeichert (Redundanz), dadurch besteht die Gefahr, dass die Daten inkonsistent sind
- physische Datenabhängigkeit
 - ◆ EDV-Kenntnisse der Dateibenutzer
 - ◆ Dateistruktur muss genau bekannt sein
 - ◆ geringfügige Änderung der Datenstruktur erfordert die Änderung der darauf zugreifenden Programme



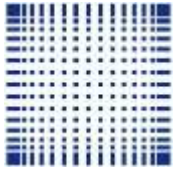
Einzeldaten ohne Integration – 3



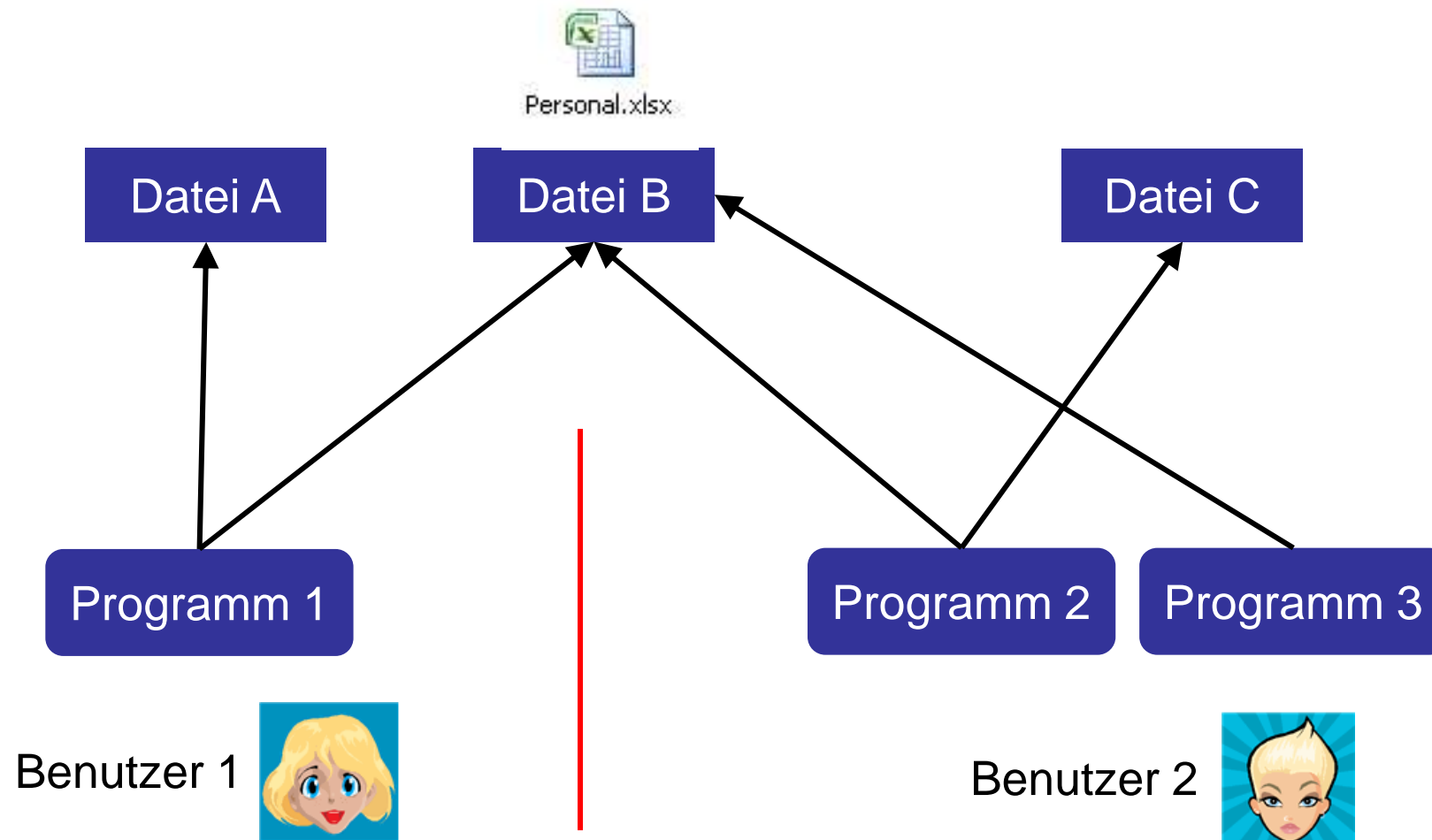


Charakteristik

- Datenerhebung, -kontrolle und -speicherung wird zentralisiert, diese Funktion wird durch den "Datenkoordinator" wahrgenommen
- Daten werden einheitlich in einem "data dictionary" beschrieben: Für jedes Feld eines jeden Datentyps einer jeden Datei erfolgt eine genaue Beschreibung des Inhaltes
- weiterhin physische Datenabhängigkeit



Physische Integration der Dateien – 2





Physische Integration der Dateien – 3

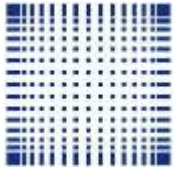
Gegenüber Einzeldateien ohne Integration

■ **Vorteil** 👍

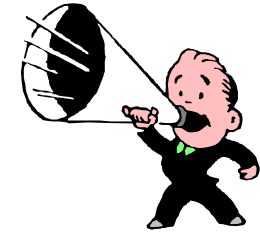
- ◆ Daten sind nicht mehr mehrfach gespeichert

■ **Nachteile** 👎

- ◆ Bei gravierenden Änderungen der Datenstruktur sind häufig sogar mehrere Programme von den Änderungen betroffen
- ◆ Zugriff auf vertrauliche Daten möglich



JETZT



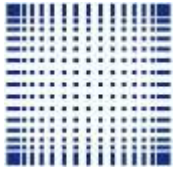
Kapitel 3.1

Zwei Schichten Modell

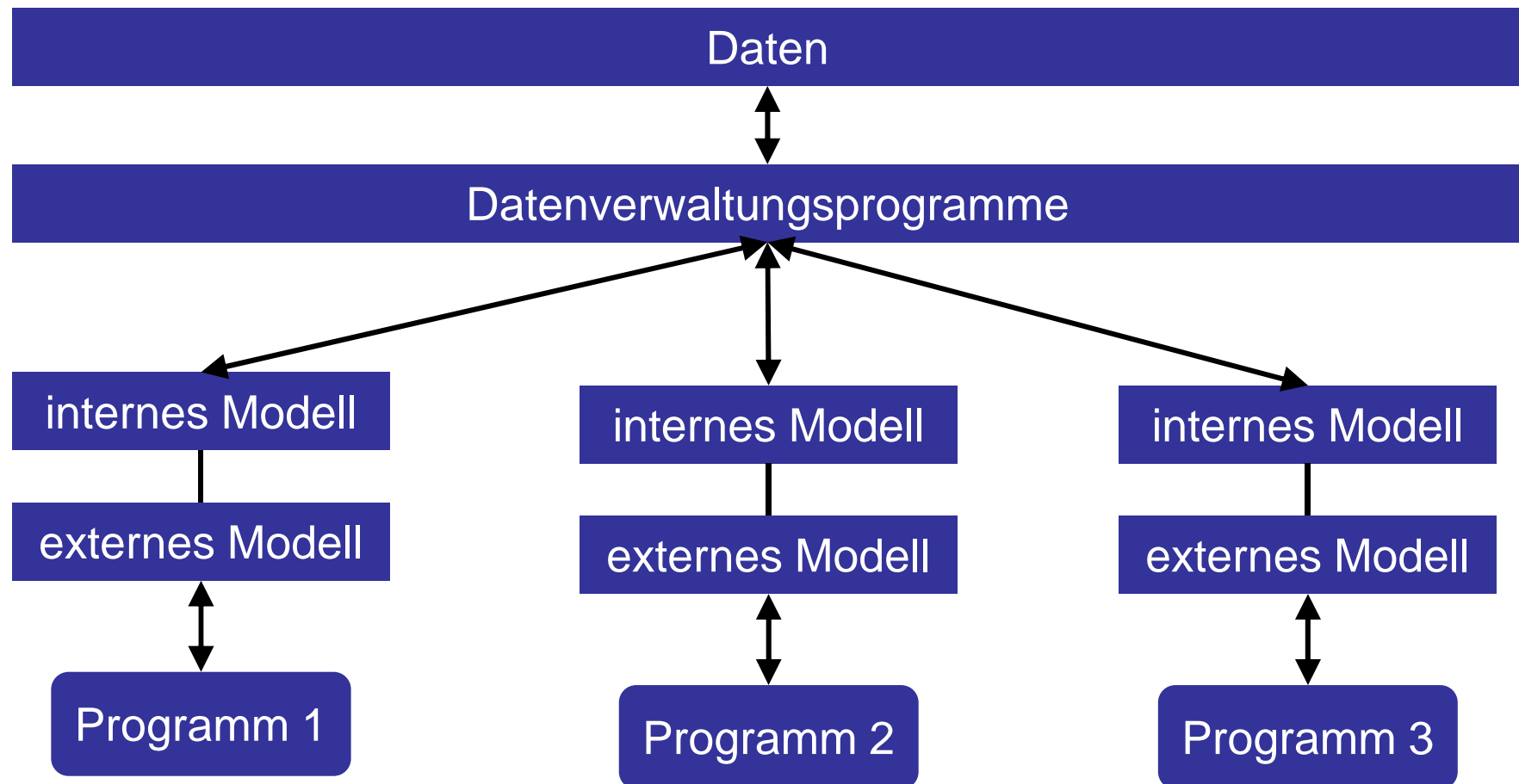


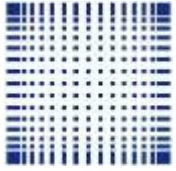
Charakteristik

- Einführung einer Trennungsebene zwischen Anwendungsaufgaben (Benutzersicht, externes Modell) und den gespeicherten Daten (interne Sicht)
- 👍 dadurch wird direkter Zugriff auf die Daten verhindert
- Daten werden von eigenen Programmen verwaltet, mit denen der Benutzer nichts mehr zu tun hat – er kommuniziert über standardisierte Schnittstelle
- 👍 Änderungen der physischen Struktur beeinflussen die Abbildungsprogramme, aber nicht das externe Modell



Zwei Schichten Modell – 2





Zum Nachdenken:

Welches der Probleme ist jetzt gelöst?

... und welches noch nicht?

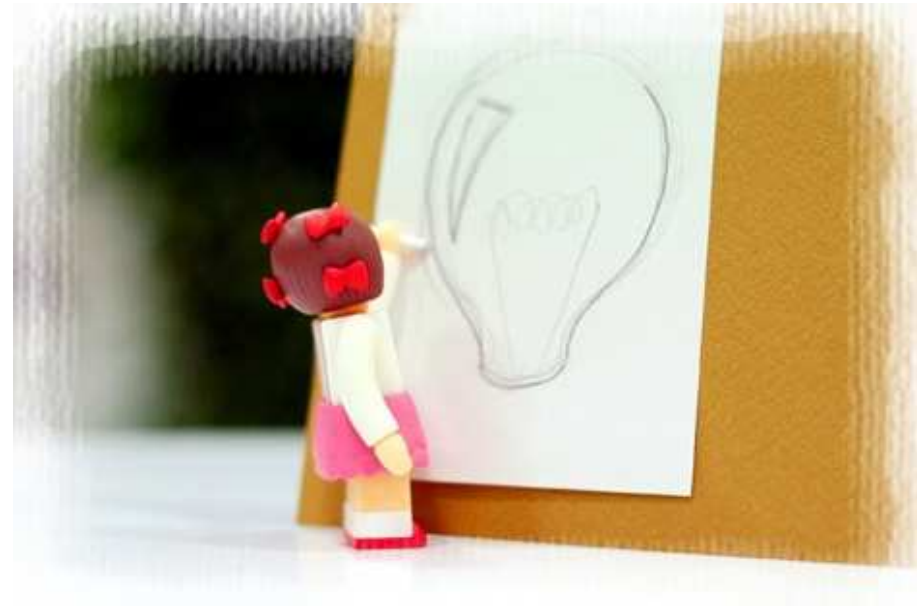
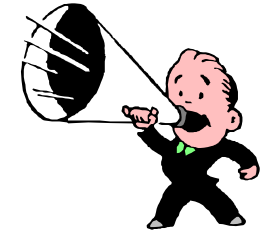
Was fehlt noch?



- Neben der physischen Datenunabhängigkeit fordert man zusätzlich noch eine:
- **Logische Datenunabhängigkeit**
 - ◆ Möglichkeit logische Änderungen an der Datenbasis durchzuführen
 - ◆ **ohne** darauf zugreifende Anwendungsprogramme signifikant zu beeinflussen
- Einführung zusätzlicher konzeptioneller Datensicht

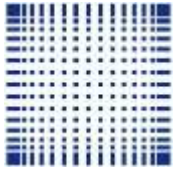


JETZT

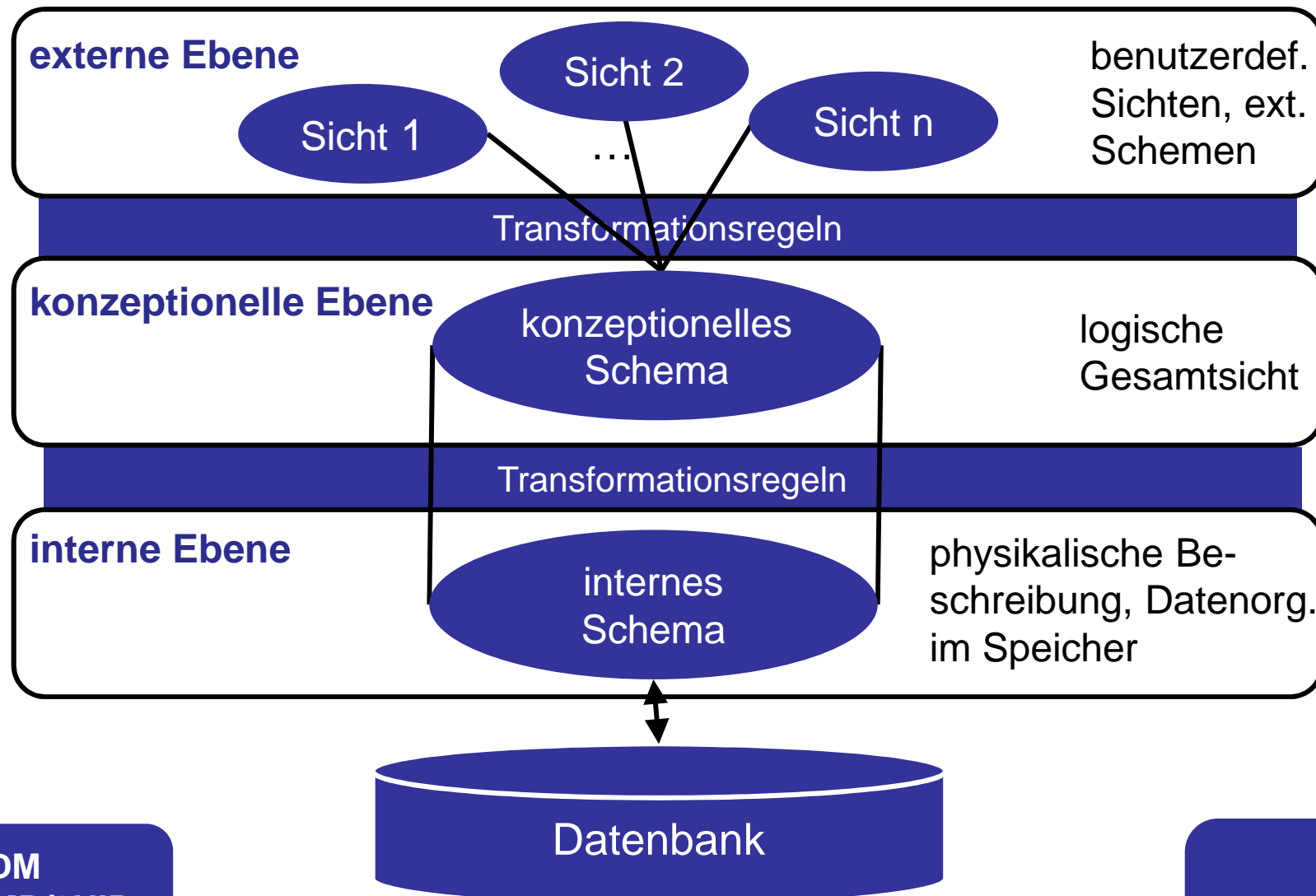


Kapitel 3.2

Drei Schichten Modell



Drei Schichten Modell





- Bereitstellung der Daten für die Anwendungen (die eigentliche Darstellung erfolgt bereits außerhalb der externen Ebene)
- Benutzersichten (Sichten) als Teile der logischen Gesamtsicht
- Benutzer hat so nur Zugriff auf die ihn betreffenden Daten
- Benutzer selektiert, liest Daten mit der Datenabfragesprache
- Benutzer fügt Daten hinzu, löscht, ändert mit der Datenmanipulationssprache



- Zusammenfassung aller Daten des Anwendungsbereiches, die in der DB gespeichert werden soll
- Beschreibung eines Ausschnitts aus der realen Welt – unabhängig von Vorgaben der Anwendungen = logische Gesamtsicht
- Das konzeptionelle Modell wird mit Hilfe der Datendefinitionssprache beschrieben
- Datenbankadministrator



- Organisation der Daten auf den Speichermedien
- Zugriffsmöglichkeiten auf die Daten
- Datenbankadministrator entwickelt
 - ◆ vom konzeptionellen Modell ausgehend
 - ◆ eine physische Datenorganisation,
 - ◆ die für alle Benutzer einen optimalen Datenzugriff ermöglicht
- Internes Modell wird direkt in die Datenbank übertragen



- Zwischen den drei Schichten erfolgt eine Transformation der Schemen ineinander
- Transformationsregeln im DBMS gespeichert
- ANSI-SPARC 1978
- Synonym: Drei Schichten Modell



Ablauf einer Datenbankabfrage – 1

- *Ein DBMS ermöglicht Daten zu speichern, wieder aufzufinden, um sie zu lesen oder zu ändern.*
- empfängt Anfragen, mit der Daten einer best. externen Sicht angefordert werden [$\rightarrow E$]
- liest die Definition der Sicht, prüft Syntax [E]
- prüft Zugriffsrechte des Benutzers auf diese Daten [E]
- Ermittlung benötigter Datenobjekte
Transformationsregeln: Externe Sicht \rightarrow konzeptionelles Schema [$E \rightarrow K$]



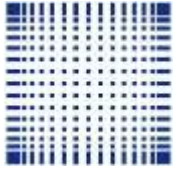
Ablauf einer Datenbankabfrage – 2

- Ermittlung phys. Datenobjekte, Zugriffspfade
Transformationsregeln: konzeptionelles Schema
→ internes Schema $[K \rightarrow I]$
- über Betriebssystem Daten lesen $[I \rightarrow]$
- Daten liegen im Systempuffer des DBMS $[\rightarrow I]$
- gewünschte Auswahl der Daten
zusammenstellen (durch entgegengesetzte
Anwendung der Transformationsregeln) $[I \rightarrow K \rightarrow E]$



Ablauf einer Datenbankabfrage – 3

- übergebene Daten für andere Benutzer sperren
 - ◆ bis zum Ende der Bearbeitung [K]
- Übergabe der Daten (im externen) Format an Anwendung bzw. Benutzer [E→]
- ... fertig?

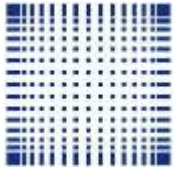


Welche Fragen gibt es?

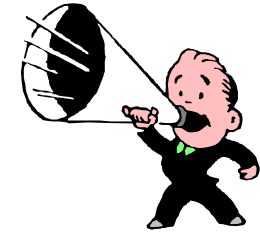


Zum Nachdenken:

*Was unterscheidet eine
Tabellenkalkulation (wie z.B. Excel) von
einem Datenbankmanagementsystem
(wie z.B. Access, Oracle, MySQL)?*

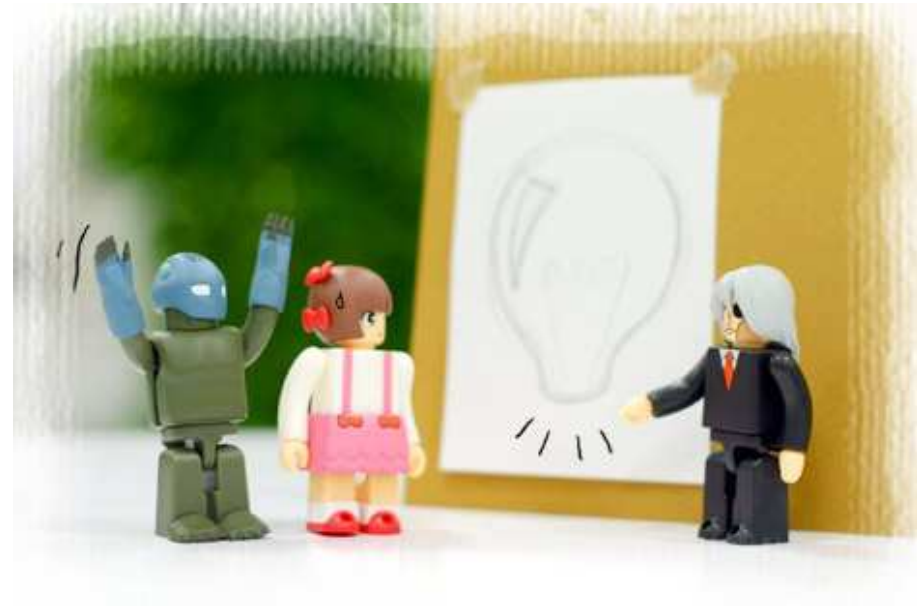


JETZT



Kapitel 4

Entity-Relationship-Modell





- **Dr. Peter Pin-Shan Chen, 1976**
- auch: Objekt-Beziehungs-Modell
- theoretisches Modell: ausschließlich für den konzeptionellen Entwurf eines Datenmodells
- seit 1976 mehrfach erweitert
- das heute am häufigsten verwendete Verfahren
- Ziel: implementierungsunabhängige Modellbildung
- Übertragung sehr leicht in das relationale Modell



- gedacht war es allerdings als rein theoretisches Modell zur Übertragung in *alle* praktischen Datenmodelle
- erst die Erweiterungen berücksichtigten die Besonderheiten von einigen Datenmodellen, i.w. die des relationalen Modells
- **Ziel: konzeptioneller Entwurf**, also ein Plan, den ein Mensch gut versteht, der inhaltliche Zusammenhänge verdeutlicht



■ Entity

- ◆ auch: Objekt, Entität, "Dinglichkeit"
- ◆ Gegenstände der realen Welt,
- ◆ die eindeutig voneinander unterscheidbar sind
- ◆ Beispiel: jedes Buch, jeder Benutzer, jede Prüfung
- ◆ Relational: oft eine Zeile einer Tabelle



■ Entity-Menge

- ◆ auch: Objektmenge, -sammlung
- ◆ Zusammenfassung ähnlicher Gegenstände der realen Welt
- ◆ Beispiel: alle Bücher, alle Benutzer, alle Prüfungen
- ◆ Relational: alle Zeilen einer Tabelle



■ Entity-Typ

- ◆ auch: Objekt-Typ, -Beschreibung
- ◆ Strukturbeschreibung (Eigenschaften) einer Entity-Menge
- ◆ Beispiel: alle Bücher haben die Attribute Nr, Titel, Autor, Verlag, ISBN, Standort
- ◆ Relational: Namen der Tabelle und ihrer Spalten, Wertebereiche der Spalten



Vorsicht *FALLE*



- Wir verwenden seit einigen Folien:
"Dinglichkeit" == Entity == Objekt
- **ABER:**
mit Objekt ist NICHT der Begriff **Objekt** aus der
objekt-orientierten Programmierung (z.B. mit
Visual Basic oder Java) gemeint!
(dort: Objekt = Eigenschaften + Methoden).
- Die Methoden werden beim (Datenbank-)Objekt
nicht mitberücksichtigt



■ Attribut

- ◆ Eigenschaften von Entity-Mengen, die allen Entities dieser Menge gemeinsam sind
- ◆ Beispiel: Note ist eine Eigenschaft aller Prüfungen
- ◆ Relational: Spaltenüberschriften einer Tabelle
- ◆ Wichtig: Die Attribute einer Entity müssen unterscheidbar sein (bedeutet: die Spaltennamen innerhalb einer Tabelle müssen sich unterscheiden)



■ Domäne

- ◆ Wertebereiche von Attributen
- ◆ Beispiel: Note hat die Domäne 1.0 .. 6.0
- ◆ Elementare Datentypen wie aus Pascal, C oder Java: Integer, Real, String
- ◆ Keine zusammengesetzten bzw. abstrakte Datentypen
- ◆ Dafür aber Unterbereichstypen



■ Schlüssel

- ◆ Menge von Attributen, die ein Entity eindeutig identifizieren
- ◆ Beispiel:
 - {Matrikelnr} ist Schlüssel für Studenten
 - {Matrikelnr} ist kein Schlüssel für Prüfungen
 - {Matrikelnr, Fach} ist ein Schlüssel für Prüfungen
- ◆ Minimalität der Menge wird nicht gefordert
- ◆ Es kann mehrere Schlüssel geben, es wird aber nur einer "ausgewählt" (bezeichnet)
- ◆ Definitionsgemäß: zumindest die Menge aller Attribute ist ein Schlüssel



■ Relationship-Typ

- ◆ Beziehung zwischen Entity-Mengen
- ◆ Beispiel: Ausleihe ist eine Beziehung zwischen Büchern und Benutzern mit einem zusätzlichen Attribut Ausleihdatum
- ◆ eine solche Beziehung: **Relationship** (analog zu Entity bzw. Tabellenzeile)
- ◆ Menge aller Beziehungen eines Relationship-Typs: **Relationship-Menge** (analog zu Entity-Menge bzw. alle Zeilen einer Tabelle)
- ◆ Relational: ebenfalls als Tabelle umgesetzt



- Relationship-Typen werden noch klassifiziert nach der Frage "Wieviele Entities einer Entity-Menge können mit wievielen Entities einer anderen Entity-Menge in Beziehung stehen?"
- **one-to-one (1:1)**
 - ◆ Für jedes Entity aus jeder Menge existiert höchstens ein zugeordnetes aus einer anderen Menge
 - ◆ Bijektion, eindeutig
 - ◆ Beispiel: Professoren, Lehrstühle



■ one-to-many (1:n)

- ◆ Ein Entity aus der Entity-Menge E_2 steht in Beziehung zu einer (evtl. leeren) Menge an Entities aus E_1 , jedes Entity in E_1 aber mit höchstens einem aus E_2
- ◆ Funktion, mehrdeutig
- ◆ Beispiel: Professoren, Vorlesungen



■ many-to-many (n:m)

- ◆ keine Einschränkung der Beziehung, beliebig viele Entities aus der Entity-Menge E_1 können mit beliebig vielen Entities aus der Menge E_2 in Beziehung stehen
- ◆ Relation, vieldeutig
- ◆ Beispiel: Studenten, Vorlesung

- Im relationalen Modell können many-to-many-Relationen **nicht** direkt übertragen werden!



■ Rekursive Beziehungen

- ◆ Möglicher, erlaubter Sonderfall
- ◆ entstehen, wenn ein Entity-Typ eine Assoziation auf sich selbst besitzt
- ◆ Beispiel: Entity-Typ *Person* und Relationship-Typ *verheiratet mit* (1:1) oder Entity-Typ *Bauteil* und Relationship-Typ *besteht aus* (1:n)
- ◆ 👎 führen häufig zu komplizierten Abfragen



■ Entity-Typen

- ◆ Entity-Name (Attribut₁, ... , Attribut_n)
 - ◆ Ein Schlüssel sollte in der Attributliste vorne stehen, der bzw. die Attributnamen werden unterstrichen
-
- BUCH (Nr, Titel, Autor, Verlag, Beschaffung, Standort)
 - BENUTZER (Nr, Name, Strasse, Wohnort, Gebdatum)



■ Relationship-Typen

- ◆ Relationship-Name (Entity-Name E_1 , Entity-Name E_2 , zusätzliches $\text{Attribut}_1, \dots$, zusätzliches Attribut_m)
- ◆ Relationship-Typen besitzen keinen Schlüssel – die eindeutige Identifikation der Relationships erfolgt durch die Schlüssel der beteiligten Entity-Typen

■ ausleihe (BUCH, BENUTZER, Leihdatum)



■ Konventionen

- ◆ ENTITY-TYPEN durchweg in GROSSBUCHSTABEN
- ◆ relationship-typen durchweg in kleinbuchstaben
- ◆ Bei Attributen den ersten Buchstabe GROSS danach klein weiter schreiben
- ◆ Schlüssel werden unterstrichen



Grafische Darstellung – 1

NAME

Entity-Typ

Name

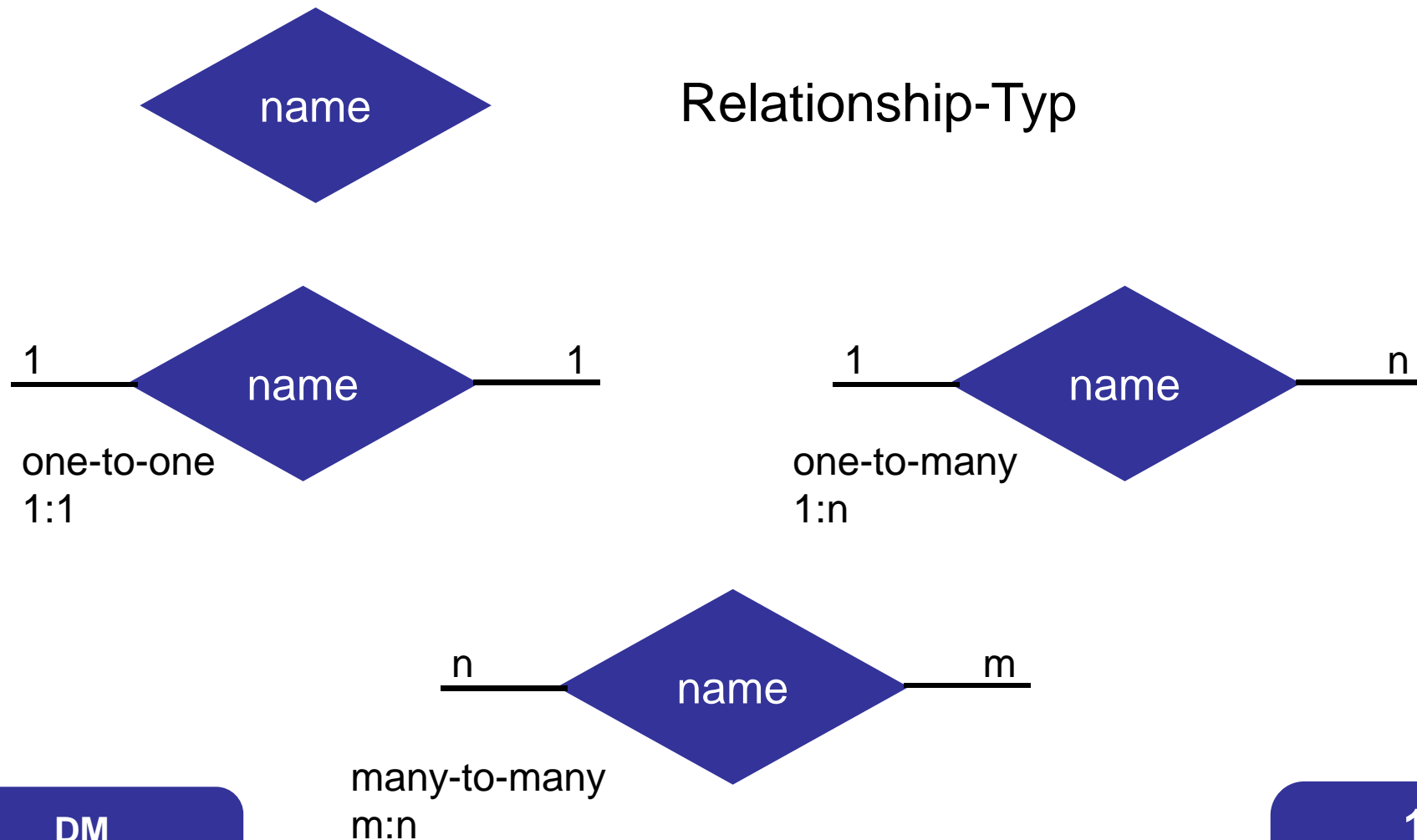
Attribut

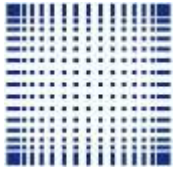
Name

... bzw. der Schlüssel

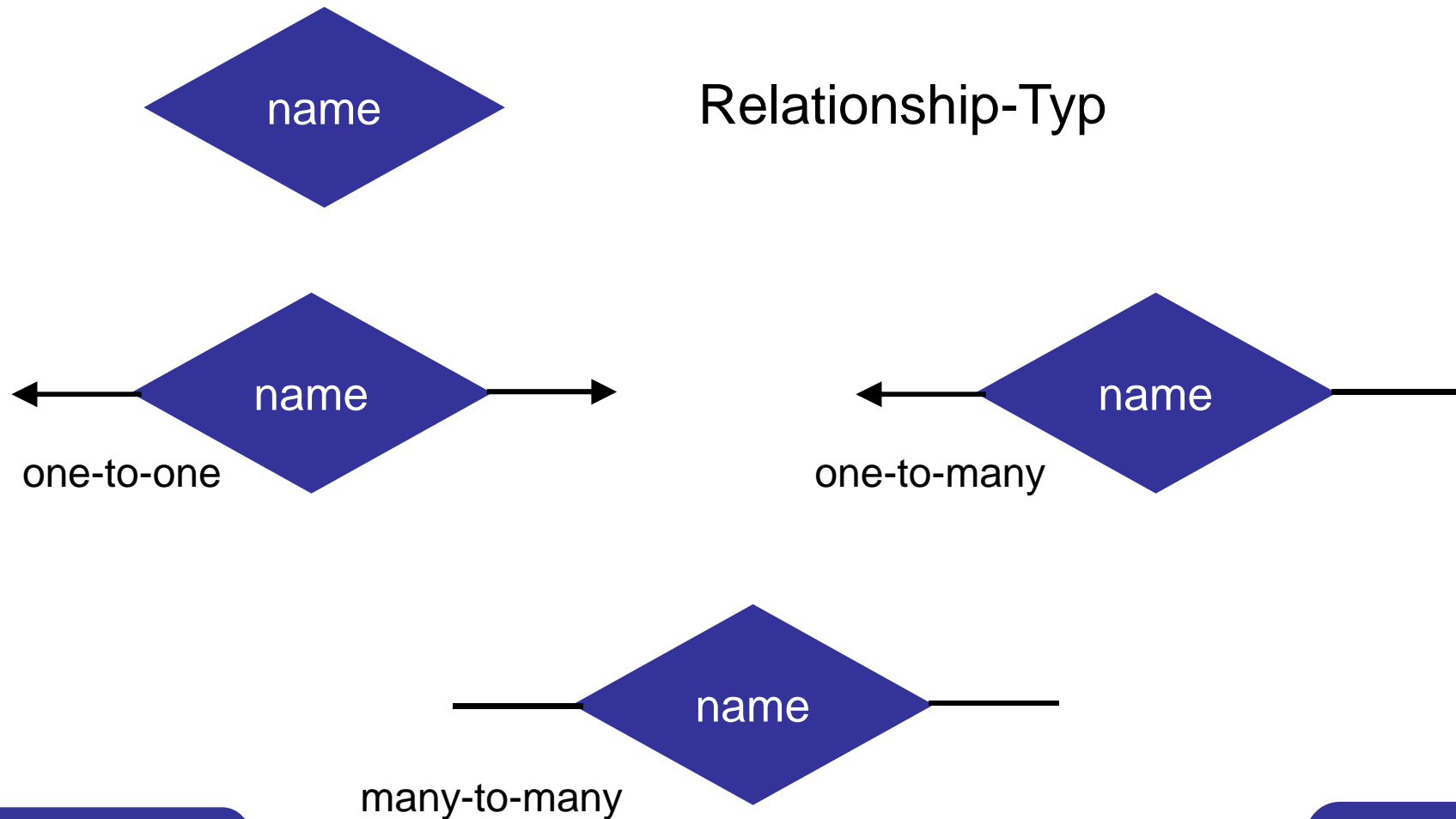


Grafische Darstellung – 2





Veraltete grafische Darstellung



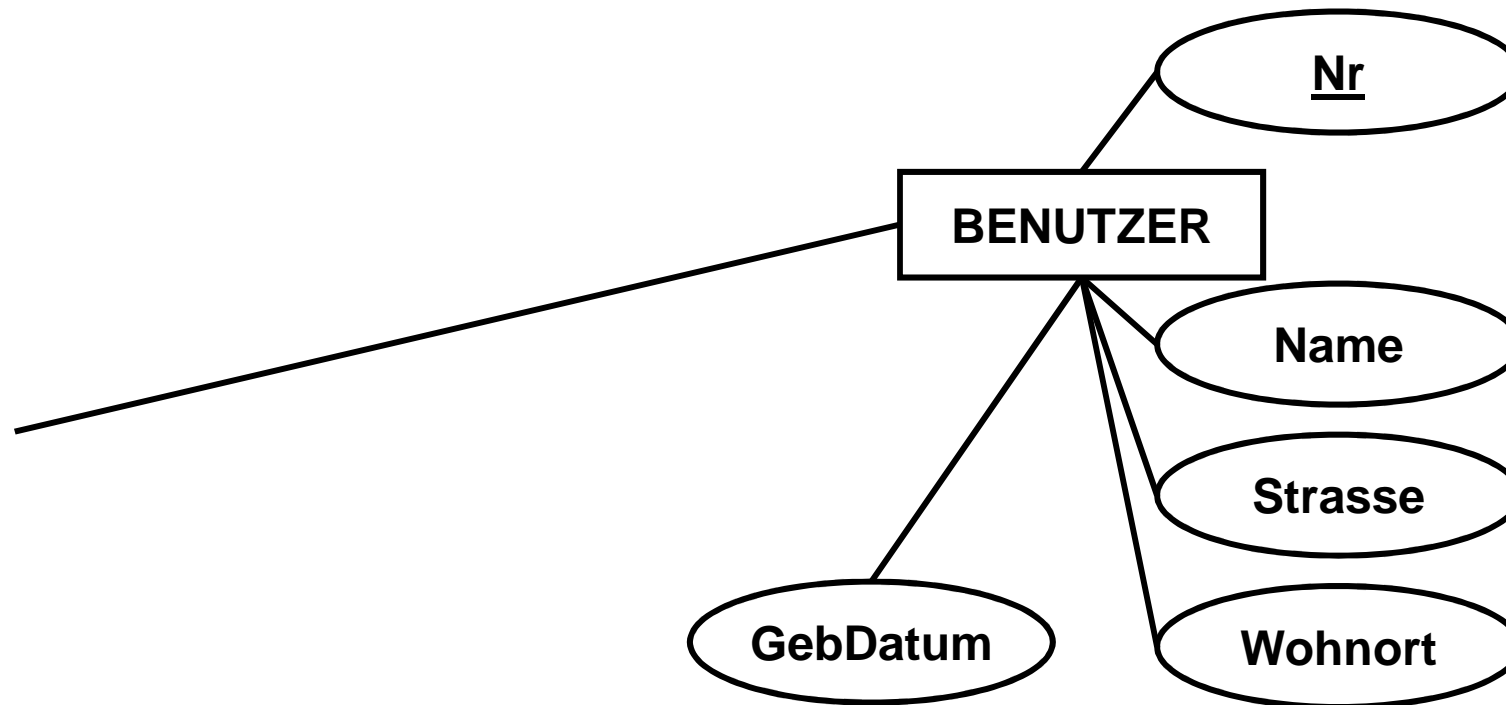


- Neben diese Diagrammen gibt es noch andere (grafische) Entwurfsmethoden
 - ◆ UML-Diagramme (zunehmende Bedeutung im oo-Umfeld; andere Vorlesungen)
 - ◆ SADT-Diagramme (werden kaum noch verwendet)

- Es gibt Software-Tools um derartige Diagramme grafisch am Bildschirm zu entwerfen
 - ◆ z.B. kommerzielle Tools wie MS Visio
 - ◆ preiswerte Freeware: Dia (Windows, Linux, Mac OS)



Beispiel – 1

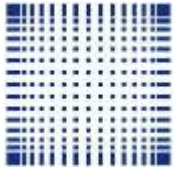


■ An der Tafel:

- ◆ BUCH (Nr, Titel, Autor, Verlag, Beschaffung, Standort)
- ◆ ausleihe (BUCH, BENUTZER, Leihdatum)



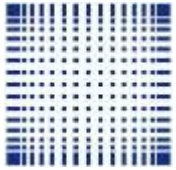
- Diese grafische Darstellung wird als **Entity-Relationship-Diagramm**, kurz **ER-Diagramm** oder **Objekt-Beziehungs-Diagramm** bezeichnet
- Die Übertragung von „natürlichsprachlichen“ Beschreibungen in ein ER-Diagramm mit entsprechender Textbeschreibung wird in der **Klausur** vorkommen



Studenten kennzeichnen sich durch ihre Matrikelnr., Name und Adresse. Bücher haben (zwingend) eine ISBN, einen Titel und einen Autor. Professoren werden gekennzeichnet durch Name, Geburtsdatum und Adresse. Lehrstühle besitzen eine Bezeichnung und einen Fachbereich. Vorlesungen haben einen Namen, Stundenumfang und Inhalt.

→ **Textdarstellung**

→ **Schlüssel festlegen**



Ausleihe: Studenten können beliebig viele Bücher ausleihen, wobei das Ausleihdatum zu erfassen ist.

Position: Ein Professor sitzt auf einem Lehrstuhl.

Veranstalter: Professoren halten Vorlesungen ab.

Teilnehmer: Studenten besuchen eine gewisse Anzahl an Vorlesungen.

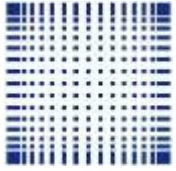
→ **Textdarstellung**

→ **Relationship-Typ Klassifikation (Kardinalität)**

→ **ER-Diagramm**



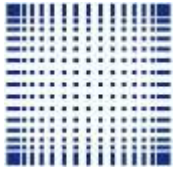
Welche Fragen gibt es?



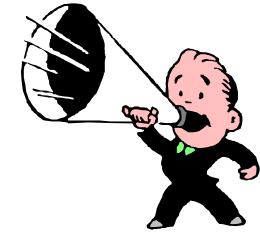
Zum Nachdenken:

*Wozu braucht man einen
theoretischen Datenbankentwurf?*

*Ist es sinnvoll, ein ER-Diagramm
zu erstellen, wenn es schon eine
Datenbank gibt?*

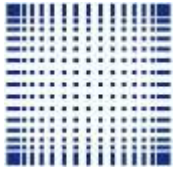


JETZT



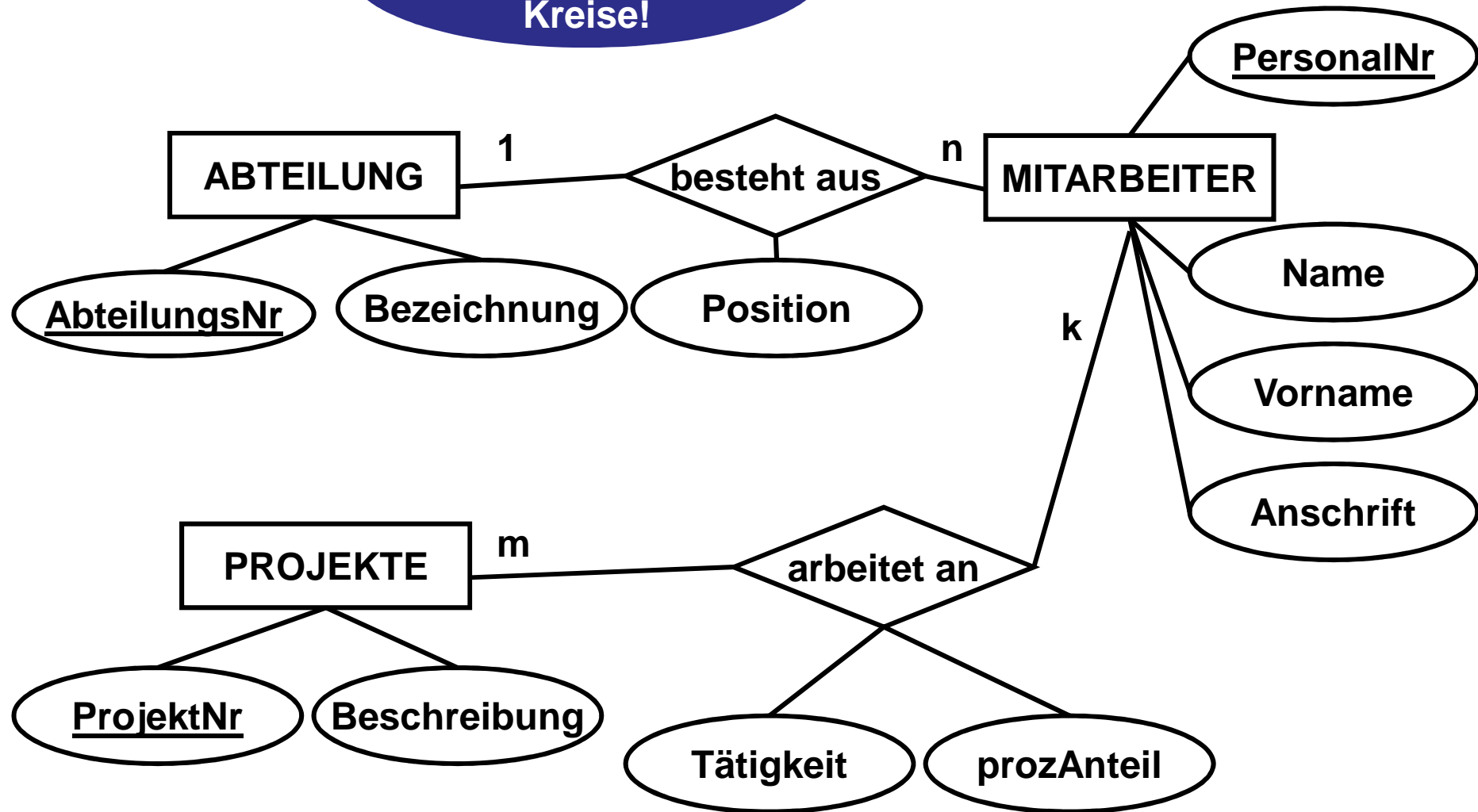
Kapitel 5

Übertragung des ER-Modells in das relationale Modell



Beispiel – 1

Keine
semantischen
Kreise!





- Dieses ER-Modell soll nun schrittweise in ein relationales Modell überführt werden
- Die Anzahl der Tabellen, die aus dem ER-Modell resultieren ist abhängig von den
 - ◆ definierten Entity-Typen und den
 - ◆ Relationship-Typen zwischen den Entitätsmengen
- Bei der Transformation geht man schrittweise vor:
 - ◆ Entity-Typen, **dann**
 - ◆ Relationship-Typen



- Für jeden Entity-Typ wird eine Tabelle erstellt, welche für jedes Attribut eine Spalte besitzt
- Primärschlüssel werden unverändert übernommen
- **Ziel:** Überführung des konzeptionellen Entwurfs für den Menschen in eine **Darstellung im relationalen Modell**, also eine technische Umsetzung, einem Schaltplan vergleichbar, der direkt in MS Access umgesetzt werden kann
- Konkret zu betrachten die Entitätstypen:
 - ◆ **ABTEILUNG**
 - ◆ **MITARBEITER**
 - ◆ **PROJEKTE**

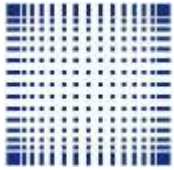



Tabelle ABTEILUNG

| | Attribut | Datentyp |
|-----------------------------------------------------------------------------------|--------------|-------------|
|  | AbteilungsNr | AutoWert |
| | Bezeichnung | Kurzer Text |

| AbteilungsNr | Bezeichnung |
|--------------|-------------|
| 1 | Personal |
| 2 | Einkauf |
| 3 | Verkauf |

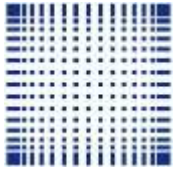



Tabelle MITARBEITER

| | Attribut | Datentyp |
|-----------------------------------------------------------------------------------|------------|-------------|
|  | PersonalNr | AutoWert |
| | Name | Kurzer Text |
| | Vorname | Kurzer Text |
| | Anschrift | Kurzer Text |

| PersonalNr | Name | Vorname | Anschrift |
|------------|------------|-----------|-------------------------------|
| 1 | Lorenz | Sophia | 03725 Mausbach Mühlweg 4 |
| 2 | Hohl | Tatjana | 49262 Mausloch Käsestr. 5 |
| 3 | Willschrei | Theodor | 03453 Katzbergen Ahornweg 4 |
| 4 | Richter | Hans | 78943 Katzenhausen Buchweg 28 |
| 5 | Wiesenland | Brunhilde | 02518 Hundsberg Mopsstr. 45 |

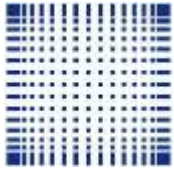

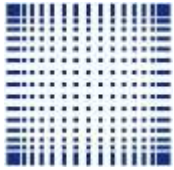


Tabelle PROJEKTE

| | Attribut | Datentyp |
|-----------------------------------------------------------------------------------|--------------|-------------|
|  | ProjektNr | AutoWert |
| | Beschreibung | Kurzer Text |

| ProjektNr | Beschreibung |
|-----------|-------------------|
| 1 | Kundenumfrage |
| 2 | Verkaufsmesse |
| 3 | Konkurrenzanalyse |



1:1, 1:n-Beziehungen – 1


- Für die Umsetzung gibt es zwei Möglichkeiten:
 1. Neue Tabelle, Attribute werden die Primärschlüssel der beiden beteiligten Tabellen.
Weitere beschreibende Attribute werden zusätzliche Spalten
Bei 1:1 kann der Primärschlüssel beliebig aus den Schlüsseln der Tabellen gewählt werden
Bei 1:n wird der Schlüssel der ":n" Tabelle zum Primärschlüssel der neuen Tabelle

👉 es wird eine weitere Tabelle benötigt

👉 Es treten keine Nullwerte auf



Tabelle besteht aus

| | Attribut | Datentyp |
|-----------------------------------------------------------------------------------|--------------|--------------|
|  | PersonalNr | Long Integer |
| | AbteilungsNr | Long Integer |
| | Position | Kurzer Text |

| PersonalNr | AbteilungsNr | Position |
|------------|--------------|---------------|
| 1 | 1 | Leiterin |
| 2 | 2 | Leiterin |
| 3 | 2 | Mitarbeiter |
| 4 | 3 | Leiter |
| 5 | 3 | Mitarbeiterin |



1:1, 1:n-Beziehungen – 2

■ Alternativ:

2. Eine der beiden Tabellen wird um eine Spalte erweitert. Bei 1:1 ist die Tabelle egal, bei 1:n wird die ":n" Tabelle erweitert. Besitzt die Beziehung beschreibende Attribute, so werden diese ebenfalls zu der zu erweiternden Tabelle hinzugefügt. Der Primärschlüssel bleibt unverändert

👎 Es können Nullwerte auftreten

👍 Es entsteht keine weitere Tabelle



Erweiterte Tabelle MITARBEITER – 1

| | Attribut | Datentyp |
|---|------------|--------------|
| ⌘ | PersonalNr | AutoWert |
| | Name | Kurzer Text |
| | Vorname | Kurzer Text |
| | Anschrift | Kurzer Text |
| | AbtNr | Long Integer |
| | Position | Kurzer Text |

- Ein Nullwert kann in diesem Beispiel dann auftreten, wenn ein Mitarbeiter (noch) keiner Abteilung zugeordnet ist

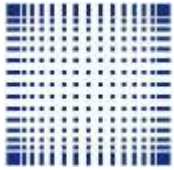


Erweiterte Tabelle MITARBEITER – 2

| PersonalNr | Name | Vorname | Anschrift | AbtNr | Position |
|------------|------------|-----------|--------------|-------|-------------|
| 1 | Lorenz | Sophia | 03725 Mau... | 1 | Leiter |
| 2 | Hohl | Tatjana | 49262 Mau... | 2 | Leiter |
| 3 | Willschrei | Theodor | 03453 Kat... | 2 | Mitarbeiter |
| 4 | Richter | Hans | 78943 Kat... | 3 | Leiter |
| 5 | Wiesenland | Brunhilde | 02518 Hun... | 3 | Mitarbeiter |

VORSICHT:

- Erweitern Sie bei einer 1:n Beziehung die "1:" Tabelle,
 - ◆ so entstehen Redundanzen und
 - ◆ das Schlüsselkriterium wird verletzt
- Betrachten wir dies einmal auf der nächsten Folie:



(falsch) erweiterte Tabelle ABTEILUNG

| | Attribut | Datentyp |
|---|--------------|--------------|
| ⌨ | AbteilungsNr | AutoWert |
| | Bezeichnung | Kurzer Text |
| | PersonalNr | Long Integer |
| | Position | Kurzer Text |

| AbteilungsNr | Bezeichnung | PersonalNr | Position |
|--------------|-------------|------------|-------------|
| 1 | Personal | 1 | Leiter |
| 2 | Einkauf | 2 | Leiter |
| 2 | Einkauf | 3 | Mitarbeiter |
| 3 | Verkauf | 4 | Leiter |
| 3 | Verkauf | 5 | Mitarbeiter |

!



- Bei m:n-Beziehungen ist immer eine zusätzliche Tabelle erforderlich
- Die Attribute dieser neuen Tabelle setzen sich aus den beiden Schlüsseln der beiden beteiligten Tabellen zusammen
- Die Vereinigung der Schlüssel der beiden beteiligten Tabellen wird Primärschlüssel der neuen Tabelle
- Die neue Tabelle kann ggf. weitere beschreibende Attribute erhalten



- Für unser Beispiel wird eine Tabelle arbeitet_an benötigt
- Das Datenbankschema wird aus
 - ◆ den beiden Primärschlüsseln von MITARBEITER und PROJEKTE und
 - ◆ den Attributen Tätigkeit und prozAnteil (prozentualer Anteil der Arbeitszeit) aufgebaut

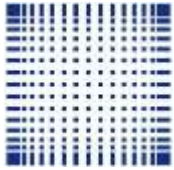


Tabelle arbeitet_an – 1

| | Attribut | Datentyp |
|---|------------|--------------|
| ⌘ | PersonalNr | Long Integer |
| ⌘ | ProjektNr | Long Integer |
| | Tätigkeit | Kurzer Text |
| | prozAnteil | Long Integer |

gefüllt mit den folgenden Daten: *(siehe nächste Folie)*

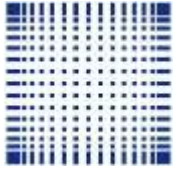


Tabelle arbeitet_an – 2

| PersonalNr | ProjektNr | Tätigkeit | prozAnteil |
|------------|-----------|---------------------------|------------|
| 2 | 1 | Leiter | 25 |
| 3 | 1 | Sachbearbeiter | 50 |
| 4 | 1 | Sachbearbeiter | 50 |
| 4 | 2 | Leiter | 25 |
| 5 | 2 | Präsentationsvorbereitung | 100 |
| 4 | 3 | Leiter | 25 |
| 2 | 3 | Sachbearbeiter | 50 |
| 3 | 3 | Sachbearbeiter | 50 |



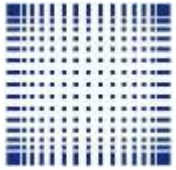
Welche Fragen gibt es?



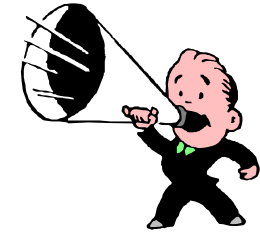
Zum Nachdenken:

*Lassen sich die Tabellen einer Datenbank
zurückführen in ein ER-Modell?*

*Woran erkennt man eine many-to-many
Beziehung im relationalen Modell?*



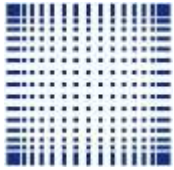
JETZT



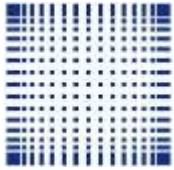
Kapitel 6

Microsoft Access

2. Beziehungen, Abfragen



- **KUNDE**(KundeNr, Nachname, Vorname, Straße, PLZ, Ort)
AUFTRAG(AuftragNr, KundeNr, ArtikelNr, Bezeichnung, Stück)
- Zum Nachdenken: *Wie sieht das ER-Diagramm aus?*
- Offensichtlich liegt hier eine 1:n – Beziehung vor – Welche?
von KundeNr (in **KUNDE**) zu KundeNr (in **AUFTRAG**)
- Offensichtlich hat unsere Modellierung eine wesentliche Einschränkung gegenüber der realen Welt:
Bei einem Auftrag kann nur ein Artikel bestellt werden.
- Jetzt brauchen wir nur noch so richtig sinnvolle Beispieldaten ☺



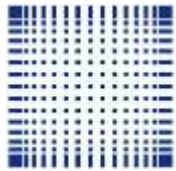
Vorsicht *FALLE*



- Es gibt im Entity-Relationship-Modell eine textuelle und eine grafische Darstellung (das ER-Diagramm) – vgl. Kapitel 4
- Da man nicht Tabellen „malen“ möchte, gibt es im relationalen Datenmodell also ebenfalls eine textuelle Darstellung – siehe das Beispiel oben und vgl. Kapitel 5
- Die beiden textuellen Darstellungen sehen sich leider sehr ähnlich – Aufpassen! Die Unterschiede werden z.B. bei den Beziehungen deutlich.



- Wir hatten eben also die textuelle Darstellung des relationalen Datenmodells vor uns!
- In **beiden** textuellen Darstellungen ist es erlaubt, den Datentyp für jedes Attribut anzugeben:
AUFTRAG(AuftragNr : AutoWert, KundeNr : LongInteger, ArtikelNr : LongInteger, Bezeichnung : Text(50), Stück : Integer)
- Der Wert in Klammern hinter dem Datentyp gibt die Feldgröße an: bei Text also die maximale Anzahl Zeichen – die Angabe darf entfallen



Beziehungen – 2

| Kunde | | | | | | | X |
|--------------------|----------|----------|---------|---------------|-------|--------------|---|
| | KundenNr | Nachname | Vorname | Straße | PLZ | Ort | |
| | 1 | Maier | Werner | Maierweg 5 | 30000 | Maiershausen | |
| | 2 | Müller | Walter | Müllerstr. 2 | 10000 | Müllersdorf | |
| | 3 | Bauer | Gerd | Bauerallee 10 | 50000 | Bauershausen | |
| * | (Neu) | | | | | | |
| Datensatz: 3 von 3 | | | | | | | |
| Kein Filter | | | | Suchen | | | |

| Auftrag | | | | | | X |
|--------------------|-----------|----------|-----------|-------------|-------|--------|
| | AuftragNr | KundenNr | ArtikelNr | Bezeichnung | Stück | |
| | 1 | 3 | 3837 | Disketten | 10 | |
| | 2 | 1 | 389 | Monitor | 2 | |
| | 3 | 3 | 9372 | Drucker | 5 | |
| | 4 | 2 | 389 | Monitor | 1 | |
| | 5 | 2 | 9272 | Tastatur | 2 | |
| | 6 | 3 | 8263 | Maus | 1 | |
| * | (Neu) | | | | | |
| Datensatz: 6 von 6 | | | | | | |
| Kein Filter | | | | | | Suchen |



Beziehungen – 3

- Noch haben wir zwei "nebeneinander her" existierende Tabellen. Bei Aufträgen könnten nicht-existente Kundennummern vergeben oder eine Kundennummer gelöscht werden, obwohl noch Aufträge für diese Kundennummer gespeichert sind → Inkonsistente Daten
- Damit das nicht passiert, müssen wir Access diese Beziehungen "bekannt" geben
- Dieses Vorgehen ist ebenfalls notwendig um Abfragen nach Daten über mehrere Tabellen realisieren zu können
- Die KundenNr in der Tabelle Auftrag ist kein Schlüssel, dient aber zum Auffinden des richtigen Kunden (über den Schlüssel von Kunde). Daher nennt man KundenNr in der Tabelle Auftrag **Fremdschlüssel**
- Ribbon [Datenbanktools] – [Beziehungen] ...
- In der Dialogmaske *Tabelle anzeigen* die Tabellen **Kunde** und **Auftrag** hinzufügen



Beziehungen – 4

- In der Tabelle *Kunde* mit der Maus das Feld *KundenNr* anklicken und bei gedrückt gehaltener Maustaste auf Attribut *KundenNr* in der Tabelle *Auftrag* ziehen und dort loslassen

Beziehungen bearbeiten

Tabelle/Abfrage: Kunde Verwandte Tabelle/Abfrage: Auftrag

KundenNr KundenNr

☒ Mit referentieller Integrität
☒ Aktualisierungsweitergabe an verwandte Felder
☒ Löschweitergabe an verwandte Datensätze

Beziehungstyp: 1:n

Erstellen
Abbrechen
Verknüpfungstyp...
Neue erstellen...

- Jetzt können Abfragen über mehrere Tabellen ausgeführt werden
- Die Checkhaken bei referentieller Integrität müssen zusätzlich gesetzt werden, um die Inkonsistenzen zu vermeiden



Dabei bedeutet ...

- **... referentielle Integrität:**

Es kann kein Auftrag angelegt werden, wenn es nicht einen zugehörigen, eingetragenen Kunden in der Tabelle Kunde gibt

- **... Aktualisierungsweitergabe:**

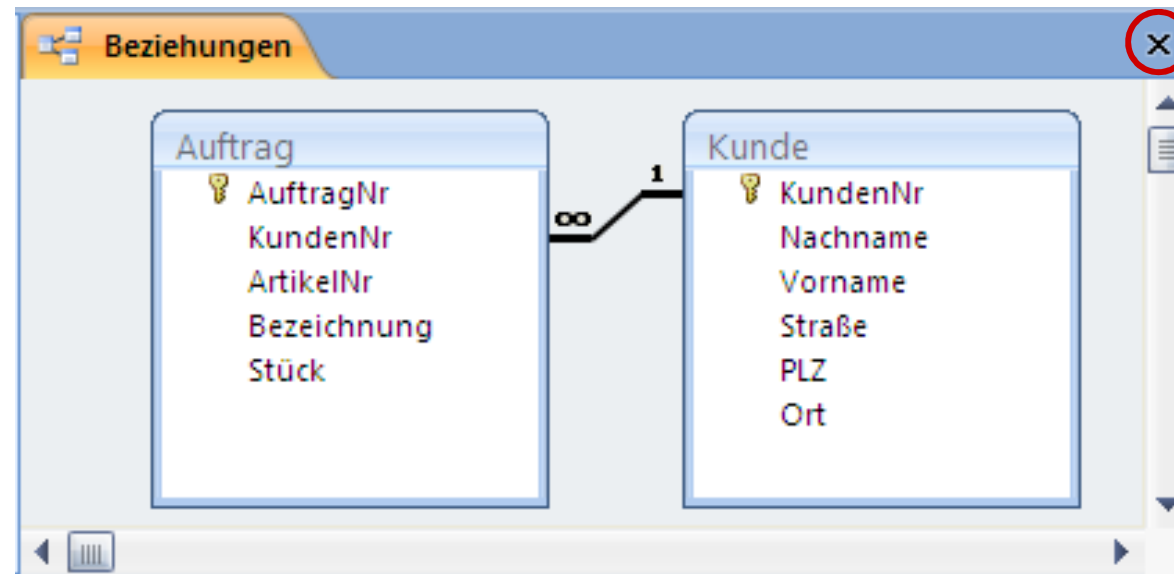
Wenn in der Tabelle Kunde die KundenNr geändert wird (Tippfehler?), so bekommen automatisch auch alle Aufträge dieses Kunden die neue KundenNr

- **... Löschweitergabe:**

Wenn ein Kunde in der Tabelle Kunde komplett gelöscht wird (Datenbereinigung?), so werden auch alle Aufträge dieses Kunden gelöscht



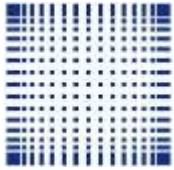
Beziehungen – 6



- Bei Verwendung der referentiellen Integrität wird der Beziehungstyp (hier 1:n) in der Grafik angezeigt und von Access jetzt bei der Eingabe von Daten geprüft
- Gleich können wir Anfragen an unseren Datenbestand stellen



- Scheinbar kennt MS Access nur 1:n – **Warum?**
- **many-to-many**
Muss über eine Zwischentabelle übersetzt werden (vergleiche hierzu Kapitel 5). Die Zwischentabelle wird über zwei 1:n Beziehungen eingebunden.
- **one-to-one**
Hier muss zusätzlich in der Tabellendefinition (vergleiche hierzu Kapitel 2) nachgeholfen werden: Der Fremdschlüssel muss bei seinen Eigenschaften **Indiziert** auf **Ja (Ohne Duplikate)** gesetzt werden.



Abfragen auf einer Tabelle – 1

- **Frage:** In welchen Aufträgen (*AuftragNr*) wird ein Monitor (*ArtikelNr* = 389) bestellt
- Ribbon [Erstellen] – [Abfrageentwurf]
- Dialog [Tabelle anzeigen] – Tabelle **Auftrag** hinzufügen
- Zuerst die anzuzeigenden Felder (*AuftragNr*) hinzufügen. Anzeigen muss einen Checkhaken haben
- Dann die Abfragebedingung festlegen:

| Feld: | AuftragNr | ArtikelNr | | |
|-------------|-------------------------------------|--------------------------|--------------------------|--------------------------|
| Tabelle: | Auftrag | Auftrag | | |
| Sortierung: | | | | |
| Anzeigen: | <input checked="" type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| Kriterien: | | =389 | | |
| oder: | | | | |

vgl.
Gültigkeits-
regeln



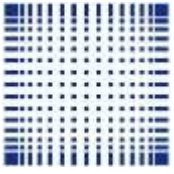
Abfragen auf einer Tabelle – 2

- Feld ArtikelNr = 389. Anzeigen wird in der Regel nicht angekreuzt
- Das * steht für alle Attribute (einer Tabelle)
- Abschließend bekommt die Abfrage einen Namen (AuftragMonitore)
- [Doppel-Linksklick] auf dem Abfrageobjekt oder [Rechtsklick] / Öffnen führt die Abfrage auf dem aktuellen Datenbestand aus

| Alle Tabellen | |
|-------------------|--|
| Kunde | |
| Kunde : Tabelle | |
| Auftrag | |
| Auftrag : Tabelle | |
| AuftragMonitore | |

| AuftragMonitore | |
|-----------------|-------|
| AuftragNr | |
| | 2 |
| | 4 |
| * | (Neu) |

Datensatz: 1 von 2



Abfragen über mehrere Tabellen – 1

- Frage: Welche Kunden (*KundeNr*, *Vorname*, *Nachname*) haben Monitore (*ArtikelNr* = 389) bestellt?
- Ribbon [Erstellen] – [Abfrageentwurf]
- Dialog [Tabelle anzeigen] – Tabelle **Auftrag** und **Kunde** hinzufügen

KundenMonitore

Auftrag

- * AuftragNr
- KundenNr
- ArtikelNr
- Bezeichnung
- Stück

Kunde

- * KundenNr
- Nachname
- Vorname
- Straße
- PLZ
- Ort

| Feld: | KundenNr | Vorname | Nachname | ArtikelNr |
|-------------|-------------------------------------|-------------------------------------|-------------------------------------|--------------------------|
| Tabelle: | Kunde | Kunde | Kunde | Auftrag |
| Sortierung: | | | | |
| Anzeigen: | <input checked="" type="checkbox"/> | <input checked="" type="checkbox"/> | <input checked="" type="checkbox"/> | <input type="checkbox"/> |
| Kriterien: | | | | =389 |
| oder: | | | | |

Bedingung

anzuzeigende Felder



Abfragen über mehrere Tabellen – 2

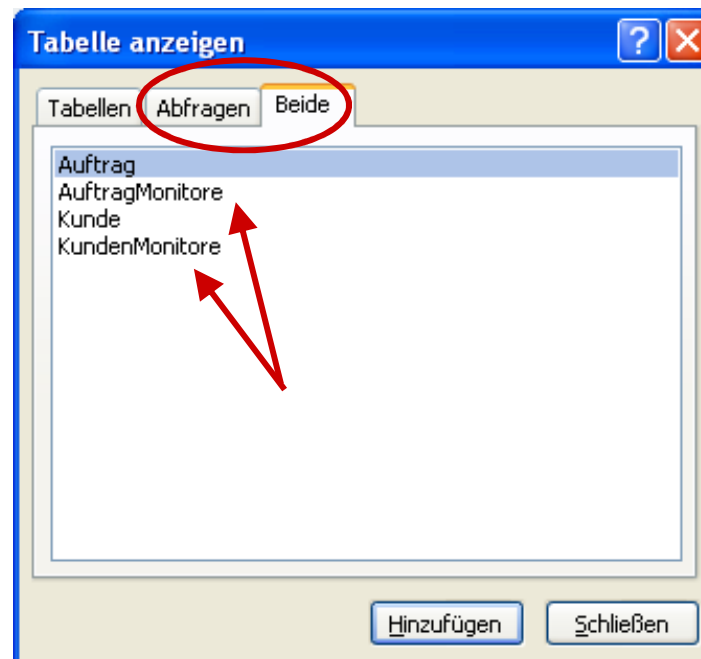
- Einen Namen für die Abfrage vergeben (KundenMonitore)
- [Doppel-Linksklick] auf dem Abfrageobjekt oder [Rechtsklick] / Öffnen führt die Abfrage auf dem aktuellen Datenbestand aus

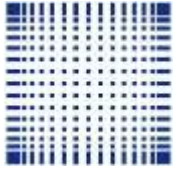
The screenshot shows a database application interface. On the left, a pane titled 'Alle Tabellen' (All Tables) lists tables under two categories: 'Kunde' and 'Auftrag'. Under 'Kunde', there is 'Kunde : Tabelle' and 'KundenMonitore'. Under 'Auftrag', there is 'Auftrag : Tabelle', 'AuftragMonitore', and 'KundenMonitore'. The 'KundenMonitore' entry is selected. On the right, a window titled 'KundenMonitore' displays a data table with three columns: 'KundenNr', 'Vorname', and 'Nachname'. The table contains two rows of data and a new row marked with an asterisk and '(Neu)'. The status bar at the bottom indicates 'Datensatz: 1 von 2' and 'Kein Filter'.

| KundenNr | Vorname | Nachname |
|----------|---------|----------|
| 1 | Werner | Maier |
| 2 | Walter | Müller |
| * | (Neu) | |



- Erinnerung an das relationale Datenmodell: Das Ergebnis aus einer Operation (z.B. einer Abfrage) auf Mengen (hier: Tabellen) ist wieder eine **Menge**
- ... also auch hier eine (wenn auch virtuelle) **Tabelle!**
- Folgerichtig kann eine Abfrage (genauer: das Ergebnis der Abfrage) wieder als Basis **z.B. für eine neue Abfrage** herangezogen werden:





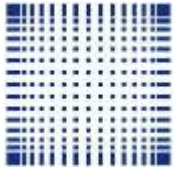
Welche Fragen gibt es?



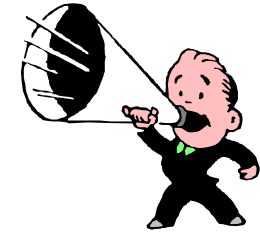
Zum Nachdenken:

Was passiert, wenn zum Zeitpunkt der Erstellung einer Beziehung zwischen zwei Tabellen schon falsche (inkonsistente) Daten in den Tabellen vorliegen?

Etwa ein Auftrag einer KundenNr, die nicht in der Tabelle Kunde erfasst ist – Ausprobieren!



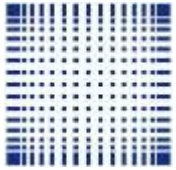
JETZT



Kapitel 7

Normalformen





Vorsicht *FALLE*



Was stimmt hier nicht?

| Postleitzahl | Ort |
|--------------|----------------------|
| 68161 | Mannheim |
| 68163 | Mannheim |
| 68199 | Mannheim |
| 38678 | Clausthal-Zellerfeld |
| 38678 | Mannheim |
| 38640 | Heidelberg |



- Ziel der Normalisierung eines DB-Schemas:
 - ◆ Anomalien („Widersprüchlichkeiten“) beheben
 - ◆ Redundanzen zu vermeiden
 - ◆ einen übersichtlichen und möglichst einfachen Aufbau der Tabellen zu erhalten
 - ◆ eine einfache Datenpflege zu ermöglichen



Voraussetzung: Eine Tabelle enthält logisch zusammengehörende Daten

Beispiel:

Für jeden Mitarbeiter werden erfasst:

- die Personalnummer, Name und Vorname,
- die Abteilungsnummer samt Abteilungsbezeichnung
- Die bearbeiteten Projekte mit Nummer, Bezeichnung und der jeweiligen Tätigkeit im Projekt

■ **Schlüssel?**



- Alle Daten werden in einer Tabelle gespeichert
- Da eine Person an mehreren Projekten gleichzeitig arbeiten kann, reicht die Personalnummer alleine nicht aus
- Um einen Primärschlüssel zu bilden wird zusätzlich die Projektnummer herangezogen
- Primärschlüssel = {PersonalNr, ProjNr}

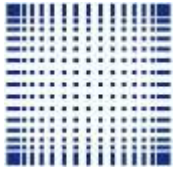


Tabelle MITARBEITER – 1

| | Attribut | Datentyp |
|---|---------------------|----------|
| ⌘ | PersonalNr | Zahl |
| | Name | Text |
| | Vorname | Text |
| | AbteilungsNr | Zahl |
| | AbtBezeichnung | Text |
| ⌘ | ProjNr | Zahl |
| | ProjektBeschreibung | Text |
| | Tätigkeit | Text |

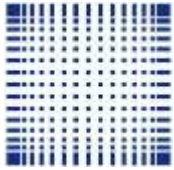


Tabelle MITARBEITER – 2

| Personal Nr | Name | Vorname | Abt Nr | Abt Bezeichnung | Proj Nr | Projekt Beschreibung | Tätigkeit |
|-------------|------------|-----------|--------|-----------------|---------|----------------------|----------------------------|
| 0001 | Lorenz | Sophia | 1 | Personal | | | |
| 0002 | Hohl | Tatjana | 2 | Einkauf | 1 | Kunden-umfrage | Leiter |
| 0002 | Hohl | Tatjana | 2 | Einkauf | 3 | Konkurrenz-analyse | Sach-bearbeiter |
| 0003 | Willschrei | Theodor | 2 | Einkauf | 1 | Kunden-umfrage | Sach-bearbeiter |
| 0004 | Richter | Hans | 3 | Verkauf | 2 | Verkaufs-messe | Leiter |
| 0004 | Richter | Hans | 3 | Verkauf | 3 | Konkurrenz-analyse | Leiter |
| 0005 | Wiesenland | Brunhilde | 3 | Verkauf | 2 | Verkaufs-messe | Präsentations-Vorbereitung |



- Bei einigen DBS müssen die Schlüsselattribute bei der Definition der Struktur direkt untereinander stehen
- Für das relationale Datenmodell hatten wir bisher immer atomare Werte gefordert
 - ◆ Ist das in unserem Beispiel der Fall?
 - ◆ Wäre nicht eine Liste von Projekten besser?
 - ◆ Tabelle atomarer Werte stellt die **erste Normalform (1NF)** dar



Unser Beispiel weist erhebliche Schwächen auf:

- Werden Tupel (Datensätze) geändert, neue eingefügt oder alte gelöscht können fehlerhafte Zustände (**Anomalien**) auftreten.
- Welche Ursache hat das?
 - ◆ → Redundanzen vermeiden und
 - ◆ → Anomalien erst gar nicht ermöglichen
 - ◆ sind ja gerade die Ziele der Normalisierung



- Durch Einfügen eines Mitarbeiters, der zu diesem Zeitpunkt an keinem Projekt arbeitet entstehen leere Datenfelder
- Verschwendung von Speicherplatz

Doch wir haben etwas übersehen:

Problem: ProjNr ist Teil des Schlüssels!

- Viele DBMS verweigern in diesem Fall die Übernahme in den Datenbestand
- Andere haben erhebliche Probleme beim Suchen und Verwalten dieses Datensatzes; es können Fehler auftreten



- Löschen Sie Mitarbeiter, so löschen Sie evtl. auch andere Informationen
- In unserem Beispiel: Löschen Sie einen Mitarbeiter, löschen Sie auch die Informationen zu dem von ihm bearbeiteten Projekt
- Noch schlimmer: War dieser Mitarbeiter der letzte an diesem Projekt arbeitende Mitarbeiter gehen damit die kompletten Informationen über ein Projekt verloren
- Semantisch falsch:
 - ◆ Da man auch Daten über abgeschlossene Projekte sammelt bzw.
 - ◆ Projekte, nur weil z.Zt. niemand an Ihnen konkret arbeitet, nicht abgeschlossen sein müssen



- Ändert sich der Familienname eines Mitarbeiters, so muss diese Änderung (**manuell !**) in allen Datensätzen durchgeführt werden, die diesen Wert enthalten
- Vorsicht bei **unterschiedlichen** Mitarbeitern **gleichen** Namens!
- Wird der Familienname nur in einem Datensatz geändert → Tabelle inkonsistent
- In diesem Fall existieren zu **einer** PersonalNr **zwei unterschiedliche** Familiennamen



- Offensichtlich beruhen die Ursachen von Anomalien auf den **Abhängigkeiten** zwischen Attributen
- Umgangssprachlich könnte man Abhängigkeit formulieren als: Ein Attribut legt ein anderes fest; ein Attribut bestimmt ein anderes
- Statt einzelnen Attributen sind auch Attributmengen möglich, die voneinander abhängig sind
- Diese Abhängigkeiten zu erkennen und zu beseitigen ist die Hauptaufgabe bei der Normalisierung
- Es gibt zwei Typen von Abhängigkeiten:
 - ◆ **Funktionale Abhängigkeiten**
 - ◆ **Funktional und transitive Abhängigkeiten**



Funktionale Abhängigkeiten

- Gegeben: Relation $R(A_1, A_2, \dots, A_n)$
- Gewählt: X, Y als zwei echte Teilmengen der Attributmenge (z.B. $X := \{A_4, A_5\}$, $Y := \{A_8\}$)
- Eine funktionale Abhängigkeit $X \rightarrow Y$ liegt vor,
 - ◆ Wenn es keine zwei Tupel in R geben kann, in denen für gleiche X -Werte verschiedene Y -Werte auftreten können
 - ◆ Der umgekehrte Fall (für gleiche Y -Werte verschiedene X -Werte) ist trotzdem erlaubt



Gegeben sei die folgende Tabelle:

LIEFERUNG(LieferNr, ArtikelNr, Artikelname, Anzahl, Lieferdatum, Lieferfirma, AnschriftLieferfirma)

Es bestehen funktionale Abhängigkeiten zwischen den Attributen:

■ **ArtikelNr** → **Artikelname**

Zu einer Artikelnummer gibt es immer genau einen Artikelnamen

■ **ArtikelNr** → **Lieferfirma**

Ein Artikel wird von einer bestimmten Lieferfirma geliefert
(Kann so sein, muss aber nicht → Semantischer Zusammenhang)

■ **Lieferfirma** → **AnschriftLieferfirma**

Jede Lieferfirma hat genau eine Anschrift. Umgekehrt könnten aber mehrere Lieferfirmen unter der gleichen Adresse residieren

■ **LieferNr** → **Lieferdatum**

Eine Lieferung erfolgt an einem bestimmten Tag
(eben dem Lieferdatum)

Immer dem
Kunden gut
zuhören!



Transitive Abhängigkeiten

Eine transitive Abhängigkeit besteht zusätzlich dann,

- wenn der Wert eines Nicht-Schlüssel-Attributes
- von einem oder mehreren Werten eines anderen Nicht-Schlüssel-Attributes abhängt

Beispiel:

LIEFERUNG(LieferNr, ArtikelNr, Artikelname, Anzahl, Lieferdatum, Lieferfirma, AnschriftLieferfirma)

- Kunde: „**Jeder Artikel wird uns von genau einer Firma geliefert. Es gibt keine Ersatzlieferanten.**“

Dann gilt die folgende transitive Abhängigkeit:

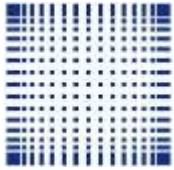
- Die Lieferfirma hängt von der Artikelnummer ab, die Lieferanschrift aber von der Lieferfirma

ArtikelNr → Lieferfirma → AnschriftLieferfirma



Normalisierungsprozess

- Die **Normalisierung** eines relationalen Schemas erfolgt in mehreren Schritten
- Dabei müssen die Daten in den Tabellen in jedem Schritt bestimmte Regeln erfüllen
- Das Resultat der Anwendung dieser Regeln heißt **Normalform**
- Durch die Normalisierung werden die Daten einer Tabelle ggf. auf mehrere Tabellen verteilt
- Die Schritte werden als 1. bis 5. Normalform bezeichnet
- In der Praxis ist eine Normalisierung bis zur 3. Normalform sinnvoll
- Darüber hinaus entstehen zu viele (kleine) Tabellen



Nicht-normalisierte Daten

| Personal Nr | Name | Vorname | Abt Nr | Abt Bezeichnung | Proj Nr | Projekt Beschreibung | Tätigkeit |
|-------------|---------|---------|--------|-----------------|---------------|------------------------------------------------------------|---------------------------|
| 0004 | Richter | Hans | 3 | Verkauf | 1, 2, 3 | Kunden- umfrage, Verkaufs- messe, Präsentation | Leiter, Sachbearbeiter |

- Nicht-normalisiert: Tupel enthalten Wertelisten (oben: ProjNr, ProjektBeschreibung, Tätigkeit)
- Tabelle ist schwierig auszuwerten:
 - ◆ Komplizierte Operation notwendig bei der Anfrage: "Welche Mitarbeiter sind an Projekt Nummer 3 beteiligt?"
 - ◆ Gar nicht auflösbar ist die Fragestellung: "Welche Tätigkeit übt Hr. Richter beim Projekt Nummer 2 aus?"



1. Normalform (1NF) – 1

Eine Tabelle befindet sich in der 1NF, wenn

- sie zweidimensional ist, d.h. ein Gebilde aus Zeilen und Spalten
- sich in jedem Datensatz nur Daten befinden, die **einem** Objekt der realen Welt gehören (Atomarität der Entität), und jeder Datensatz nur einmal vorkommt
- sich in jeder Spalte(nzeile) nur Daten befinden, die einem Attribut entsprechen (Atomarität des Attributs), und das Attribut nur einmal in der Tabelle vorkommt
- für jedes Attribut nur ein Wert (ggf. NULL) eingetragen ist



Zur Überführung in die 1NF

- Entfernen Sie alle Mehrfacheinträge (z.B. abstrakte Datentypen) in einem Attribut. Jedem Attribut eines Datensatzes darf höchstens ein (atomarer) Wert zugewiesen sein

Beispiel:

- Die zu Anfang dieses Kapitels betrachtete Tabelle MITARBEITER liegt bereits in 1NF vor



Probleme nach der 1NF

- Redundanzen: Mitarbeiterdaten, Abteilungsnamen und Projektnamen treten mehrfach auf
- Die Tabelle enthält voneinander unabhängige Sachgebiete, z.B. Mitarbeiter, Abteilungen, Projekte (Entitäten sind vermischt)
- Daten können nicht eindeutig identifiziert werden. Z.B. kann der Abteilungsname Einkauf nur über eine Personal- und Projektnummer ermittelt werden



2. Normalform (2NF)

Eine Tabelle befindet sich in 2NF, wenn

- jedes Nicht-Schlüsselfeld vom ganzen Primärschlüssel (evtl. aus mehreren Attributen) abhängig ist.
- Wichtig hierbei ist, dass die Datenfelder nicht nur von einem Teilschlüsselfeld, sondern vom gesamten Schlüsselfeld abhängig sind

Zur Überführung von der 1NF in die 2NF

- Zerlegen Sie die Tabelle in kleinere Tabellen, so dass in jeder Tabelle alle Nicht-Schlüsselfelder nur noch vom Primärschlüssel abhängen
- Besteht der Primärschlüssel aus mehreren Attributen, muss getestet werden, ob Attribute nur von einem Teil des Primärschlüssels abhängen → Dieser Teil des Primärschlüssels und die abhängigen Attribute sind in eine neue Tabelle zu überführen



Beispiel – 1

- Attribute Name, Vorname hängen nur von einem Teil des Schlüssels (PersonalNr) ab
- AbtNr wird ebenfalls durch PersonalNr bestimmt
- Alle anderen Attribute hängen nicht oder nur zum Teil vom Attribut PersonalNr ab
- ergibt die neue Tabelle:
PERSONAL(PersonalNr, Name, Vorname, AbtNr)

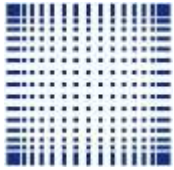
| | Attribut | Datentyp |
|---|------------|----------|
| 🔑 | PersonalNr | Zahl |
| | Name | Text |
| | Vorname | Text |
| | AbtNr | Zahl |



Beispiel – 2

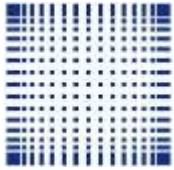
- Das Attribut AbtBezeichnung ist nicht vom Primärschlüssel abhängig,
- sondern vom Attribut AbtNr
- Kein weiteres Attribut hängt vom Attribut AbtNr ab
- ergibt die neue Tabelle:
ABTEILUNG(AbteilungsNr, AbtBezeichnung)

| | Attribut | Datentyp |
|---|----------------|----------|
| ⌘ | AbteilungsNr | Zahl |
| | AbtBezeichnung | Text |



- Analoge Vorgehensweise bei den Projekten
- Das Attribut Projektbeschreibung ist nur von einem Teil des Primärschlüssels (dem Attribut ProjNr) abhängig
- Kein weiteres Attribut hängt vom Attribut ProjNr ab
- ergibt die neue Tabelle:
PROJEKT(ProjNr, ProjektBeschreibung)

| | Attribut | Datentyp |
|---|---------------------|----------|
| ⌘ | ProjNr | Zahl |
| | ProjektBeschreibung | Text |



Beispiel – 4

- Die Tätigkeit, in der ein Mitarbeiter an einem Projekt arbeitet, ergibt sich nur aus der Kombination der Attribute ProjNr und PersonalNr
- Weitere Attribute der "Ursprungs-" Tabelle MITARBEITER sind nicht mehr zu betrachten
- ergibt die neue Tabelle:
ARBEITET_AN(ProjNr, PersonalNr, Tätigkeit)

| | Attribut | Datentyp |
|---|------------|----------|
| 🔑 | ProjNr | Zahl |
| 🔑 | PersonalNr | Zahl |
| | Tätigkeit | Text |



3. Normalform (3NF) – 1

Tabelle befindet sich in der 3NF,

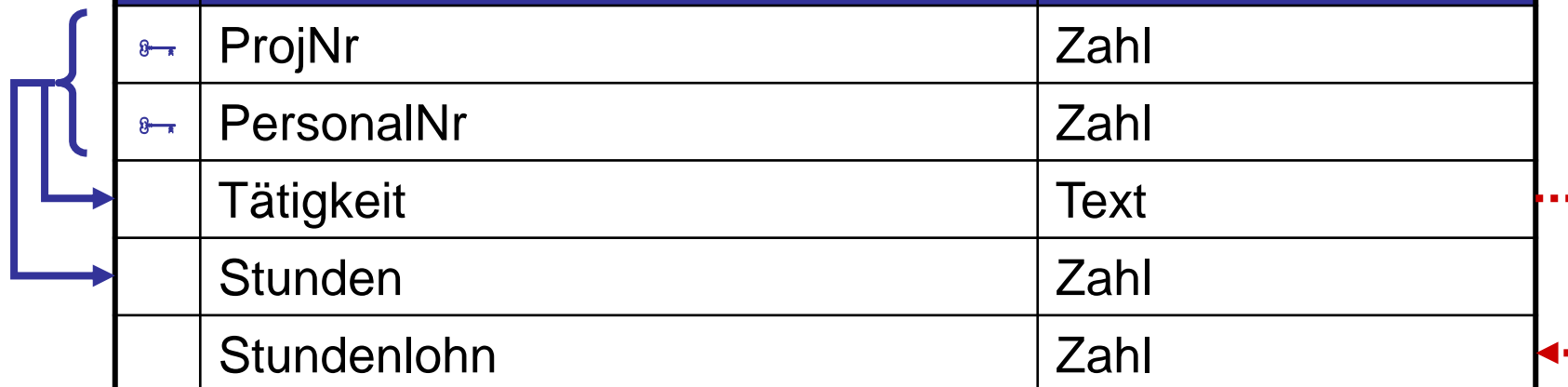
- wenn keine transitiven Abhängigkeiten mehr vorliegen
- d.h. kein Nicht-Schlüsselfeld mehr ausschließlich über ein anderes Nicht-Schlüsselfeld identifizierbar ist
- Klar: Transitive Abhängigkeiten verursachen ebenfalls Datenredundanz und -inkonsistenz
- Das bisher verwendete Beispiel liegt bereits in 3NF vor



3. Normalform (3NF) – 2

Betrachten wir aber eine Erweiterung von ARBEITET_ALS:

- Zusätzlich: die Attribute Stunden und Stundenlohn
- Der Stundenlohn ergibt sich aus der Art der Tätigkeit
- Damit ist der Stundenlohn nicht direkt, sondern transitiv vom Schlüssel {ProjNr, PersonalNr} abhängig



| | Attribut | Datentyp |
|---|-------------|----------|
| ⌘ | ProjNr | Zahl |
| ⌘ | PersonalNr | Zahl |
| | Tätigkeit | Text |
| | Stunden | Zahl |
| | Stundenlohn | Zahl |

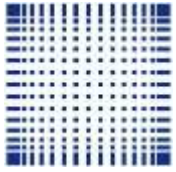


Tabelle ARBEITET_ALS

| PersonalNr | ProjektNr | Tätigkeit | Stunden | Stundenlohn |
|------------|-----------|--------------------------------|---------|-------------|
| 0002 | 1 | Leitung | 25 | 50,00 |
| 0003 | 1 | Bearbeitung | 55 | 30,00 |
| 0004 | 1 | Bearbeitung | 70 | 30,00 |
| 0004 | 2 | Leitung | 25 | 50,00 |
| 0005 | 2 | Präsentations- vorbereitung | 160 | 35,00 |
| 0004 | 3 | Leitung | 25 | 50,00 |
| 0002 | 3 | Bearbeitung | 80 | 30,00 |
| 0003 | 3 | Bearbeitung | 65 | 30,00 |



3. Normalform (3NF) – 3

Zur Überführung von der 2NF in die 3NF

- Beseitigen Sie alle transitiven Abhängigkeiten durch Aufteilen der Tabelle in mehrere Tabellen,
- in denen alle Nicht-Schlüsselfelder (nur noch) direkt vom gesamten Schlüsselfeld abhängig sind

In unserer Betrachtung:

- Die Tabelle ARBEITET_ALS wird in zwei Tabellen zerlegt
- Die Attribute Stundenlohn und Tätigkeit ergeben eine neue Tabelle TÄTIGKEIT(Tätigkeit, Stundenlohn)

↑
Textfeld !



- (Lange) Textfelder sind als Schlüsselfelder ungeeignet,
 - ◆ da sie mehr Speicherplatz für den Index benötigen
 - ◆ bei evtl. Beziehungen zwischen Tabellen komplizierter zu verknüpfen sind als Zahlen
 - Verknüpfung an sich als Operation
 - Änderung des Textfeldes bedingt auch eine Änderung des Textes in der bezugnehmenden Tabelle
- Abhilfe: Einführung zusätzlicher Schlüsselfelder, also
 - ◆ TÄTIGKEIT(TätigkeitsNr, Tätigkeit, Stundenlohn)
 - ◆ ARBEITET_ALS(ProjNr, PersonalNr, Stunden, **TätigkeitsNr**)



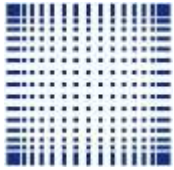
Tabelle ARBEITET_ALS

Tabelle ARBEITET_ALS

| | Attribut | Datentyp |
|---|--------------|----------|
| ⌘ | PersonalNr | Zahl |
| ⌘ | ProjNr | Zahl |
| | TätigkeitsNr | Zahl |
| | Stunden | Zahl |

Tabelle TÄTIGKEIT

| | Attribut | Datentyp |
|---|--------------|----------|
| ⌘ | TätigkeitsNr | Zahl |
| | Tätigkeit | Text |
| | Stundenlohn | Zahl |



- Offensichtlich zerfällt ein DB-Schema im Laufe des Normalisierungsprozesses in immer mehr Tabellen
- Es gibt weitere NF, die in der Praxis aber nur selten angewendet werden
- Jede weitere NF → weitere Tabellen → Geschwindigkeitsverlust bei Abfragen und Transaktionen
- Ziel: Ein **guter Kompromiss** zwischen der Speicherung von abhängigen Daten (evtl. auch redundanten Daten) und der Verarbeitungsgeschwindigkeit



Boyce-Codd-Normalform (BCNF) – 1

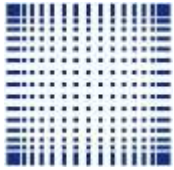
- Abhängigkeiten von einzelnen Schlüsseln oder Schlüsselattributen werden bis zur 3NF nicht berücksichtigt. Diese Abhängigkeiten werden in der BCNF beseitigt
- Eine Relation befindet sich in BCNF, wenn
 - ◆ kein Attribut funktional abhängig von einer Attributgruppe ohne Schlüsseleigenschaft ist



Boyce-Codd-Normalform (BCNF) – 2

Bedeutet:

- In einer Relation bestehen keine funktionalen Abhängigkeiten zwischen einem minimalen Schlüssel (nicht zusammengesetzten Schlüssel) und den Attributen
- Die meisten Relationen der 3NF befinden sich bereits in BCNF
- Nur Relationen,
 - ◆ die mehrere überlappende, minimale Schlüssel besitzen,
 - ◆ die eine funktionale Abhängigkeit erzeugen und somit wiederum Redundanzen hervorrufen,
 - ◆ sind zu überführen (= aufzuteilen)



Beispiel – 1

Tabelle PRÜFUNG

| PNr | MatNr | Fach | Note |
|-----|-------|---------------------------|------|
| 45 | 1234 | Datenbanken | 1.0 |
| 45 | 4711 | Datenbanken | 3.0 |
| 45 | 5678 | Datenbanken | 2.0 |
| 56 | 1234 | Grundlagen der Informatik | 4.0 |

- Primärschlüssel {PNr, MatNr}, Sekundärschlüssel {MatNr, Fach}
- Offensichtlich besteht eine 1:1 Beziehung zwischen der Prüfungsnummer PNr und dem Fach
- **einziges Nicht-Schlüsselattribut ist Note** (wirklich?), damit liegt PRÜFUNG in 3NF vor
- trotzdem Änderungsanomalie beim Attribut Fach



- Ziel der BCNF ist eben die Beseitigung der Anomalien für Schlüsselattribute
- Folgende Zerlegungen sind sinnvoll
 - ◆ PRÜFUNG(PNr, MatNr, Note)
PFACH(PNr, Fach) *oder*
 - ◆ PRÜFUNG(MatNr, Fach, Note)
PFACH(PNr, Fach)
- Beide Zerlegungen führen zu BCNF-Relationen, die Änderungsanomalie ist verschwunden



4. + 5. Normalform (4NF + 5NF)

- In der 4NF werden mehrwertige Abhängigkeiten von Attributmengen zu einem sog. Superschlüssel (übergeordneten Schlüssel) eliminiert
- Ist eine verlustlose Zerlegung in Einzelabhängigkeiten in der 4NF nicht möglich, werden in der 5NF weitere Primärschlüssel hinzugefügt.
- Dies wird solange wiederholt, bis nur noch Einzelabhängigkeiten der Attribute von einem oder anderen Primärschlüssel bestehen (keine Join Dependencies mehr)
- 4NF, 5NF führen genauso wie die BCNF zu einer größeren Menge von Relationen



Beispiel – 1

- Die räumliche Zuordnung von Arbeitsgruppen und Arbeitsplätzen ist in einer Relation gespeichert:

| Arbeitsgruppe | Raum | Arbeitsplatz |
|---------------|------|--------------|
| 1 | 1 | 1 |
| 1 | 1 | 2 |
| 1 | 2 | 1 |
| 2 | 1 | 4 |
| 2 | 1 | 5 |
| 3 | 3 | 1 |
| 3 | 3 | 2 |
| 4 | 1 | 1 |

- Relation befindet sich in 3NF, **da einziger Primärschlüssel die Kombination aller Attribute**
- Trotzdem Redundanzen: Jede Arbeitsgruppe benötigt mehrere Arbeitsplätze und in jedem Raum gibt es mehrere Arbeitsplätze.

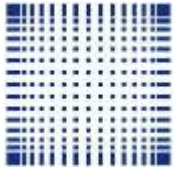


Beispiel – 2

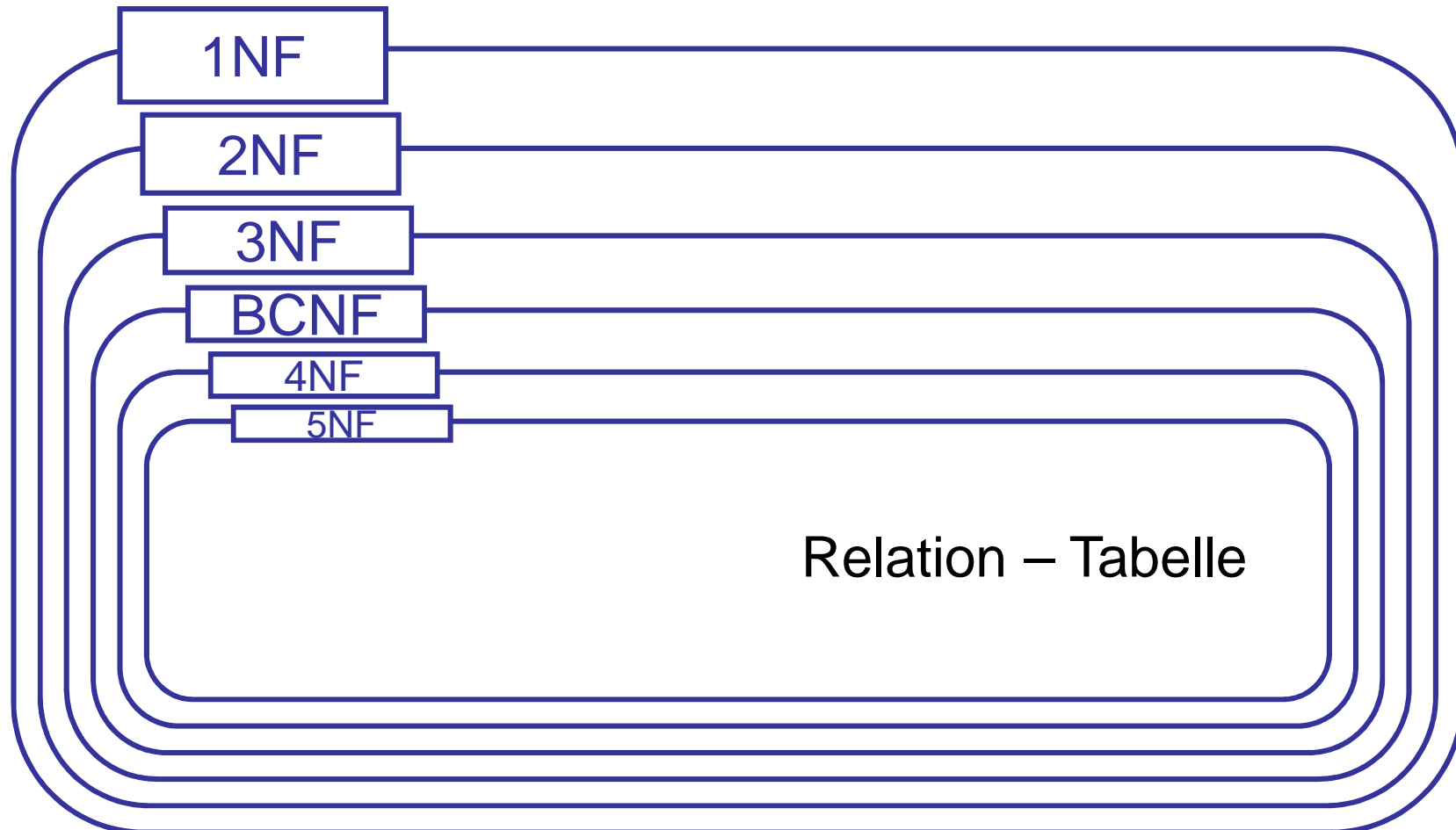
- Der Primärschlüssel besitzt somit **mehrwertige Abhängigkeiten**
- Diese lassen sich durch Aufteilung in zwei Relationen beseitigen
- Die Ergebnisrelationen befinden sich dann in 4NF und 5NF, da eine verlustfreie Zerlegung möglich ist.

| Arb_Raum | Arbeitsgruppe | Raum |
|----------|---------------|------|
| 11 | 1 | 1 |
| 12 | 1 | 2 |
| 21 | 2 | 1 |
| 33 | 3 | 3 |
| 41 | 4 | 1 |

| Arb_Raum | Arbeitsplatz |
|----------|--------------|
| 11 | 1 |
| 11 | 2 |
| 12 | 1 |
| 21 | 4 |
| 21 | 5 |
| 33 | 1 |
| 33 | 2 |
| 41 | 1 |



Zusammenhang der Normalformen





Welche Fragen gibt es?



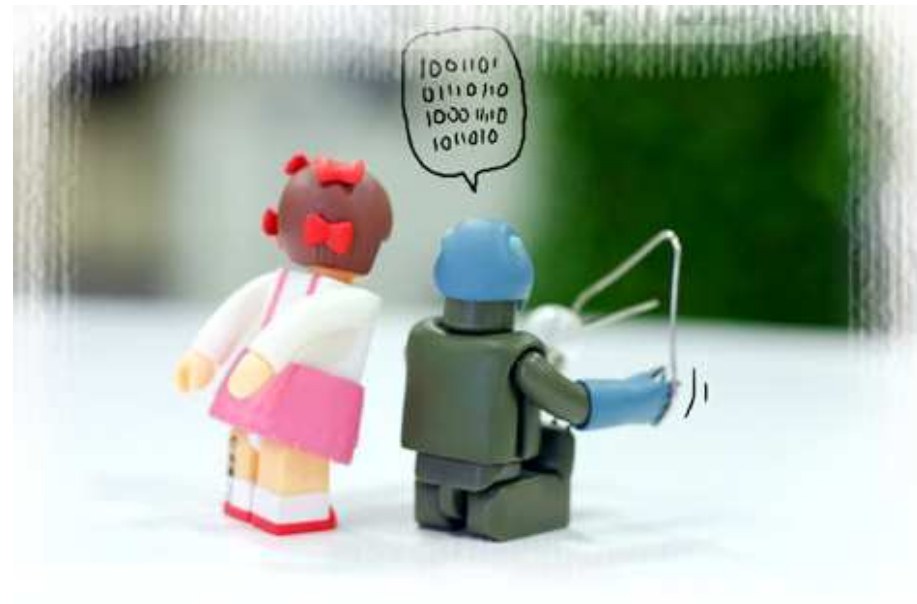
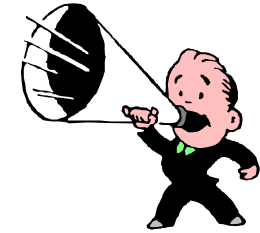
Zum Nachdenken:

Wann verwenden Sie den Mechanismus der Normalisierung, wann ist es vielleicht besser ein ER-Diagramm zu erstellen?

Welche Entwurfsmethode verwenden Sie, wenn noch gar kein Datenbankschema vorliegt?



JETZT



Kapitel 8

Structured Query Language (SQL)



Minimale Lösung – nur MySQL Client

■ Pool Rechner

- ◆ unter Linux booten
- ◆ Terminal starten
- ◆ mysql steht als Kommando zur Verfügung

■ Windows oder Mac OS Rechner zu Hause

- ◆ dev.mysql.com
- ◆ Downloads: MySQL Workbench

■ Linux Rechner zu Hause

- ◆ `sudo apt-get install mysql-client` (← Ubuntu, Debian)
- ◆ oder die MySQL Workbench



Maximale Lösung – eigenen MySQL Server

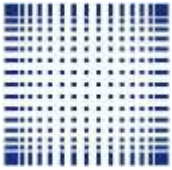
- Windows oder Mac OS Rechner zu Hause
 - ◆ XAMPP installieren (Angaben gleich)
 - ◆ gibt es sogar USB Stick friendly
- Linux Rechner zu Hause
 - ◆ `sudo apt-get install mysql-server` (← Ubuntu, Debian)

Komfortables Addon – eine Benutzeroberfläche

- für alle Konfigurationen
 - ◆ PHPMYAdmin (www.phpmyadmin.net) installieren
 - ◆ für Linux ist auch das Paket `phpmyadmin` verfügbar



- Ein komplettes Paket für die Entwicklung von Webanwendungen: <http://www.apachefriends.org>
- Ein Paket mit
 - ◆ MySQL
 - ◆ Apache, PHP
 - ◆ aber auch mit PHPmyAdmin, Perl etc.
 - ◆ in aktuellen Versionen
 - ◆ vorkonfiguriert (aber nur für Testzwecke! Sicherheit!)
 - ◆ Für Windows, Linux und MacOS
- Wenn der Rechner ins Internet geht entweder den MySQL Server herunter fahren oder (besser!) die Installation absichern!



MySQL Workbench – 1

The screenshot displays the MySQL Workbench interface. The top menu bar includes File, Edit, View, Query, Database, Server, Tools, Scripting, and Help. The left sidebar contains a Navigator pane with sections for MANAGEMENT (Server Status, Client Connections, Users and Privileges, Status and System Variables, Data Export, Data Import/Restore), INSTANCE (Startup / Shutdown, Server Logs, Options File), and SCHEMAS (Filter objects, phpmyadmin, **vorlesung**, Tables, Views, Stored Procedures, Functions). The main workspace shows a SQL File named 'vorlesung' with the query `select * from mitarbeiter;`. Below the query editor, the 'Result Set Filter' section displays a table with 5 rows and 5 columns: `personalnr`, `nachname`, `vorname`, `abteilung`, and `anschrift`. The 'Output' pane at the bottom shows the execution log with the message: '1 20:13:21 select * from mitarbeiter LIMIT 0, 1000 5 row(s) returned 0.000 sec / 0.000 sec'. The 'SQL Additions' pane on the right shows the syntax for the `SELECT` statement.

| personalnr | nachname | vorname | abteilung | anschrift |
|------------|------------|-----------|-----------|---------------------------------|
| 1 | Lorenz | Sophia | 1 | 03725 Mausbach Mühlenweg 4 |
| 2 | Hohl | Tatjana | 2 | 49262 Mausloch Käsereistr. 5 |
| 3 | Willschrei | Theodor | 2 | 03453 Katzbergen Ahornweg 4 |
| 4 | Richter | Hans | 3 | 78943 Katzenhausen Buchenweg 28 |
| 5 | Wiesenland | Brunhilde | 3 | 02518 Hundsberg Mopsstr. 45 |

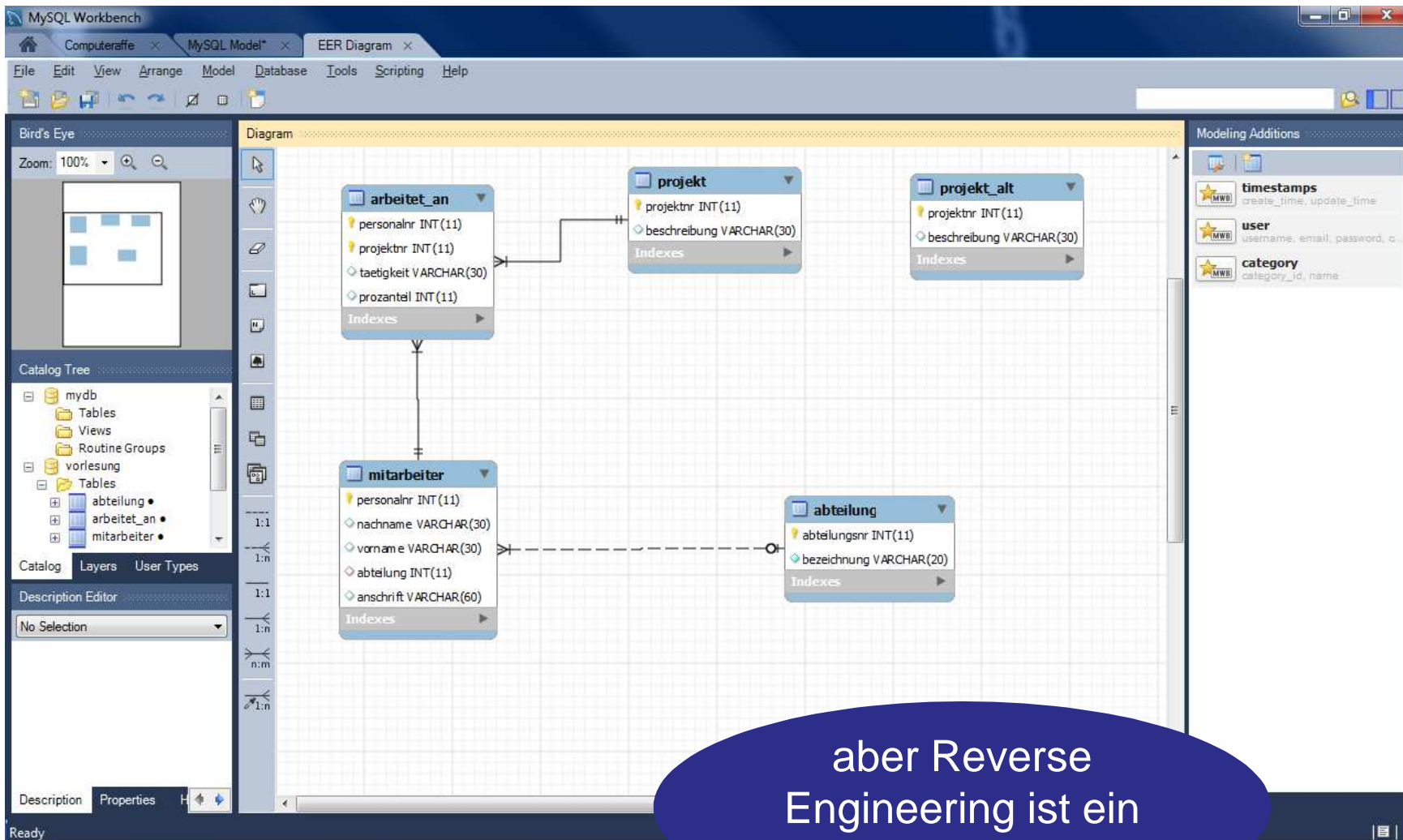
Output

| Time | Action | Message | Duration / Fetch |
|------------|-----------------------------------------|-------------------|-----------------------|
| 1 20:13:21 | select * from mitarbeiter LIMIT 0, 1000 | 5 row(s) returned | 0.000 sec / 0.000 sec |

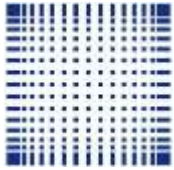
wirkt eher wie ein
Administrationstool



MySQL Workbench – 2



aber Reverse Engineering ist ein mächtiges Tool



PHPMyAdmin – 1

computeraffe / localhost / x

computeraffe/phpmyadmin/index.php?token=9b24c0181c3041...

phpMyAdmin

(Letzte Tabellen) ...

information_schema
mysql
performance_schema
phpmyadmin
vorlesung
 Neu
 abteilung
 arbeitet_an
 mitarbeiter
 projekt
 projekt_alt

Server: localhost » Datenbank: vorlesung

Struktur SQL Suche Abfrage Exportieren Importieren Operationen Rechte Mehr

| Tabelle | Aktion | Datensätze | Typ | Kollation | Größe | Überhang |
|--------------------------------------|-------------------------------------------------|------------|--------|-------------------|---------|----------|
| <input type="checkbox"/> abteilung | Anzeigen Struktur Suche Einfügen Leeren Löschen | ~3 | InnoDB | latin1_swedish_ci | 16 KiB | - |
| <input type="checkbox"/> arbeitet_an | Anzeigen Struktur Suche Einfügen Leeren Löschen | ~8 | InnoDB | latin1_swedish_ci | 32 KiB | - |
| <input type="checkbox"/> mitarbeiter | Anzeigen Struktur Suche Einfügen Leeren Löschen | ~5 | InnoDB | latin1_swedish_ci | 32 KiB | - |
| <input type="checkbox"/> projekt | Anzeigen Struktur Suche Einfügen Leeren Löschen | ~3 | InnoDB | latin1_swedish_ci | 16 KiB | - |
| <input type="checkbox"/> projekt_alt | Anzeigen Struktur Suche Einfügen Leeren Löschen | ~2 | InnoDB | latin1_swedish_ci | 16 KiB | - |
| 5 Tabellen Gesamt | | 21 | InnoDB | latin1_swedish_ci | 112 KiB | 0 B |

☐ Alle auswählen markierte:

Druckansicht Strukturverzeichnis

Erzeuge Tabelle

Name: Anzahl der Spalten:

OK

muss auf dem gleichen
Rechner wie der MySQL
Server installiert werden

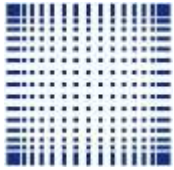


PHPMyAdmin – 2

The screenshot shows the PHPMyAdmin interface in a web browser. The left sidebar displays a tree view of databases, with 'vorlesung' selected. The main area shows a query result for the 'mitarbeiter' table. The query is 'SELECT * FROM mitarbeiter LIMIT 0, 30'. The result is a table with 5 rows and 6 columns: personalnr, nachname, vorname, abteilung, and anschrift. Each row has action buttons (Bearbeiten, Kopieren, Löschen) to its left. The interface also includes a top navigation bar with options like 'Struktur', 'SQL', 'Suche', 'Abfrage', 'Exportieren', 'Importieren', 'Operationen', 'Rechte', and 'Mehr'. A status bar at the bottom indicates 'Zeige : Anfangs-Datensatz: 0 Anzahl der Datensätze: 30 Kopfzeilen alle 100 Zeilen'.

| | personalnr | nachname | vorname | abteilung | anschrift |
|--------------------------------------------------------------------------------------------------------|------------|------------|-----------|-----------|---------------------------------|
| <input type="checkbox"/> Bearbeiten <input type="checkbox"/> Kopieren <input type="checkbox"/> Löschen | 1 | Lorenz | Sophia | 1 | 03725 Mausbach Mühlenweg 4 |
| <input type="checkbox"/> Bearbeiten <input type="checkbox"/> Kopieren <input type="checkbox"/> Löschen | 2 | Hohl | Tatjana | 2 | 49262 Mausloch Käsereistr. 5 |
| <input type="checkbox"/> Bearbeiten <input type="checkbox"/> Kopieren <input type="checkbox"/> Löschen | 3 | Willschrei | Theodor | 2 | 03453 Katzbergen Ahornweg 4 |
| <input type="checkbox"/> Bearbeiten <input type="checkbox"/> Kopieren <input type="checkbox"/> Löschen | 4 | Richter | Hans | 3 | 78943 Katzenhausen Buchenweg 28 |
| <input type="checkbox"/> Bearbeiten <input type="checkbox"/> Kopieren <input type="checkbox"/> Löschen | 5 | Wiesenland | Brunhilde | 3 | 02518 Hundsberg Mopsstr. 45 |

unterstützt besser die Arbeit
des Datenbankentwicklers



Vorsicht *FALLE*



- SQL ist formatfrei. Alle Anweisungen könnten in einer Zeile fort geschrieben werden, dient aber nicht der Übersichtlichkeit und sollte vermieden werden
- SQL unterscheidet Groß-/Kleinschreibung **NICHT**
- **SQL ≠ SQL**
 - ◆ betrifft z.B. die Wertebereiche der Datentypen
 - ◆ viele Syntaxkonstrukte, sogar ganze Befehle
 - ◆ Die Normung von SQL ist nur eine Empfehlung!
 - ◆ Funktioniert etwas nicht, ist (intensives) Handbuchstudium des verwendeten SQL-Servers angesagt



- Structured Query Language
- entstanden aus SEQUEL (Structured English QUery Language; IBM)
- als Vereinfachung: einfacher verständlich, Formulierung von Abfragen ohne Quantoren
- 1989 das erste Mal vom ANSI genormt (SQL1; SQL-89)
- 1992 erfolgte die erneute Normung durch ISO und ANSI der inzwischen erweiterten SQL als ISO/IEC 9075:1992; DIN 66315; SQL2; SQL-92 ... Danach immer mal wieder „Updates“
- Drei Conformance Level
 - ◆ Entry Level: etwa SQL-89; grundlegende Befehle zum Anlegen, Verwalten von Datenbanken, Tabellen
 - ◆ Intermediate Level: zusätzlich Datum/Zeit-Typ; Mengenoperationen
 - ◆ Full Level: zusätzlich weitere Funktionen (z.B. Integritätsbedingungen über mehrere Tabellen), die z.T. bis heute nicht implementiert sind



1. Datenbank anlegen: Container für die Tabellen und alle weiteren Datenbankobjekte
2. Tabellen erstellen: Festlegung der Struktur durch Angabe der Attribute und dazugehörigen Wertebereiche etc.
3. Eingabe der Daten (in die Tabellen): Tabellen sind die einzigen Datenbankobjekte, die physisch Daten speichern können
4. Auswertung der Daten: Durch Abfragen etc. können Daten aufgerufen, verändert, gelöscht werden



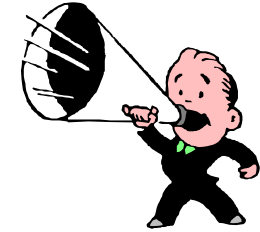
- Die Eingabe von SQL-Anweisungen kann sich über mehrere Zeilen erstrecken,
 - ◆ daher bedarf es eines Terminierungszeichens für Anweisungen
 - ◆ vergleichbar zu Programmiersprachen wie C, C++, Java etc.
 - ◆ wird bei den meisten RDBMS das Semikolon als Terminierungszeichen eingesetzt

- Bei den folgenden Syntaxdiagrammen sollen folgende Konventionen gelten
 - ◆ [...] optionale Anteile
 - ◆ ... | ... alternative Anteile

- SQL-Schlüsselworte dürfen nicht als Bezeichner für Datenbankobjekte (Datenbanken, Tabellen, ...) verwendet werden



JETZT



Abschnitt 8.1

SQL als DDL



Vom Umgang mit Datenbanken

■ **SHOW DATABASES;**

Die Anweisung listet alle auf dem Server vorhandenen Datenbanken auf

- ◆ funktioniert *sofort* nach der Installation von MySQL
- ◆ Warum?

■ **USE datenbankname;**

wählt die angegebene Datenbank aus und öffnet sie zur Verwendung

- Bei anderen SQL-Servern ist ebenfalls der Befehl
CONNECT datenbankname USER name PASSWORD pass;
verbreitet
- In MySQL erfolgt die Authentifizierung beim Start des Clients
`mysql -p -u benutzername [-h hostname]`



CREATE DATABASE – 1

Erzeugen einer DB

- **CREATE DATABASE [IF NOT EXISTS]** datenbankname;
- CREATE erzeugt ein Datenbank-Objekt, hier eine DATABASE
- Am Ende der Anweisung wird der Datenbankname angegeben
- Existiert die DB bereits → Fehlermeldung. Diese kann durch die optionale Angabe von IF NOT EXISTS unterdrückt werden
- IF NOT EXISTS wird nicht von allen SQL-Servern unterstützt
- Sie müssen die Rechte besitzen, Datenbanken anlegen zu dürfen
- Durch das Anlegen einer Datenbank werden Sie deren Besitzer



CREATE DATABASE – 2

- Für die Vergabe von Datenbanknamen und anderen Bezeichnern legt jeder SQL-Server seine eigenen Regeln fest
- In MySQL darf der Datenbankname bis zu 64 Zeichen lang sein und alle in einem Verzeichnisnamen erlaubten Zeichen enthalten
 - ◆ MySQL speichert Datenbanken als Unterverzeichnis des Installationsverzeichnisses
 - ◆ Vorsicht Stolperfalle – Portierung einer MySQL DB von einem LINUX auf ein Windows System und umgekehrt:
 - Groß-/Kleinschreibung
 - erlaubte Zeichen etc.



■ Ganzzahlige Werte

- ◆ **SMALLINT** (2 Byte; -32768 bis 32767)
- ◆ **INTEGER** (4 Byte; -2147483648 bis 2147483647)
- ◆ besonders schneller Zugriff
- ◆ gut geeignet auch für Primärschlüssel

■ Fließkommazahlen

- ◆ **FLOAT** (4 Byte; 7 signifikante Stellen)
- ◆ **DOUBLE PRECISION** (8 Byte; 15 signifikante Stellen)
- ◆ Es gelten ähnliche Regeln für die Genauigkeit der Fließkommazahlen wie in Java



■ Festkommazahlen

- ◆ **NUMERIC**(Skalierung, Präzision)
- ◆ **DECIMAL**(Skalierung, Präzision)
- ◆ Parameter Skalierung (1-15) bestimmt die Anzahl der Ziffern links vom Dezimalkomma (-punkt !)
- ◆ Parameter Präzision (1-15) legt die Anzahl der Nachkommastellen fest, die gespeichert werden
- ◆ bei **NUMERIC** werden *exakt* die in Präzision angegebenen Nachkommastellen gespeichert
- ◆ bei **DECIMAL** werden *mindestens* die in Präzision angegeben Nachkommastellen gespeichert



- **DATE** (8 Byte; 1.1.100 bis 11.12.5941)
- speichert Datumswerte, aber auch Zeitangaben
- Eingabeformat Datum
 - tt.mm.jjjj (12.09.2003)
 - tt-mmm-jjjj (12-SEP-2003)
 - mm-tt-jjjj (09-12-2003)
- Das Jahr kann auch zweistellig angegeben werden, es wird dann das aktuelle Jahrhundert angenommen
- Monatsnamen: JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC
- Eingabeformat Zeit
 - hh:mm:ss (17:34:55)
- Wird beides angegeben, so steht das Datum zuerst, gefolgt von einem Leerzeichen und schließlich die Zeit



Zeichen und Texte

- Es gibt Datentypen mit variabler und fester Länge
- **CHAR(Länge)** – dient zum Speichern beliebiger Textinformationen; maximale Länge wird als Parameter übergeben. Diese Anzahl Zeichen wird stets gespeichert
- **VARCHAR(Länge)** – genau wie **CHAR**, aber es wird nur die wirklich benötigte Anzahl Zeichen (+ 2 Byte Längeninformation) pro Tupel gespeichert
- **VARCHAR** sollte nicht bei sehr kurzen Informationen sondern nur bei stark (in der Länge) variierenden Attributwerten verwendet werden
- Der Zugriff auf **CHAR** ist schneller, verbraucht dafür aber auch mehr Speicherplatz



Zeichen und Texte – 2

- **BLOB** – Dieser Datentyp wird zum Speichern großer, auch binärer Datenmengen eingesetzt, z.B. sehr großer Textdateien, Grafiken, Bilder oder Videos
- **TEXT** – wie **BLOB**, lediglich die Sortierreihenfolge ist bei **TEXT**-Attributen unabhängig von der Groß-/Kleinschreibung
- Der Einsatz dieser Datentypen sollte genau abgewogen werden, im Hinblick auf
 - ◆ die Größe der DB
 - ◆ die Zugriffsgeschwindigkeit bei Abfragen



Datentypumwandlung

- Datentypen lassen sich ineinander umwandeln
- Notwendig bei
 - ◆ Berechnung von Ausdrücken
 - ◆ Zuweisungen von Werten (Literal oder Attributwert) an einen Attributwert von einem anderen Typ
- Typumwandlung erfolgt nach ähnlichen Regeln wie in Java
- Implizite Umwandlung: Konvertierung vom "niedrigeren" in den "höheren" Datentyp (z.B. **SMALLINT** in **INTEGER**)
- Explizite Umwandlung:
CAST (Wert AS Datentyp)
Die Funktion wandelt den angegebenen Wert in den Datentyp um, der nach dem Schlüsselwort **AS** angeführt ist
- **BLOB**, **TEXT** können nicht umgewandelt werden.
Andere **CASTs** sind implementationsabhängig



CREATE TABLE – 1

- Tabellen sind die einzigen Datenbankobjekte , in denen Daten gespeichert werden können
- Jeder Zugriff auf Daten erfolgt über Tabellen
- Bei jede Abfrage oder Auswertung muss der Name der Tabelle (mit) angegeben werden

- Datenbank = Container für logisch zusammengehörende Tabellen
- Innerhalb einer Datenbank muss der Tabellennamen eindeutig sein,
- darf aber in mehreren Datenbanken eines DBMS verwendet werden

- Sie müssen die Rechte besitzen, um Tabellen erstellen zu können
- Durch das Anlegen einer Tabelle werden Sie deren Besitzer



CREATE TABLE – 2

- Eine Tabelle mit den zugehörigen Attributen erstellen Sie mit dem Befehl **CREATE TABLE**
- Dieser Befehl bietet viele weitere Optionen, z.B.
 - ◆ Festlegung von Schlüsseln
 - ◆ Erstellen von Indizes
 - ◆ Definieren von Standardwerten
 - ◆ Anlegen von Integritätsbedingungen
- Im Folgenden soll eine Tabelle mit vier Feldern zum Speichern von Mitarbeiterdaten angelegt werden
- Gespeichert werden die Personalnummer, der Vorname, der Nachname und die Adresse



CREATE TABLE – 3

- (1) **CREATE TABLE** t_ma
 - (2) (id **INTEGER NOT NULL**,
 - (3) vname **VARCHAR**(100), name **VARCHAR**(100),
 - (4) adr **TEXT**);
-

- (1) Die neue Tabelle erhält den Namen t_ma. In den Klammern erfolgt die Definition der Datenfelder
- (2) Das erste Attribut heißt id und ist vom Typ **INTEGER**, kann also ganze Zahlen speichern. Gleichzeitig wird festgelegt, dass dieses Datenfeld nicht leer bleiben darf (**NOT NULL**)
- (3) Die Felder vname und name sind als Zeichenketten mit einer maximalen Länge von 100 Zeichen angelegt. Beim Datentyp **VARCHAR** werden nur die aktuell benötigten Zeichen gespeichert
- (4) Das letzte Attribut adr ist vom Typ **TEXT** und ermöglicht die Speicherung größerer Textmengen variabler Länge



- Der Datentyp **TEXT** steht nicht in allen RDBMS zur Verfügung
- Tabellennamen lassen sich von anderen Datenbankobjekten (Abfragen, Stored Procedures etc.) leichter unterscheiden durch das Voranstellen eines Präfix, z.B. t_ für Tabellen, i_ für Indizes, v_ für Sichten
- Die Wahl kurzer Tabellennamen (etwa Abkürzungen oder sinnvolle Akronyme) erleichtert die Tipparbeit, aber es existieren weitaus weniger Konventionsvorgaben im Vergleich zur Programmiersprache Java
- Die maximale Länge eines Tabellennamens ist vom RDBMS abhängig



CREATE TABLE – 4

■ Syntax

CREATE TABLE tabellenname

(datenfeld1 datentyp1 [**DEFAULT** wert1|**NULL**|**NOT NULL**] [**AUTO_INCREMENT**],

...

datenfeldn datentypn [**DEFAULT** wertn|**NULL**|**NOT NULL**] [**AUTO_INCREMENT**],

[**PRIMARY KEY**(datenfeldliste)]) ;

- Mit der Anweisung **CREATE TABLE** wird eine neue leere Tabelle erstellt. Danach folgt der gewünschte Bezeichner
- In runden Klammern folgen die Definition der einzelnen Datenfelder. Für jedes Datenfeld muss dabei ein eindeutiger Name innerhalb der Tabelle und ein Datentyp angegeben werden
- Mit der Angabe **PRIMARY KEY** kann eine Menge von Attributen (durch Kommata getrennte Liste) als Primärschlüssel ausgezeichnet werden. Bei manchen RDBMS ist diese Angabe zwingend
- MySQL quittiert die erfolgreiche Ausführung mit der Meldung: Query ok



Die optionalen Parameter:

- **NOT NULL**

Mit diesem Parameter wird die Eingabe eines Wertes für das entsprechende Datenfeld erzwungen. Diese Angabe ist für Schlüsselfelder zwingend

- **NULL** (auch: DEFAULT NULL)

Dieses Datenfeld erhält standardmäßig keinen Wert (also auch nicht 0 oder die leere Zeichenkette). Dies entspricht in SQL dem Wert **NULL**



CREATE TABLE – 6

■ **DEFAULT** standardwert

Dieser Parameter definiert einen Standardwert, den dieses Datenfeld annimmt, sofern kein Wert eingegeben wird

Beispiel:

```
...  
name VARCHAR(100) DEFAULT "Meier",  
...
```

■ **AUTO_INCREMENT** (MySQL-spezifisch)

Der Wert dieses Datenfeldes ergibt sich durch Erhöhung um 1 des gleichen Datenfeldes des letzten Datensatzes. Dieser Wert kann vom Benutzer nicht verändert werden und die Option ist nur auf Ganzzahlen anwendbar. Sie ist besonders für den Primärschlüssel geeignet, da auf diese Weise eindeutige Schlüsselwerte erzeugt werden



- Andere RDBMS haben andere **AUTO_INCREMENT** Parameter oder Mechanismen
- Andere RDBMS bieten ggf. mehr oder weniger Möglichkeiten die Tabellenstruktur festzulegen
- Neben dem Tabellennamen verlangt die **CREATE TABLE**-Anweisung die Definition mindestens eines Datenfeldes



- Bereits beim Erstellen einer Tabelle können Regeln für die Daten definiert werden
- Eine Gültigkeitsprüfung wird durch das Schlüsselwort **CONSTRAINT** (engl., Zwang, Nebenbedingung) eingeleitet
- Dadurch erfolgt bereits bei der Eingabe von Werten eine semantische Prüfung
- Eine weitere Möglichkeit: **Berechnete Attribute**, die nicht in der Datenbank gespeichert werden, aber so Platz sparen und die Gefahr von Inkonsistenzen mindern

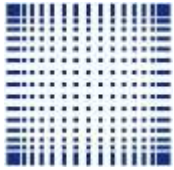


Beispiel – 1

- In einer (neuen) Tabelle der Mitarbeiter, soll sichergestellt werden, dass die id immer größer als 10 ist

```
CREATE TABLE t_ma  
  (id INTEGER NOT NULL,  
   vname VARCHAR(100),  
   name VARCHAR(100),  
   adr TEXT,  
   CONSTRAINT c_pruef CHECK(id > 10) );
```

- Die Gültigkeitsprüfung erhält ebenfalls einen durch den Benutzer festzulegenden Namen (c_pruef)
- Die Korrektheit der Gültigkeitsprüfung erfolgt in diesem Beispiel durch die Angabe eines Prädikats (id > 10)



- In einer Tabelle t_lager soll für jeden Artikel die vorhandene Stückzahl und der Preis gespeichert werden, hieraus ergibt sich der aktuelle Warenwert dieses Artikels

```
CREATE TABLE t_lager  
  (id INTEGER NOT NULL,  
   stueck INTEGER DEFAULT 1,  
   preis FLOAT NOT NULL,  
   wert COMPUTED BY(stueck * preis) );
```

- Das Datenfeld wert ist ein berechnetes Attribut (berechnetes Datenfeld). Der Wert ergibt sich durch die Angabe eines Ausdrucks
- Damit die Berechnung aber überhaupt durchgeführt werden kann, dürfen die Felder stueck und preis nicht **NULL** sein



- Berechnete Attribute dürfen bei der Dateneingabe nicht mit Werten befüllt werden
- Berechnete Attribute werden von MySQL (noch) nicht unterstützt
- Abhilfe: Es gibt einen vergleichbaren Mechanismus, der berechnete Attribute innerhalb von Abfragen ermöglicht



Syntax – Innerhalb einer **CREATE TABLE** Anweisung:
datenfeld **COMPUTED BY** (berechnung),
CONSTRAINT name1 **CHECK** (bedingung),

- Für berechnete Attribute erfolgt keine Angabe eines Datentyps – er ergibt sich durch die Berechnung

Beispiele für Bedingungen:

- nummer <= 100
- name NOT LIKE "%Ä%"
wahr, wenn name den Buchstaben Ä nicht enthält
- abteilung IN ("Einkauf", "Verkauf")
wahr, wenn abteilung den Wert Einkauf oder Verkauf trägt



SHOW TABLES

Anzeige aller Tabellen

SHOW TABLES [**FROM** dbname] [**LIKE** "muster"];

- Die Anweisung **SHOW TABLES** zeigt ohne die opt. Parameter eine Liste der vorhandenen Tabellen in der aktuell geöffneten Datenbank an
- Mit dem Schlüsselwort **FROM** kann eine andere DB ausgewählt werden
- Mit **LIKE** kann die Anzeige der Tabellen auf ein bestimmtes Muster eingeschränkt werden. Weiteres zu Mustern später



ALTER TABLE – 1

- Die Struktur einer Tabelle können Sie jederzeit über die Anweisung **ALTER TABLE** ändern
- Hierbei können Sie
 - ◆ Datenfelder hinzufügen oder löschen
 - ◆ Datenfelddefinitionen verändern
 - ◆ Gültigkeitsprüfungen hinzufügen oder löschen
 - ◆ Schlüssel hinzufügen oder löschen
- **ADD** fügt hinzu
- **DROP** löscht



Beispiel – 1

- Der Tabelle t_lager wird das neue Datenfeld artikel mit dem Standardwert unbekannt hinzugefügt
ALTER TABLE t_lager
ADD artikel VARCHAR(20) DEFAULT "unbekannt";
- Nachträglich wird das Feld id in der Tabelle t_ma als Primärschlüssel festgelegt
ALTER TABLE t_ma
ADD PRIMARY KEY (id);



- Für die Tabelle t_lager wird eine zusätzliche Gültigkeitsbedingung eingeführt. Die Eingabe für das Feld preis muss größer 0 sein

```
ALTER TABLE t_lager  
  ADD CONSTRAINT c_ppruef CHECK (preis > 0);
```

- Das Feld wert wird aus der Tabelle t_lager gelöscht

```
ALTER TABLE t_lager  
  DROP wert;
```



ALTER TABLE – 2

SYNTAX

ALTER TABLE tabellenname

[**ADD** feld typ [**DEFAULT** wert|**NULL**|**NOT NULL**] [**AUTO_INCREMENT**]]

[**ADD PRIMARY KEY** (datenfelder)]

[**ADD CONSTRAINT** name **CHECK** (bedingung)]

[**DROP** objektname] ;

Dabei gilt:

- Bei DROP kann sich der Objektname auf
 - ◆ ein Attribut,
 - ◆ einen Schlüssel (Objektname = **PRIMARY KEY**) oder
 - ◆ eine Gültigkeitsprüfung beziehen



ALTER TABLE – 3

- Beim Hinzufügen eines Datenfeldes erhalten die bereits vorhandenen Datensätze ein neues leeres Datenfeld, ggf. aufgefüllt mit dem Standardwert
- Beim Löschen eines Datenfeldes werden alle darin enthaltenen Werte gelöscht
- Das Definieren eines Datenfeldes, Primärschlüssels oder einer Gültigkeitsprüfung folgt der gleichen Syntax wie bei der Erstellung einer Tabelle
- Gültigkeitsprüfungen gelten nur für die nach der Änderung neu eingefügten Datensätze



DROP DATABASE

Löschen der kompletten DB

- **DROP DATABASE [IF EXISTS]** datenbankname;
- Mit der SQL-Anweisung **DROP** wird ein Datenbankobjekt gelöscht. Zum Löschen einer Datenbank wird **DATABASE** als Objekttyp angegeben
- Der Name der Datenbank folgt am Ende der Anweisung
- Die optionale Anweisung **IF EXISTS** verhindert das Auftreten von Fehlermeldungen, falls die Datenbank nicht existiert
- Es erfolgt kein Warnhinweis ö.ä.
- Zum Löschen sind die entsprechenden Rechte notwendig



DROP TABLE

Syntax – Löschen einer Tabelle

DROP TABLE tabellenname;

- Mit dieser Anweisung wird die angegebene Tabelle mit allen enthaltenen Daten gelöscht
- Sie müssen entweder Eigentümer der Tabelle sein oder die notwendigen Rechte zum Löschen der Tabelle besitzen
- Es dürfen keine Transaktionen aktiv sein, die die betreffende Tabelle verwenden
- Sofern noch andere Referenzen auf diese Tabelle bestehen (z.B. durch die Definition von virtuellen Tabellen, den Sichten), müssen diese vorher entfernt werden.



Sekundärschlüssel – 1

- Die Definition eines Sekundärschlüssels durch das Schlüsselwort **UNIQUE** vermeidet ebenfalls Redundanzen, da keine doppelten Werte auftreten dürfen

Beispiel:

```
CREATE TABLE t_artikel  
  (id INTEGER NOT NULL AUTO_INCREMENT ,  
   name VARCHAR(100) NOT NULL ,  
   code VARCHAR(30) NOT NULL ,  
   lieferant INTEGER ,  
   bemerkung VARCHAR ,  
   PRIMARY KEY (id) ,  
   UNIQUE (name, code) ) ;
```



Sekundärschlüssel – 2

- Das Datenfeld id soll als Primärschlüssel verwendet werden. Da die Definition unabhängig von der Auszeichnung als Primärschlüssel ist, erfolgt zusätzlich die Angabe **NOT NULL**
- Für jeden Artikel wird eine eindeutige Kombination aus Name und Code gespeichert (→ Schlüsselkandidat → Angabe **NOT NULL**)
- Mit der **UNIQUE** Klausel werden die Felder name und code (zusammen) zum Sekundärschlüssel erklärt



Sekundärschlüssel – 3

Syntax

```
CREATE TABLE tabellenname  
(datenfeld1 datentyp1 ... , ...  
UNIQUE [schlüsselname] (datenfeldliste) ) ;
```

- Optional kann bei einigen DBMS der Sekundärschlüssel einen Namen erhalten → sinnvoll bei Fehlermeldungen zu Sekundärschlüssel-Verletzungen

Nachträglich

```
ALTER TABLE name ADD UNIQUE [schlüsselname] (datenfeldliste) ;  
ALTER TABLE name DROP INDEX schlüsselname ;
```

- Ein Sekundärschlüssel kann mit **DROP INDEX** nur dann gelöscht werden, wenn er bei der Definition einen Namen bekommen hat
- Eine nachträgliche Definition eines Sekundärschlüssels ist nur dann möglich, wenn diese Datenfelder als **NOT NULL** deklariert wurden



Fremdschlüssel – 1

- Fremdschlüssel sind Attribute in einer Relation R_2 , die in einer anderen Tabelle R_1 Primärschlüssel sind
- Auf diese Weise werden Beziehungen oder Nachschlagetabellen realisiert
- Damit die **referenzielle Integrität** gewahrt bleibt, dürfen in R_1 keine Datensätze gelöscht werden, auf die in R_2 noch Verweise existieren
- Durch die Fremdschlüssel wird die referenzielle Integrität festgelegt und bei entsprechend fehlerhaften **INSERT**, **UPDATE** oder **DELETE** Anweisungen mit einer Fehlermeldung quittiert
- Grad der Unterstützung ist stark vom DBMS abhängig!



```
CREATE TABLE t_ma_proj  
  (ma_id INTEGER NOT NULL ,  
   proj_id INTEGER NOT NULL ,  
   FOREIGN KEY (ma_id) REFERENCES t_ma (id) ,  
   FOREIGN KEY (proj_id) REFERENCES t_proj (id) ) ;
```

- Diese Tabelle realisiert eine m:n Beziehung zwischen Mitarbeitern (t_ma, Primärschlüssel Attribut id) und den bearbeiteten Projekten (t_proj, Primärschlüssel Attribut id)
- Offensichtlich kann jede Tabelle mehrere Fremdschlüssel besitzen. Schwierig zu handhabende sog. **zirkuläre Referenzen** sind gekennzeichnet durch gegenseitige Fremdschlüsseldefinitionen zwischen zwei Tabellen



Syntax

```
CREATE TABLE tabellenname  
(datenfeld1 datentyp ..., ...  
FOREIGN KEY (feldliste) REFERENCES tabellenname (feldliste)  
  [ ON UPDATE option ] [ ON DELETE option ] ) ;
```

- Fremdschlüssel werden wie Primär- und Sekundärschlüssel am Ende der **CREATE TABLE** Anweisung festgelegt
- Die optionalen Referenzoptionen für **UPDATE** bzw. **DELETE** legen das Verhalten bei Verletzungen fest
 - ◆ **NO ACTION**
Die Anweisung wird abgebrochen
 - ◆ **SET NULL**
Alle referenzierten Datenfelder werden auf **NULL** gesetzt



- ◆ **SET DEFAULT**

Alle referenzierten Datenfelder werden auf den **DEFAULT**-Wert zurückgesetzt

- ◆ **CASCADE**

Beim Löschen werden alle Tupel in anderen Tabellen, die auf dieses Tupel verweisen, ebenfalls gelöscht

Beim Aktualisieren werden alle referenzierten Tupel in den entsprechenden Tabellen ebenfalls geändert

- Ohne angegebene Option wird **NO ACTION** implizit angenommen



Nachträglich

```
ALTER TABLE tabellenname ADD  
FOREIGN KEY (feldliste) REFERENCES tabellenname (feldliste)  
[ON UPDATE option] [ON DELETE option] ;
```

- Parameter entsprechend zur Definition von Fremdschlüsseln in der **CREATE TABLE** Anweisung
- In der Praxis werden häufig
 - ◆ Primär- und Sekundärschlüssel in der **CREATE TABLE** Anweisung und
 - ◆ Fremdschlüssel nachträglich mittels **ALTER TABLE** Anweisung festgelegt
 - ◆ Warum?



- Die Suche in großen Datenbeständen kann mit Indizes erheblich beschleunigt werden
- Liegt ein Index für eine Tabelle vor, durchsucht das DBMS die Tabelle bei Abfragen auf den Indexfeldern nicht mehr sequentiell, sondern nur den wesentlich kompakteren Index
- Indizes werden meist in gesonderten Dateien verwaltet – ebenfalls aus Performancegründen
- Indizes werden mit **CREATE INDEX** angelegt



CREATE INDEX i_mitarbeiter_name **ON** t_ma (name) ;

- Legt einen Index für die Nachnamen der Mitarbeiter an

- Dadurch werden Abfragen wie

SELECT * FROM t_ma **WHERE** name **LIKE** "M%" ;
wesentlich beschleunigt

- Im Gegensatz zu einem Sekundärschlüssel sind bei einem Index doppelte Werte zulässig
- i_ ist ein geeigneter Präfix für den Namen von Indexobjekten, der Name wird aber nur zum (späteren) Löschen benötigt
- Ein Index ist richtungsabhängig – Standard ist aufsteigend



Syntax

```
CREATE [UNIQUE] INDEX [indexname] ON  
tabellenname (datenfeld1 [ASC|DESC], ... , datenfeldn  
[ASC|DESC]);
```

- Durch die Angabe von **UNIQUE** wird ein Sekundär-schlüssel realisiert
- **ASC** (aufsteigend, Standardwert) bzw. **DESC** (absteigend) legen die Sortierorientierung des Index fest
- Mit **SHOW INDEX FROM** **tabellenname**;
können Sie alle definierten Indizes der aktuell verwendeten Tabelle anzeigen lassen



- Für die an Primär-, Sekundär- und Fremdschlüssel beteiligten Datenfelder werden automatisch Indizes erstellt
- Bei gering variierenden Datenwerten oder Tabellen mit wenigen Datensätzen sind Indizes wenig sinnvoll
- Indizes sind gut bei großen Datenmengen für Datenfelder (über) die häufig abgefragt wird
- Datenfelder die häufig auf- und absteigend sortiert werden, benötigen zwei Indizes



- Aggregatfunktionen (später ...) nutzen keinen Index, da z.B. zur Summenbildung alle Tupel durchlaufen werden müssen

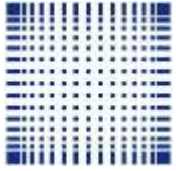
Index löschen

DROP INDEX indexname [**ON** tabellenname] ;

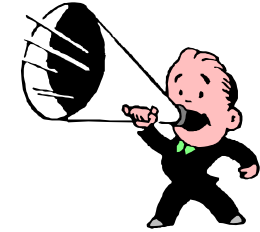
- Bei MySQL ist die Angabe des Tabellennamens nicht optional, sondern Pflicht
- Sind große Datenmengen aus einer Tabelle zu löschen oder in sie einzufügen, werden in der Praxis häufig die Indizes vorher gelöscht und danach wieder angelegt



- Es fehlen noch Konzepte
 - ◆ DDL: Sichten
 - ◆ DCL: Vergabe von Benutzerrechten etc.
- Hierzu müssen aber zuerst einige DML Konzepte vorher verstanden werden
- Folgen als Einschübe im DML Kapitel



JETZT

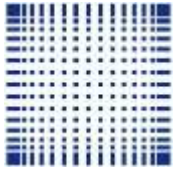


Abschnitt 8.2

SQL als DML



- Für das Einfügen von Datensätzen gibt es die **INSERT INTO** Anweisung
- Wird benutzt um einen ganzen Datensatz oder nur einzelne Attribute davon mit Werten zu belegen
- In der Realität werden die Eingaben eher über eine Konsole oder ein Anwendungsprogramm vorgenommen
- Sie müssen die Einfüge Rechte auf dieser Tabelle besitzen (oder deren Besitzer sein)
- Zeichenketten, Text- und Datumsangaben müssen in Anführungszeichen oder Apostrophe eingeschlossen werden. Bei Zahlen kann dies entfallen



```
CREATE TABLE t_ma_dt  
  (id INTEGER NOT NULL AUTO_INCREMENT,  
   name VARCHAR(50), vname VARCHAR(50), str VARCHAR(150),  
   plz VARCHAR(5), ort VARCHAR(50), alt INTEGER,  
   PRIMARY KEY(id) );
```

```
INSERT INTO t_ma_dt (name, vname, str, plz, ort, alt) VALUES  
  ("Teuber", "Klaus", "Berliner Str. 3", "04651", "Rochlitz", 23);
```

```
INSERT INTO t_ma_dt (name, vname, str, plz, ort) VALUES  
  ('Teuber', 'Klaus', 'Berliner Str. 3', '04651', 'Rochlitz');
```

```
INSERT INTO t_ma_dt (name, vname) VALUES  
  ("Schäfer", "Rosalie");
```



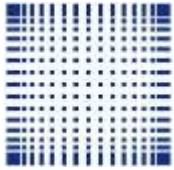
Syntax

INSERT INTO tabellenname (feld1, ... , feldX) **VALUES**
(wert1, ... , wertX);

- Nach dem Tabellennamen folgt in runden Klammern eine Liste der Attribute
- Hierdurch wird eine Reihenfolge für Eingabewerte in runden Klammern nach **VALUES** festgelegt
- Textinformationen etc. müssen in " oder ' eingeschlossen werden (Vorsicht, z.T. implementierungsabhängig)
- Sofern erlaubt, ist die direkte Zuweisung von **NULL** möglich



- Nicht alle Datenfelder müssen einen Wert erhalten. Die Auswirkung ist Abhängig von der Definition des Datenfeldes in der **CREATE TABLE** Anweisung
 - ◆ **DEFAULT** wert
der Standardwert wert wird gespeichert
 - ◆ **NULL**
Datenfeld bleibt leer, enthält **NULL**
 - ◆ **NOT NULL**
Fehlermeldung! Für dieses Feld muss ein Wert angegeben werden, es wird kein Tupel erzeugt
 - ◆ Keine Angabe
Datenfeld bleibt leer, enthält **NULL**



- Die zweite Form der **INSERT**-Anweisung erlaubt das Einfügen mehrerer Datensätze in einem Arbeitsschritt
- Die einzufügenden Daten werden dabei mit Hilfe einer Abfrage aus einer zweiten, ebenfalls vorher existierenden Tabelle gewonnen

Syntax:

```
INSERT INTO tabellenname1 (feld1, ... , feldX)  
      SELECT [*|datenfeldliste] FROM tabellenname2  
      [WHERE bedingung] ;
```

- Neu hinzugekommen ist anstelle der **VALUES**-Klausel eine gültige **SELECT**-Abfrage (genaueres hierzu – wie gesagt – später)
- Der Aufbau der zurück gelieferten Datensätze muss dabei mit den Angaben in der Datenfeldliste übereinstimmen
- Die Datenfeldliste kann entfallen – in diesem Fall muss die Anzahl der durch die Abfrage zurück gelieferten Datenfelder mit der der Zieltabelle übereinstimmen



- Müssen die Daten eines Datensatzes geändert werden, wäre eine Möglichkeit, den Datensatz zu löschen und neu einzufügen

Es geht aber einfacher:

- Mit der **UPDATE** Anweisung ist es möglich einen oder auch mehrere Datensätze gleichzeitig zu aktualisieren
- Hierzu müssen Sie entweder Besitzer der Tabelle sein oder mit den entsprechenden Rechten ausgestattet sein



```
UPDATE t_ma_dt SET alt = 45  
      WHERE name = "Backhaus" AND vname = "Erna";
```

- Das Alter von Erna Backhaus wird aktualisiert

```
UPDATE t_ma_frankfurt  
      SET ort="Frankfurt/Main";
```

- Alle Datensätze erhalten für das Feld ort den Wert "Frankfurt/Main"

```
UPDATE t_lager  
      SET stueck = stueck + 10  
      WHERE stueck < 10;
```

- Bei allen Datensätzen (Artikel im Lager) die weniger als 10 mal vorhanden sind, wird die Stückzahl um 10 erhöht (Einkauf)



SYNTAX

UPDATE tabellenname **SET** feld1=wert1, ..., feldX=wertX
[**WHERE** bedingung] ;

- Nach dem Schlüsselwort **SET** folgen eine oder mehrere Zuweisungen. Die Ausdrücke (Berechnungen möglich) müssen zum Datentyp des jeweiligen Attributs zuweisungskompatibel sein (ggf. durch einen **CAST**)
- Der Wert darf sogar eine Unterabfrage sein (später ...)
- Mit **WHERE** kann die Menge der zu aktualisierenden Datensätze eingeschränkt werden. Wird diese Klausel nicht angegeben, sind alle Datensätze von den Änderungen betroffen



Beispiele Wertzuweisungen

- Zuweisung einer Textinformation
SET bezeichner = "Gerätenummer"
- Speichern der Zahl 200 im Feld anzahl
SET anzahl = 200
- Der aktuelle Wert des Feldes Anzahl wird verdoppelt
SET anzahl = anzahl * 2
- Durch eine Unterabfrage (in einer anderen Tabelle) wird der Artikelname für einen Bezeichner ermittelt
SET bezeichner = (**SELECT** name **FROM** t_artikel **WHERE** id = 20)
- Unterabfragen müssen so angelegt werden, dass sie nur genau einen Wert als Ergebnis haben



- Mit der Anweisung **DELETE** können Sie einzelne oder mehrere Datensätze löschen
- Die Auswahl der betroffenen Datensätze kann analog zu den Möglichkeiten der **UPDATE** Anweisung eingegrenzt werden
- Die Löschung der Daten erfolgt ohne Sicherheitsabfrage oder Warnhinweis!
Vorsichtiger Umgang mit dieser Anweisung ist also dringend angeraten
- Sie müssen entweder Besitzer der Tabelle sein oder die entsprechenden Rechte besitzen



DELETE FROM t_lager **WHERE** stueck = 0;

- Alle Datensätze werden gelöscht, deren Wert im Feld stueck gleich 0 ist

DELETE FROM t_ma_dt **WHERE** ort = "Frankfurt";

- Es werden alle Datensätze aus der Tabelle t_ma_dt gelöscht, bei denen als Ort der Wert "Frankfurt" gespeichert ist

DELETE FROM t_ma_dt;

- Ohne Bedingung: ALLE Datensätze werden gelöscht



Syntax

DELETE FROM tabellenname [**WHERE** bedingung] ;

- Die Anweisung wird durch die Schlüsselworte **DELETE FROM** eingeleitet und gefolgt durch die Angabe des Namens der Tabelle in der gelöscht werden soll
- Um den Löschvorgang auf einen oder mehrere Datensätze einzugrenzen kann (analog zur **UPDATE** Anweisung) in einer **WHERE** Klausel eine bedingung formuliert werden
- Ohne Bedingung werden ohne Nachfrage alle Datensätze gelöscht. Als Resultat verbleibt die leere Tabelle



- Die gezielte Abfrage der gespeicherten Informationen ist die wahrscheinlich **schwierigste** Aufgabe beim Umgang mit Datenbanken
- ... aber auch die **häufigste** ...
- In SQL: **SELECT** Anweisung
 - ◆ viele optionale (aber leider auch viele proprietäre) Erweiterungen
 - ◆ ähneln Fragen oder Aufforderungen
 - ◆ nicht mit der Selektion zu verwechseln



***Formulieren einer Abfrage bedeutet:
Fragen bzw. "Umgangssprache"
in DML-Konstrukte transformieren!***

***Schreiben Sie diese Frage in normaler
Sprache immer als Kommentar zu
einer Abfrage zur Dokumentation!***



SELECT * FROM t_ma;

- Zeige alle Datensätze der Tabelle t_ma

SELECT vname, name **FROM** t_ma;

- Zeige Vorname und Name aller Datensätze der Tabelle t_ma an
- Projektion!
- Die **SELECT** Anweisung kann extrem komplex und durchaus auch sehr lang werden



- Die **SELECT** Anweisung liefert alle oder ausgewählte Datenfelder von Tabellen
- Die Daten müssen hierzu nicht nur einer Tabelle entstammen. Es sind auch verknüpfte Anfragen über zwei oder mehrere Tabellen möglich
- Weitere Optionen der **SELECT** Anweisung
 - ◆ Eingrenzen der Datensätze
 - ◆ Sortierung
 - ◆ Gruppierung
 - ◆ Auswertung



- Einfachstes Beispiel

SELECT * FROM tabellenname;

- Gesucht: Vorname (vanme), Name (name), Postleitzahl (plz) und Ort (ort) der Mitarbeiter (Tabelle t_ma), die im Postleitzahlbereich 6 wohnen und jünger als 40 (alt) sind. Die ausgegebenen Datensätze sollen aufsteigend nach der Postleitzahl sortiert ausgegeben werden

```
SELECT vname, name, plz, ort  
FROM t_ma  
WHERE plz LIKE '6%' AND alt < 40  
ORDER BY plz;
```



Syntax, einfache Anfragen

```
SELECT [DISTINCT] *|datenfelder  
FROM tabellenname  
[WHERE Bedingung]  
[[GROUP BY datenfelder [HAVING Bedingung]]  
[ORDER BY datenfelder [ASC|DESC]]  
[LIMIT [start ,] anzahl] ;
```

- Abfrage wird mit dem Schlüsselwort **SELECT** eingeleitet
- **DISTINCT** unterdrückt doppelte Datensätze
- ... die z.B. durch die Ausblendung von Spalten entstehen



- * steht für alle Datenfelder oder es wird eine (geordnete → Ausgabe) Liste der Datenfelder angegeben
- Nach dem Schlüsselwort **FROM** steht der Tabellename
- Die **WHERE** Klausel kann zum Einschränken der Datensätze benutzt werden
- Mit **ORDER BY** wird eine Sortierreihenfolge festgelegt



- Durch **GROUP BY**, gefolgt von der Angabe eines oder mehrerer Datenfelder, können Gruppierungen von Datensätzen vorgenommen werden
- Diese Gruppierungen können wiederum durch **HAVING** Klausel mit Bedingung wieder eingeschränkt werden
- Durch die Angabe von **LIMIT** kann die Anzahl der ausgegebenen Datensätze begrenzt werden und optional bei einem Startdatensatz beginnen



Spalten umbenennen

- Für eine benutzerfreundlichere Ausgabe
- oder für weitere Operationen
- können Spalten mit dem Schlüsselwort **AS** umbenannt werden

Beispiel:

```
SELECT name AS Familienname, vname AS Vorname, plz  
      AS Postleitzahl, alt AS "Alter"  
FROM t_ma;
```

WARUM?

SQL Schlüsselworte,
Leerzeichen



- In die Ergebnisdaten können konstante Werte in die Ausgabe einbezogen werden
- zur Unterscheidung von Feldnamen in " oder ' zu stellen

Beispiel:

```
SELECT "Mitarbeiter" AS angestellt_als, vname, name  
FROM t_ma_dt  
WHERE id < 4 ;
```

| angestellt_als | vname | name |
|----------------|--------|----------|
| Mitarbeiter | Hans | Schubert |
| Mitarbeiter | Klaus | Teubner |
| Mitarbeiter | Rosina | Schäfer |



- **SELECT** liefert standardmäßig alle Datensätze, auf die die Anfrage zutrifft
- mit **LIMIT** kann die Ausgabe begrenzt werden

Beispiele:

```
SELECT vname, name FROM t_ma  
WHERE plz = "38678" LIMIT 10 ;
```

- ◆ Begrenzt die Ausgabe auf die ersten 10 Tupel

```
SELECT vname, name FROM t_ma  
WHERE plz = "38678" LIMIT 30,10 ;
```

- ◆ Begrenzt die Ausgabe auf 10 Tupel ab dem 30. Tupel

- **LIMIT** ist nicht bei allen DBMS verfügbar



Doppelte Ausgaben vermeiden – 1

- In der Tabelle Mitarbeiter werden mehrere Mitarbeiter im gleichen Ort bei gleicher Postleitzahl wohnen
- Was liefert dann:
SELECT plz, ort **FROM** t_ma ;
da reine Projektion → doppelte Datensätze
- Ausgabe wirkt unübersichtlich, das Ergebnis ist wahrscheinlich sogar gar nicht erwünscht
- Ergebnis ist keine Menge



Doppelte Ausgaben vermeiden – 2

- Die Angabe von **DISTINCT** in der **SELECT** Klausel vermeidet die Ausgabe doppelter Werte bzw. Wertepaare

Beispiel:

SELECT DISTINCT plz, ort **FROM** t_ma ;

- ◆ liefert nur paarweise unterschiedliche plz, ort Wertekombinationen
- ◆ damit ein anderes Ergebnis
- ◆ Dieses Ergebnis stellt wieder eine Menge dar



- In einer **SELECT** Abfrage sind Berechnungen möglich

Beispiele:

```
SELECT id, preis, preis * 1.16 AS Verkaufspreis  
FROM t_lager ;
```

- ◆ Fügt der Ausgabe eine zusätzliche Spalte mit dem Verkaufspreis inkl. Umsatzsteuer hinzu

```
SELECT id, preis, preis * stueck AS Lagerwert  
FROM t_lager ;
```

- ◆ Berechnet den Lagerwert des jeweiligen Artikels und gibt ihn in einer gesonderten Spalte aus

- Berechnungen können nur mit numerischen Datenfeldern durchgeführt werden
- SQL erlaubt: + - * / (Punkt- vor Strichrechnung)
Klammerungen () zur Beeinflussung der Reihenfolge



- Die **WHERE** Klausel erlaubt die Einschränkung einer Ergebnismenge
- in der **SELECT** Abfrage
aber auch bei **UPDATE**, **DELETE**
- Wichtig bei Tabellen mit vielen (tausenden) Datensätzen
- Sehr schnelle Ausführung (wenn z.B. die Indizes geschickt gewählt wurden)
- Die **WHERE** Klausel gestattet eine große Vielfalt von Operationen zur Formulierung der Bedingung



■ Vergleichsoperationen

preis < 100

name = "Meier"

Vergleicht den Wert eines Datenfeldes mit einem Literal

■ Bereichsprüfung

preis **BETWEEN** 10.0 **AND** 100.0

Prüft, ob der Wert eines Attributs innerhalb eines Bereichs liegt

■ Elementprüfung

abteilung **IN** ("Einkauf", "Verkauf")

Prüft, ob der Wert eines Feldes in einer angegebenen Liste vorkommt



■ Mustervergleich

name **LIKE** "M%"

Prüft das Attribut auf Übereinstimmung mit einem Muster

■ Nullwertprüfung

preis **IS NULL**

Bedingung erfüllt, wenn dieses Datenfeld keinen Wert enthält

■ Logische Operatoren

preis < 100.0 **AND** preis > 10.0

name <> "Meier" **OR** vname <> "Andreas"

Verknüpfen von Bedingungen

■ Unterabfragen

preis < (**SELECT** bruttopreis **FROM** ...)



- In der **WHERE** Klausel sind nur Attribute und Literale erlaubt – Ersatznamen (Definition durch **AS**) sind zu diesem Zeitpunkt nicht bekannt und daher **nicht erlaubt**
- Liste der Vergleichsoperatoren für numerische Werte
< , > , <> , = , >= , <=
- Liste der Vergleichsoperatoren für Textwerte
<> , = , **LIKE**-Operator
(MySQL erlaubt zusätzlich < , > , etc.)
- Vorsicht: SQL unterscheidet in der Syntax Groß-/Kleinschreibung nicht – wohl aber bei Datenwerten
"Berlin" <> "berlin"



- Bereichsüberprüfung mit den Schlüsselworten **BETWEEN** und **AND**
- nicht nur für numerische Werte

Syntax:

SELECT ... FROM ...
WHERE datenfeld **BETWEEN** untergrenze
AND obergrenze ;

- Die Werte für die Unter- und Obergrenze sind im gültigen Wertebereich mit enthalten



```
SELECT * FROM t_lager  
WHERE preis >= 10.0 AND preis <= 100.0 ;
```

- ◆ ist eine gültige SQL Abfrage, kann (und soll) aber eleganter formuliert werden als

```
SELECT * FROM t_lager  
WHERE preis BETWEEN 10.0 AND 100.0 ;
```

```
SELECT * FROM t_ma  
WHERE name BETWEEN 'Be' AND 'Bo' ;
```

- ◆ Liefert alle Datensätze, bei denen der Nachname mit "Be" (einschließlich) bis "Bn" (einschließlich) beginnen
- ◆ "Bo" (keine weiteren Zeichen!) wird auch noch erkannt
- ◆ Alle Namen die mit "Bo" **beginnen** werden **nicht ausgegeben**



Die Anfrage

```
SELECT * FROM t_ma  
WHERE ort = 'Berlin' OR ort = 'Leipzig' OR  
      ort = 'Hamburg' ;
```

- ◆ ist gültig – kann (und soll) aber eleganter formuliert werden als:

```
SELECT * FROM t_ma  
WHERE ort IN ('Berlin', 'Leipzig', 'Hamburg') ;
```

- Die Bedingung wird durch die Verwendung des **IN**-Operators übersichtlicher – spätestens bei mehr als zwei Vergleichen



Syntax

SELECT ... FROM ...

WHERE datenfeld **IN** (wert1, ... , wertX) ;

- Vor dem Schlüsselwort **IN** steht das eine Attribut, mit dem verglichen wird
- Die Werteliste ist eine Kommata getrennte Liste von Literalen. Textwerte müssen in " oder ' eingeschlossen werden
- Allgemein: Bei Fließkommazahlen gelten im Bezug auf die Darstellungs- und Speichergenauigkeit die gleichen "Vorsichtsregeln" wie bei Programmiersprachen



- Die Directory Anweisung verschiedener Betriebssysteme erlaubt die Angabe von sog. Wildcards bei der Ausgabe von Dateinamen
 - dir *.txt
 - für alle Dateien mit dem Suffix .txt
- Das geht in SQL auch 😊
- In Verbindung mit dem **LIKE**-Operator sind folgende Platzhalterzeichen (Wildcards) erlaubt
 - ◆ Prozentzeichen %
steht für kein, ein oder mehrere beliebige Zeichen
 - ◆ Unterstrich, Underscore _
steht für exakt ein beliebiges Zeichen



- name **LIKE** "F%"
Funke, Franz, Fahrmann
- name **LIKE** "%son"
Benson, Jenson, Morrison
- name **LIKE** "%ill%"
Miller, Filler, Ofillson
- name **LIKE** "M_ller"
Müller, Möller, Miller
- name **LIKE** "_____"
Adas, Funk, Tier
- name **LIKE** "M%er_"
Mopfert, Mehnert, Meinert, Maierl

(Anm.: sind 4 Unterstriche)



Syntax

SELECT ... FROM ...

WHERE datenfeld **LIKE** "muster" [**ESCAPE** "Zeichen "];

- Was tun, wenn auf Gleichheit mit einem Platzhalterzeichen getestet werden soll? Es gibt kein "Standard"-Escapezeichen wie in C, C++ oder Java
- Dafür gibt es den optionalen ESCAPE Parameter. Angegeben wird das Zeichen plus ein Leerzeichen
... **WHERE** name **LIKE** 'Abt\ _%' **ESCAPE** \ ' ;
trifft z.B. zu für den Wert 'Abt_Organisation'



Logische Operatoren

- In einer **WHERE** Klausel kann die Bedingung als Verknüpfung mehrerer Bedingungen durch **logische** Operatoren formuliert werden
- Zulässig sind
 - ◆ **AND** (UND-Verknüpfung, binär, beide Bedingungen müssen erfüllt sein)
 - ◆ **OR** (ODER-Verknüpfung, binär, mindestens eine Bedingung muss erfüllt sein)
 - ◆ **NOT** (Nicht-Operator; unär; die Bedingung wird negiert)

Beispiele:

```
... WHERE name LIKE 'M%' AND ort <> 'Krefeld' ;  
... WHERE name LIKE 'M%' AND (ort = 'Berlin' OR ort = 'Leipzig')  
    AND alt < 50 ;  
... WHERE NOT (name LIKE 'M%') ;
```




Gruppierungen – 1

- **GROUP BY** Klausel gestattet Datengruppierungen nach einem festzulegenden (Wert-) Kriterium

Beispiel:

```
SELECT ort AS wohnort, COUNT(name)  
FROM t_ma GROUP BY ort ;
```

Datensätze zählen
(Aggregatfunktion,
später ...)

Tippfehler ???

| Wohnort | COUNT(name) |
|-------------|-------------|
| Bad Lausick | 7 |
| Belin | 1 |
| Berlin | 12 |
| Ernsee | 1 |
| Gera | 8 |
| Hamburg | 10 |



Gruppierungen – 2

- Mit dem optionalen **HAVING** kann für die Gruppierung zusätzlich eine Bedingung angegeben werden

Beispiel:

von eben – ABER: Es interessieren nur die Orte, in denen 10 oder mehr Mitarbeiter wohnen

```
SELECT ort AS wohnort, COUNT(name) FROM t_ma  
GROUP BY ort HAVING COUNT(name) >= 10 ;
```

| Wohnort | COUNT(name) |
|---------|-------------|
| Berlin | 12 |
| Hamburg | 10 |



Syntax

```
SELECT ... FROM ... [WHERE ...]  
GROUP BY feld1[, feld2, ..., feldX] [HAVING bedingung]
```

- **GROUP BY** ist besonders dazu geeignet die Ausgabe von Datenfeldern und Aggregatfunktionen zu kombinieren (was ohne **GROUP BY** zu einem Fehler führen würde)
- Die Gruppierung erfolgt nach dem ersten angegebenen Datenfeld, innerhalb der ersten Gruppierung nach dem zweiten angegebenen Datenfeld etc.
- Gruppierungen werden in der Literatur auch als Verdichtung oder Datenverdichtung bezeichnet
- **HAVING** ist nur in Verbindung mit **GROUP BY** zulässig



Syntax

```
SELECT ... FROM ... [WHERE ...] [GROUP BY ...]  
ORDER BY feld1 [,feld2, ..., feldX] [ASC | DESC] ;
```

- Das Abfrageergebnis wird nach dem ersten Datenfeld sortiert, bei Gleichheit des ersten Datenfeldes nach einem evtl. angegebenen zweiten Datenfeld etc.
- Standardmäßig wird aufsteigend sortiert (optionale aber überflüssige Angabe **ASC**), für eine absteigende Sortierung ist **DESC** anzugeben
- Eine Sortierung nach Aggregatfunktionen ist nur bei einigen DBMS (z.B. MySQL) möglich



SELECT vname, name, plz, ort **FROM** t_ma **ORDER BY** name, vname ;

- ◆ Es wird nach dem Nachnamen sortiert, bei gleichem Nachnamen nach dem Vornamen

SELECT * **FROM** t_lager **ORDER BY** preis **DESC** ;

- ◆ Die Tabelle t_lager wird absteigend nach dem Preis ausgegeben

SELECT ort **AS** Wohnort, **COUNT**(name) **FROM** t_ma
GROUP BY ort **ORDER BY** ort;

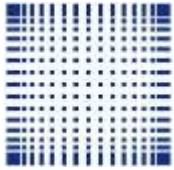
- ◆ Das **GROUP BY** Beispiel wird aufsteigend sortiert nach dem Ortsnamen ausgegeben

- Gemischte Sortierreihenfolgen sind in Standard-SQL nicht möglich



Aggregatfunktionen – 1

- In vielen Fällen werden nicht einzelne Datensätze benötigt, sondern Berechnungen oder Auswertungen über alle (oder eine eingeschränkte Gruppe von) Datensätzen gefordert
- In einigen DBMS werden Aggregatfunktionen nur schleppend langsam ausgeführt, weil jeder Datensatz einzeln angefasst werden muss



SELECT COUNT(*) FROM tabellenname ;

- ◆ Liefert die Anzahl der Datensätze einer Tabelle

SELECT COUNT(DISTINCT ort) FROM t_ma ;

- ◆ Liefert die Anzahl unterschiedlicher Datenwerte eines Attributes – hier die Anzahl der Wohnorte der Mitarbeiter

SELECT MAX(preis) AS Maximum, MIN(preis) AS Minimum, AVG(preis) AS ImMittel FROM t_lager ;

- ◆ Liefert den maximalen, den minimalen und den durchschnittlichen Preis aller Artikel in der Lagerdatenbank
- ◆ Zur besseren Lesbarkeit wurden die Spalten umbenannt

SELECT MAX(preis) AS Maximum, MIN(preis) AS Minimum, AVG(preis) AS ImMittel FROM t_lager
WHERE stueck >= 100;

- ◆ dito, aber nur über die Artikel aggregiert, die mindestens 100 mal auf Lager sind



- **COUNT(*)**
Liefert die Anzahl der Datensätze
- **COUNT(datenfeld)**
Liefert die Anzahl der Werte außer **NULL** Werten in der Ergebnismenge einer **SELECT** Abfrage oder Gruppierung
- **COUNT(DISTINCT * | datenfeld)**
wie (jeweils) oben, aber es werden nur unterschiedliche Werte gezählt
- **AVG(datenfeld)**
Liefert den Durchschnittswert eines Datenfeldes der Abfrage oder Gruppierung



- **MIN**(datenfeld), **MAX**(datenfeld)
Liefert den kleinsten bzw. größten Wert eines Datenfeldes der Abfrage oder Gruppierung
- **SUM**(datenfeld)
Liefert die Summe der Werte eines Datenfeldes in der Abfrage oder Gruppierung
- In Abfragen können Datenfelder und Aggregatfunktionen nicht gemischt werden

Fehler:

```
SELECT ort, COUNT(name) FROM t_ma ;
```



Aggregatfunktionen – 4

- Aggregatfunktionen können nicht in einer **WHERE** Klausel wohl aber in einer **HAVING** Klausel eingesetzt werden:

```
SELECT proj_id COUNT (ma_id) FROM t_ma_proj  
GROUP BY proj_id HAVING COUNT(ma_id)>2;
```

- Listet die Projekte auf, an denen mehr als zwei Mitarbeiter beteiligt sind
- Vorsicht: Lange Antwortzeiten
- Verschachtelungen von Aggregatfunktionen (z.B. die durchschnittliche Anzahl von Mitarbeitern an einem Projekt) sind so nicht möglich



Skalarfunktionen – 1

- Skalarfunktionen bearbeiten einen einzelnen Wert
SELECT vname, name **FROM** t_ma
WHERE UPPER(ort) = 'BERLIN' ;
- Liefert alle Vor- und Nachnamen der Mitarbeiter aus Berlin, ungeachtet von Groß-/Kleinschreibung des Wohnortes im Datensatz
SELECT SQRT(16) ;
Gibt die Quadratwurzel von 16 = 4.000000 aus

Auswahl von Skalarfunktionen (implementierungsabhängig);

- **ABS**(zahl) – Absolutwert der Zahl
- **CEILING**(zahl) – Ermittelt die kleinste Ganzzahl, die nicht kleiner als zahl ist



- **MOD**(zahl, teiler) – Ermittelt den Divisionsrest
 - **RAND**() – Liefert eine Zufallszahl
 - **ROUND**(zahl [, stellen]) – Rundet die Zahl
 - **SIGN**(zahl) – Ermittelt das Vorzeichen der Zahl
 - **SQRT**(zahl) – Berechnet die Quadratwurzel der Zahl
 - **LENGTH**(string) – ermittelt die Länge des String
 - **LOWER**(string) – wandelt den String in Kleinbuchstaben
 - **UPPER**(string) – wandelt den String in Großbuchstaben
-
- Die Syntax und Anwendung weiterer Skalarfunktionen ist im Handbuch bzw. der Online-Hilfe des jeweiligen DBMS nachzulesen



Tabellen verknüpfen – 1

- In der Praxis werden für Abfragen häufig die Daten aus mehreren Tabellen gleichzeitig, verknüpft benötigt

```
SELECT t_ma.vname, t_ma.name, t_ma.abtnr, t_abt.*  
FROM t_ma, t_abt  
WHERE t_ma.abtnr = t_abt.id ;
```

- In der **FROM** Klausel werden die beteiligten Tabellen angegeben
- Die **WHERE** Klausel enthält das (bzw. die) Prädikate zur Verknüpfung der Tabellen (hier ein Equi-Join) und ggf. weitere Bedingungen zur Einschränkung der Datensätze
- Die Attribute werden den Tabellen durch Punkt-Notation zugeordnet

| vname | name | abtnr | id | name | ort |
|-------|---------|-------|-----|---------|---------|
| Frank | Grinter | 1 | 1 | Einkauf | Leipzig |
| Laura | Kärner | 2 | 2 | Verkauf | Berlin |
| ... | ... | ... | ... | ... | ... |



Tabellen verknüpfen – 2

- Beim Verknüpfen von Tabellen ist die Angabe der Tabellennamen an vielen Stellen notwendig. Um die Abfragen kürzer zu halten, können Ersatznamen vergeben werden

```
SELECT m.vname, m.name, m.abtnr, a.*  
FROM t_ma AS m, t_abt AS a  
WHERE m.abtnr = a.id ;
```

- Einige SQL Dialekte definieren Ersatznamen ohne **AS**
... **FROM** t_ma m, t_abt a ...

- Alternativ kann der Verbund mehrerer Tabellen innerhalb einer **SELECT** Abfrage mit dem Schlüsselwort **JOIN** erreicht werden



Cross-, Full-Join, Kartesisches Produkt

SELECT datenfelder

FROM tabelle1 **CROSS JOIN** tabelle2 ... ;

- Jeder Datensatz aus Tabelle1 wird mit jedem Datensatz aus Tabelle2 verbunden (Kartesisches Produkt)

- Beispiel:

SELECT t_ma.*, t_proj.*

FROM t_ma **CROSS JOIN** t_proj

WHERE t_ma.ort = 'Berlin' ;

- Beantwortet die Frage: Welche Mitarbeiter aus Berlin könnten an welchen Projekten mitarbeiten?



Inner-, Equi-Join

```
SELECT feldliste FROM tabelle1 INNER JOIN tabelle2  
ON tabelle1.feld = tabelle2.feld [WHERE ... [ORDER BY ...]] ;
```

- Verknüpfung der Tabellen erfolgt über die Äquivalenz der Datenfelder

- Das Beispiel vom Anfang des Abschnitts, diesmal mit INNER JOIN

```
SELECT t_ma.vname, t_ma.name, t_ma.abtnr, t_abt.*  
FROM t_ma INNER JOIN t_abt  
ON t_ma.abtnr = t_abt.id ;
```

- Es können mehr als zwei Tabellen verknüpft werden (in der Praxis sind Joins über mehr als 10 Tabellen keine Seltenheit). Hierbei wird das erste Verknüpfungsergebnis mit der dritten Tabellen verknüpft und so fort – Syntax

```
SELECT feldliste FROM ((tabelle1 INNER JOIN tabelle 2 ON  
bedingung) INNER JOIN tabelle3 ON bedingung) INNER JOIN ...  
[WHERE ... [ORDER BY ...]] ;
```

- Klammerungen in **FROM** Klausel unnötig, erhöhen hier die Lesbarkeit



Natural-Join

- Bei einem Inner-Join können doppelte Datensätze auftreten
- Ein Natural-Join ist ein Inner-Join, mit paarweise unterschiedlichen Datensätzen. In SQL2 gibt es kein explizites Schlüsselwort hierfür
- Kann aber mit **SELECT DISTINCT ...** erreicht werden

Theta-Join

- Ist ein Inner-Join, bei dem im Unterschied zum Equi-Join in der **WHERE** Klausel bei der Verknüpfung die Operatoren $<$ $>$ \leq \geq (statt $=$) verwendet werden



Outer-Join

- Beim Inner-Join wird über die Übereinstimmung von Datenfeldern verknüpft
- Will man aber eine Tabelle um eine andere erweitern ist ein Outer-Join angebracht
- Wird kein passender Datensatz in der zweiten Tabelle gefunden werden die entsprechenden Attribute mit **NULL** Werten belegt
- Ein **LEFT OUTER JOIN** erweitert die linke Tabelle, ein **RIGHT OUTER JOIN** erweitert die rechte, ein **FULL OUTER JOIN** erweitert beide beteiligten Tabellen

Syntax

SELECT feldliste **FROM** tab1 **LEFT|RIGHT|FULL OUTER JOIN** tab2
ON bedingung ... ;

- tab1 wäre die linke Tabelle (LEFT), tab2 die rechte Tabelle (RIGHT)



Self-Join

- Es müssen nicht immer zwei unterschiedliche Tabellen miteinander verknüpft werden
- Eine Tabelle kann mit sich selbst verknüpft werden – die Vergabe von Ersatznamen ist in diesem Fall zwingend

- Beispiel:

```
SELECT a1.name, a2.name, a1.ort  
FROM t_abt AS a1 INNER JOIN t_abt AS a2  
ON a1.ort = a2.ort  
WHERE a1.id <> a2.id ;
```

Listet alle Abteilungen an deren Standort sich weitere Abteilungen befinden

- Die **WHERE** Klausel dient der Unterdrückung identischer Datensätze



Vereinigungs-, Differenz-, Schnittmengen

- Die Mitarbeiter (Tabelle t_ma) aus Leipzig und die aus Frankfurt (gespeichert in der Extra-Tabelle t_ma_frankfurt) sollen vereinigt ausgegeben werden :
SELECT name, ort **FROM** t_ma **WHERE** ort = 'Leipzig'
UNION
SELECT name, ort **FROM** t_ma_frankfurt ;
- Anders als beim Join werden ohne weitere Bedingung, Verknüpfung zwei Datensatzmengen vereinigt (**UNION**), Differenz- (**MINUS**) bzw. Schnittmengen (**INTERSECT**) gebildet
- Die Attribute müssen exakt die gleichen Namen haben – ggf. durch vorgehende Spaltenumbenennung
- Duplikate werden vorher eliminiert. Durch die Angabe von **UNION ALL** bzw. **INTERSECT ALL** ist in einigen SQL Dialekten die Aufnahme von Duplikaten in das Ergebnis möglich



Rückblick: Relationenalgebra

- SQL implementiert die Relationenalgebra und Relationenkalkül

SELECT A1, A2, ... , An
FROM R1, R2, ... , Rm
WHERE P ;

- Das entspricht:

$$\pi_{A1, A2, \dots, An}(\sigma_P(R1 \times R2 \times \dots \times Rm))$$

- Betrachten wir jetzt die Implementierung des Tupel- und Domänenkalküls



- **Unterabfragen (Subqueries)** sind verschachtelte Anfragen, die Verschachtelungstiefe ist ggf. vom DBMS begrenzt
- Unterabfragen können gebildet werden mit
 - ◆ Mengenoperationen (**IN, NOT IN**)
 - ◆ Quantoren (**ANY, ALL, EXISTS, NOT EXISTS**)
 - ◆ und mit Aggregatfunktionen

Mengenoperationen

```
SELECT name FROM t_ma WHERE abtnr IN  
  (SELECT id FROM t_abt WHERE name LIKE 'E%') ;
```

- Funktionsweise ist analog zur IN Klausel mit einer Menge von Literalen



ANY, ALL

```
SELECT vname, name FROM t_ma  
WHERE alt > ANY (SELECT alt FROM t_ma  
                WHERE ort = 'Frankfurt') ;
```

- ◆ Finde alle Mitarbeiter, die älter sind als mindestens einer der Mitarbeiter aus Frankfurt

```
SELECT vname, name FROM t_ma  
WHERE alt > ALL (SELECT alt FROM t_ma  
                WHERE ort = 'Frankfurt');
```

- ◆ Finde alle Mitarbeiter, die älter sind als alle (jeder) Mitarbeiter aus Frankfurt



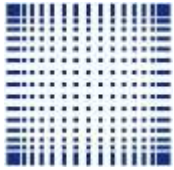
EXISTS, NOT EXISTS

- Die **EXISTS** Klausel testet das Ergebnis einer **SELECT** Abfrage und gibt **TRUE** zurück, wenn die Ergebnisrelation ungleich der leeren Menge ist

Beispiel:

```
SELECT vname, name FROM t_ma  
WHERE NOT EXISTS (SELECT * FROM t_ma_proj  
                   WHERE t_ma.id = ma_id) ;
```

- ◆ Ermittelt alle Mitarbeiter, die an keinem Projekt mitarbeiten
- ◆ t_ma.id ist ein Bezug auf die äußere Query!



Aggregatfunktionen

```
SELECT vname, name FROM t_ma  
WHERE alt > (SELECT AVG(alt) FROM t_ma) ;
```

- ◆ Listet alle Mitarbeiter auf, die älter als das Durchschnittsalter aller Mitarbeiter sind
- ◆ Die eingeschachtelte Abfrage darf nur einen Ergebniswert zurückliefern

```
SELECT name FROM t_lager  
WHERE preis < (SELECT AVG(preis) FROM t_lager)  
AND stueck < (SELECT AVG(stueck) FROM t_lager) ;
```

- ◆ Listet alle Artikel deren Preis niedriger als der Durchschnittspreis und die Stückzahl niedriger als die Durchschnittsstückzahl aller Artikel ist
- Subqueries mit IN können von innen nach außen entwickelt werden. Alle anderen Typen erfordern bei der Auswertung



- Allgemein gilt für Unterabfragen: Geschachtelte Abfragen sind vollwertige **SELECT** Abfragen mit allen bisher behandelten Syntaxmöglichkeiten und können natürlich unterschiedliche Tabellen abfragen
- Unterabfragen mit **IN** können von innen nach außen entwickelt werden
- Unterabfragen mit Quantoren bzw. Aggregatfunktionen erfordern innen und außen einen jeweiligen Vergleich Datensatz für Datensatz
- Sie heißen daher **korellierte Unterabfragen (corellated subqueries)**



Subquery – 7

- ☹️ *Aggregatfunktionen können nicht verschachtelt werden*
- 😊 Geht doch, entweder
 - ◆ mit Hilfe von Sichten oder
 - ◆ einige SQL-Dialekte erlauben Unterabfragen in der **FROM** Klausel. Dadurch ist es möglich aus einem Zwischenergebnis weitere Auswertungen zu erhalten

```
SELECT AVG(ProWohnort)
FROM
  (SELECT COUNT(ort) AS ProWohnort FROM t_ma
   GROUP BY ort) ;
```

Funktioniert
das?

- ◆ Ermittelt die durchschnittlichen Mitarbeiter pro Wohnort



DDL-Einschub: Sichten – 1

- Sichten sind virtuelle (also so nicht in der DB gespeicherte) Tabellen, die als Ergebnis aus
 - ◆ einer Abfrage und/oder
 - ◆ einer Verknüpfung von Tabellen entstehen
- Dabei liefern sie aber den jeweils aktuellen Datenstand
- Sichten können in SQL wie "echte" Tabellen (weiter-)verwendet werden
- Unter bestimmten Bedingungen ist auch das Einfügen, Ändern und Löschen von Datensätzen möglich
- Die Unterstützung von Sichten ist stark implementierungsabhängig (vgl. Codd's Regeln)



CREATE VIEW v_ma_abt1 AS

SELECT abtnr, vname, name **FROM** t_ma **WHERE** abtnr = 1 ;

- ◆ Definiert eine Sicht auf die Mitarbeiter aus Abteilung 1

SELECT * FROM v_ma_abt1 ;

CREATE VIEW v_ma_proj (MNr, MName, PNr, PNAme) AS

SELECT m.id, m.name, p.id, p.name **FROM** t_ma_proj mp

INNER JOIN t_ma m **ON** mp.ma_id = m.id

INNER JOIN t_proj p **ON** mp.proj_id = p.id ;

- ◆ Welche Mitarbeiter sind an welchen Projekten beteiligt?
Hierbei interessieren nur bestimmte Attribute, die gleich (für eine bessere Lesbarkeit) umbenannt werden

SELECT * FROM v_ma_proj **WHERE** PNr = 1 ;

- ◆ Welche Mitarbeiter arbeiten an Projekt 1? Ohne Sicht wäre die Formulierung dieser Abfrage sehr aufwändig



Syntax

CREATE VIEW viewname [(feldliste)] **AS**
SELECT abfrage ;

- Die **SELECT** Abfrage darf eine **WHERE** Klausel enthalten
- Die **SELECT** Abfrage darf nicht **GROUP BY**, **HAVING**, **ORDER BY** oder **UNION**, **INTERSECT**, **MINUS** Konstrukte enthalten

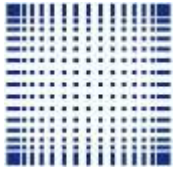
Löschen

DROP VIEW viewname ;

- Wird diese Sicht gerade verwendet (z.B. von anderen SQL Anweisungen), kann sie nicht gelöscht werden
- Falls Sichten existieren, die auf der Sicht viewname basieren, werden diese ebenfalls gelöscht



- In einer Sicht, die lediglich auf Projektionen beruht, können problemlos Daten modifiziert werden
- Es müssen sich allerdings für Einfüge Operationen alle Felder in der Sicht befinden, die in der Basistabelle nicht den **NULL** Wert annehmen dürfen
- Die meisten DBMS lehnen alle Datenmodifikationen für Sichten ab, die aus mehr als einer Tabelle bestehen
- Liegt der Sicht eine Abfrage mit **WHERE** Klausel zugrunde, werden Operationen undurchsichtiger:
- Werden z.B. Daten in die Sicht eingefügt, die die Sicht definierende **WHERE** Klausel nicht erfüllen, kann der Erfolg der Einfüge Operation nicht durch die Anzeige aller Datensätze der Sicht kontrolliert werden



Abhilfe – Syntax

CREATE VIEW viewname **AS SELECT** abfrage
WITH CHECK OPTION ;

Beispiel:

CREATE VIEW v_ma_abt1 **AS**
SELECT * FROM t_ma **WHERE** abtnr = 1
WITH CHECK OPTION ;

INSERT INTO v_ma_abt1 (name, vname, abtnr)
VALUES ('Funke', 'Franziska', 2) ;

verstößt gegen die **WHERE** Klausel
→ **Fehlermeldung, INSERT wird nicht ausgeführt**



DCL-Einschub: Rechte – 1

- Die DCL ist zwar von Anfang an Bestandteil von SQL – aber so gut wie gar nicht genormt
 - ◆ → Befehle (z.B. **GRANT**), Syntax sind stark implementierungsabhängig
 - ◆ → Studium von Online-Hilfe, Handbuch
 - ◆ Wir betrachten hier MySQL
- Rechteverwaltung ist zwingend notwendig
 - ◆ Daten vor unbefugtem Zugriff schützen
 - ◆ Daten vor versehentlicher Modifikation schützen



- Benutzer werden von MySQL in einer Tabelle verwaltet
- Benutzer hinzufügen
USE mysql ;
INSERT INTO user (host, user, password) **VALUES**
("localhost", "dbb1", **PASSWORD**("geheim")) ;
FLUSH PRIVILEGES ;
- Es wird in der Tabelle user, in der Systemdatenbank mysql ein neuer Datensatz = ein neuer Benutzer auf dem Rechner localhost (= 127.0.0.1; hier ist der Netzwerkname des Computers anzugeben) mit Account dbb1 und dem Passwort geheim angelegt
- "%" als Host steht für alle Computer im Netzwerk
- "%.fh-mannheim.de" steht für alle Knoten in der fh-mannheim.de Domain



Syntax

GRANT rechteliste **ON** objekt **TO** benutzer ;

Rechteliste (Auswahl)

| | |
|-------------------|-----------------------------------------------------------------|
| ALL | – gewährt alle Rechte |
| SELECT | – Leserecht, Recht zur Ausführung von Abfragen |
| UPDATE | – Änderungsrecht, UPDATE Anweisung |
| INSERT | – Einfügerecht, INSERT Anweisung |
| DELETE | – Löschrecht, DELETE Anweisung |
| REFERENCES | – Recht zur Definition von Regeln der referenziellen Integrität |

- Mehrere Rechte dürfen durch Kommata getrennt angegeben werden
- Andere DBMS haben ggf. andere Rechte-Stufen
- Als Objekt ist der Name von Datenbanken, Tabellen, Sichten nennbar



- Bei den Rechten **SELECT**, **INSERT**, **UPDATE** können die Rechte auf einzelne Datenfelder eingeschränkt werden:
GRANT UPDATE (stueck) **ON** t_lager **TO** pfunke ;

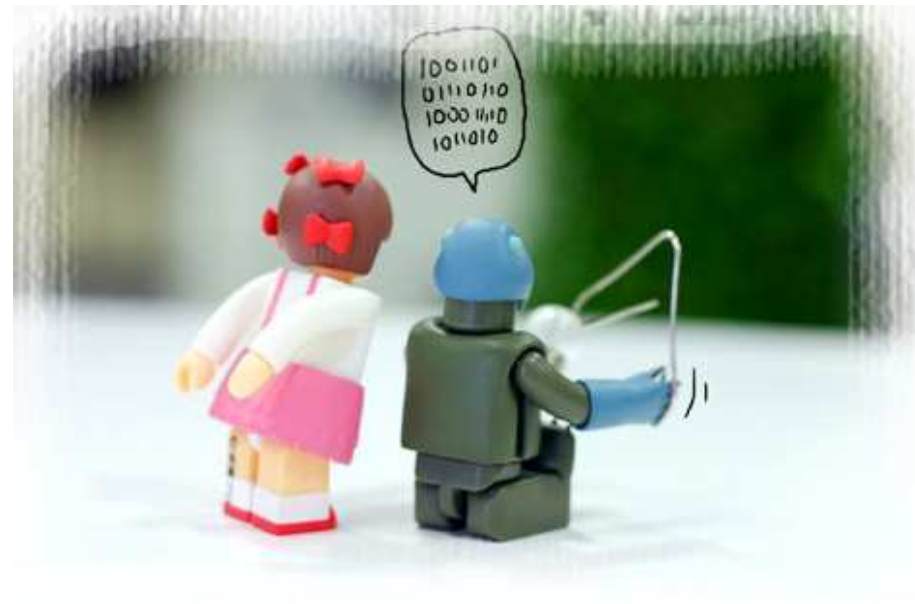
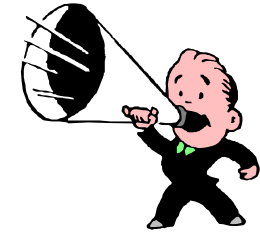
Entzug von Rechten – Syntax

REVOKE rechteliste **ON** objekt **FROM** benutzer ;

- Parameter analog zur **GRANT** Anweisung
- Löschen eines Benutzers erfolgt durch Löschen des entsprechenden Datensatzes aus der Tabelle user in der Systemdatenbank mysql



JETZT



Kapitel 9

Die Theorie hinter der Abfragesprache

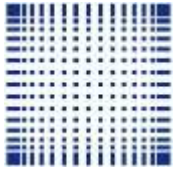
DM
3IB/3IMB/3UIB

325

WS 2017/18



- Sprachen, die mit Relationalen DB arbeiten, müssen in der Lage sein, folgende Operationen auszuführen:
 - ◆ Anlegen von neuen Relationen
 - ◆ Verändern von Relationen
 - ◆ Löschen von Relationen
 - ◆ Erzeugen von Relationen aus vorhandenen Relationen mit ausgewählten Tupeln und ausgewählten Attributen
- Das **Retrieval** (Auffinden gewünschter Daten) bildet den (schwierigeren) Kern der Sprache
- Relationale Datenbanksprachen (wie SQL) setzen sich aus zwei Sprachparadigmen zusammen:
 - ◆ Relationenalgebra
 - ◆ Relationenkalkül



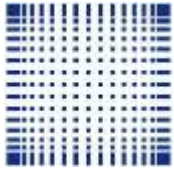
Beispiel – 1

Relation MITARBEITER1

| Personal Nr | Name | Vorname | Abteilung | Anschrift |
|-------------|------------|-----------|-----------|---------------------------------|
| 0001 | Lorenz | Sophia | 1 | 03725 Mausbach Mühlenweg 4 |
| 0002 | Hohl | Tatjana | 2 | 49262 Mausloch Käsereistr. 5 |
| 0003 | Willschrei | Theodor | 2 | 03453 Katzbergen Ahornweg 4 |
| 0004 | Richter | Hans | 3 | 78943 Katzenhausen Buchenweg 28 |
| 0005 | Wiesenland | Brunhilde | 3 | 02518 Hundsberg Mopsstr. 45 |

Relation MITARBEITER2

| Personal Nr | Name | Vorname | Abteilung | Anschrift |
|-------------|-----------|---------|-----------|----------------------------------|
| 0011 | Schneider | Jakob | 1 | 06238 Frankenhausen Bussardweg 7 |
| 0012 | Putzmann | Gertrud | 3 | 45345 Weidenau An der Alm 2 |



Beispiel – 2

Relation PROJEKT

| ProjektNr | Beschreibung |
|-----------|-------------------|
| 1 | Kundenumfrage |
| 2 | Verkaufsmesse |
| 3 | Konkurrenzanalyse |

Relation PROJEKT2

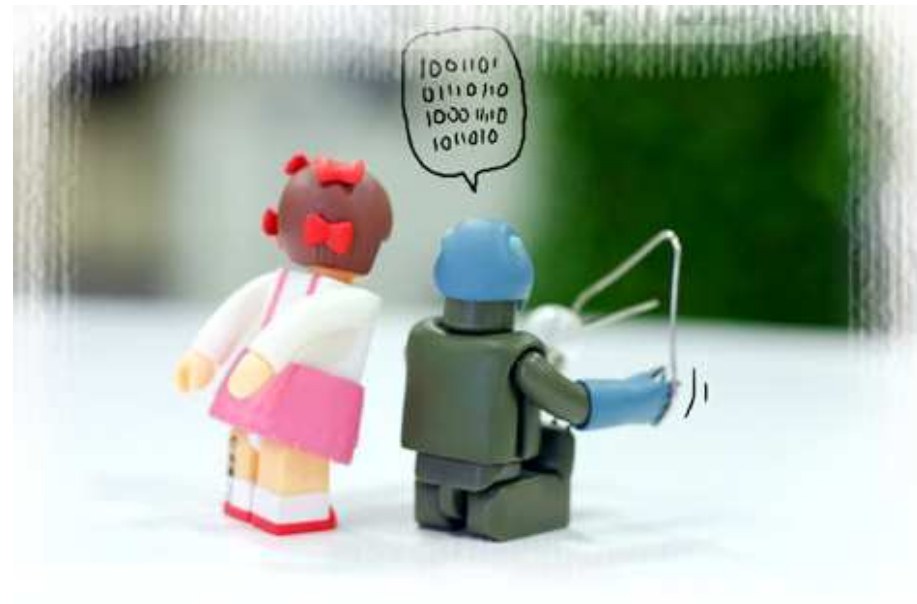
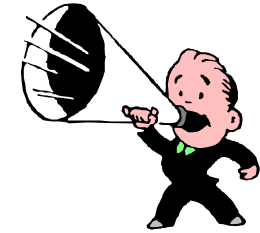
| ProjektNr | Beschreibung |
|-----------|----------------------------|
| 1 | Kundenumfrage |
| 4 | Wirtschaftlichkeitsanalyse |

Relation ARBEITET_AN

| Personal Nr | Projekt Nr | Tätigkeit | Stunden |
|-------------|------------|--------------|---------|
| 0002 | 1 | Leitung | 25 |
| 0003 | 1 | Bearbeitung | 55 |
| 0004 | 1 | Bearbeitung | 70 |
| 0004 | 2 | Leitung | 25 |
| 0005 | 2 | Präsentation | 160 |
| 0004 | 3 | Leitung | 25 |
| 0002 | 3 | Bearbeitung | 80 |
| 0003 | 3 | Bearbeitung | 65 |



JETZT



Relationenalgebra

DM
3IB/3IMB/3UIB

329

WS 2017/18



- **Algebra** ist ein System, welches aus nicht leeren Mengen und Operationen auf diesen Mengen besteht
 - ◆ nicht leere Menge = Relationen
 - ◆ Operationen = z.B. Datenbankabfragen
 - ◆ Basisrelationen = vorhandene Relationen
 - ◆ Ergebnisrelationen = Ergebnisse (Relationen), die durch Anwendung der Operationen entstehen, aber nicht in der DB gespeichert werden

- Im Folgenden Betrachtung der Operationen, die für eine relationale Sprache unverzichtbar sind



- Die **Projektion** (Spaltenauswahl) stellt nur bestimmte Attribute einer Relation dar, alle anderen werden weggelassen
- Die Anzahl der Tupel bleibt unverändert
- **Beispiel:**

$\pi_{\text{MITARBEITER1}}(\text{Name, Vorname, Anschrift})$

| Name | Vorname | Anschrift |
|------------|-----------|---------------------------------|
| Lorenz | Sophia | 03725 Mausbach Mühlenweg 4 |
| Hohl | Tatjana | 49262 Mausloch Käsereistr. 5 |
| Willschrei | Theodor | 03453 Katzbergen Ahornweg 4 |
| Richter | Hans | 78943 Katzenhausen Buchenweg 28 |
| Wiesenland | Brunhilde | 02518 Hundsberg Mopsstr. 45 |



- Die **Selektion** (Auswahl) liefert Teilmenge der Tupel einer Relation
- Die Tupel der Ergebnisrelation werden anhand der zu erfüllenden Selektionsbedingung ermittelt
- Die Teilmenge enthält alle Attribute der Basisrelation
- **Beispiel:**
 $\sigma_{\text{ARBEITET_AN}}(\text{ProjektNr} = 2)$

| Personal Nr | Projekt Nr | Tätigkeit | Stunden |
|-------------|------------|--------------|---------|
| 0004 | 2 | Leitung | 25 |
| 0005 | 2 | Präsentation | 160 |

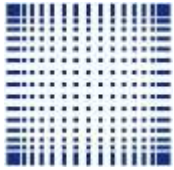


Vereinigung \cup

- Die **Vereinigung** vereinigt zwei Tupelmengen (Relationen) A und B
- Voraussetzung: A und B haben (exakt) die gleichen Attribute
- Die Vereinigung $V = A \cup B$ enthält alle Tupel, alle Attribute
- **Beispiel:** MITARBEITER := MITARBEITER1 \cup MITARBEITER2

Relation MITARBEITER

| Personal Nr | Name | Vorname | Abteilung | Anschrift |
|-------------|------------|-----------|-----------|----------------------------------|
| 0001 | Lorenz | Sophia | 1 | 03725 Mausbach Mühlenweg 4 |
| 0002 | Hohl | Tatjana | 2 | 49262 Mausloch Käsereistr. 5 |
| 0003 | Willschrei | Theodor | 2 | 03453 Katzbergen Ahornweg 4 |
| 0004 | Richter | Hans | 3 | 78943 Katzenhausen Buchenweg 28 |
| 0005 | Wiesenland | Brunhilde | 3 | 02518 Hundsberg Mopsstr. 45 |
| 0011 | Schneider | Jakob | 1 | 06238 Frankenhausen Bussardweg 7 |
| 0012 | Putzmann | Gertrud | 3 | 45345 Weidenau An der Alm 2 |



- Die **Differenz** $D := A - B$ enthält alle Tupel der Relation A, die nicht in der Relation B enthalten sind
- Voraussetzung (analog zur Vereinigung): A und B haben (exakt) die gleichen Attribute
- Die Ergebnisrelation enthält alle Attribute von A (bzw. B)
- **Beispiel:**
 $\text{DIFFERENZ} := \text{PROJEKT} - \text{PROJEKT2}$

Relation DIFFERENZ

| ProjektNr | Beschreibung |
|-----------|-------------------|
| 2 | Verkaufsmesse |
| 3 | Konkurrenzanalyse |



Durchschnitt \cap

- Der **Durchschnitt** $D := A \cap B$ enthält nur die Tupel die in beiden Relationen enthalten sind
- Voraussetzung (analog zu Vereinigung, Differenz): A und B haben (exakt) die gleichen Attribute
- **Beispiel:**
 $\text{DURCHSCHNITT} := \text{PROJEKT} \cap \text{PROJEKT2}$

Relation DURCHSCHNITT

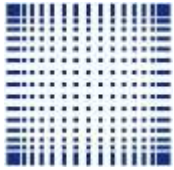
| ProjektNr | Beschreibung |
|-----------|---------------|
| 1 | Kundenumfrage |



- Attribute lassen sich in einer neuen Relation **umbenennen**
- Die Wertebereiche bleiben dabei erhalten, die Ergebnisrelation enthält alle Tupel der Basisrelation
- Zur Ausgabe von Tupeln mit aussagekräftigeren Namen
- Noch wichtiger: Zur Ausführung von Operationen, die (exakt) gleiche Attribute verlangen
- **Beispiel:**
ERGEBNIS := ARBEITET_AN_{PersonalNr←Personalnummer}

Relation ARBEITET_AN

| Personalnummer | Projekt Nr | Tätigkeit | Stunden |
|----------------|------------|-------------|---------|
| 0002 | 1 | Leitung | 25 |
| 0003 | 1 | Bearbeitung | 55 |
| 0004 | 1 | Bearbeitung | 70 |
| ... | ... | ... | ... |



Kartesisches Produkt \times

- Das **kartesische Produkt** $K = A \times B$ wird (wie in der Mengenlehre) durch Kombination aller Tupel der Relation A mit allen Tupel der Relation B gebildet
- Die Menge der Attribute ist die Vereinigungsmenge der Attributmengen der Relationen A und B (Achtung: mögliche Namenskonflikte!)
- **Beispiel:**
KPRODUKT := MITARBEITER2 \times PROJEKT

Relation KPRODUKT

| Personal Nr | Name | Vorname | Abteilung | Anschrift | Projekt Nr | Beschreibung |
|-------------|-----------|---------|-----------|-----------|------------|-------------------|
| 0011 | Schneider | Jakob | 1 | ... | 1 | Kundenumfrage |
| 0011 | Schneider | Jakob | 1 | ... | 2 | Verkaufsmesse |
| 0011 | Schneider | Jakob | 1 | ... | 3 | Konkurrenzanalyse |
| 0012 | Putzmann | Gertrud | 3 | ... | 1 | Kundenumfrage |
| 0012 | Putzmann | Gertrud | 3 | ... | 2 | Verkaufsmesse |
| 0012 | Putzmann | Gertrud | 3 | ... | 3 | Konkurrenzanalyse |



Verbund (Join) ⋈ – 1

- In der Praxis liefert das kartesische Produkt "zu viel", man interessiert sich nur für einen bestimmten Teil aller möglichen Kombinationen
- Dies liefert der **Verbund**. Die Teilmenge ergibt sich durch eine Bedingung
- Es gibt verschiedene Möglichkeiten zwei Relationen für einen Verbund zu kombinieren

Natürlicher Verbund (Natural-Join)

- Zwei Relationen R1 und R2 werden über die Gleichheit zweier Attribute miteinander verbunden
- Es werden nur die Tupel des möglichen kartesischen Produkts übernommen, bei denen eben diese Attributwerte übereinstimmen
- Das Vergleichsattribut ist in der Ergebnisrelation nur einmal enthalten (obwohl es im kartesischen Produkt zweimal vorhanden wäre)
- Namensgleichheit der Attribute wird nicht gefordert



Beispiel

- **Gesucht:** Welcher Mitarbeiter arbeitet an welchem Projekt?

Damit gesucht: Der natürliche Verbund zwischen MITARBEITER1 und ARBEITET_AN bei Gleichheit im (jeweiligen!) Attribut PersonalNr

VERBUND := MITARBEITER1 \bowtie _{PersonalNr = PersonalNr} ARBEITET_AN

| Personal Nr | Name | Vorname | Abteilung | Anschrift | Projekt Nr | Tätigkeit | Stunden |
|-------------|------------|-----------|-----------|-----------|------------|--------------|---------|
| 0002 | Hohl | Tatjana | 2 | ... | 1 | Leitung | 25 |
| 0003 | Willschrei | Theodor | 2 | ... | 1 | Bearbeitung | 55 |
| 0004 | Richter | Hans | 3 | ... | 1 | Bearbeitung | 70 |
| 0004 | Richter | Hans | 3 | ... | 2 | Leitung | 25 |
| 0005 | Wiesenland | Brunhilde | 3 | ... | 2 | Präsentation | 160 |
| 0004 | Richter | Hans | 3 | ... | 3 | Leitung | 25 |
| 0002 | Hohl | Tatjana | 2 | ... | 3 | Bearbeitung | 80 |
| 0003 | Willschrei | Theodor | 2 | ... | 3 | Bearbeitung | 65 |



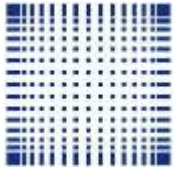
- Im **Theta-Join** sind zusätzlich die Vergleichsoperationen $<$, $>$, ... zulässig. Das Vergleichsattribut ist in der Ergebnisrelation zweimal vorhanden (da nicht zwingend identisch)
- Der **Equi-Join** erlaubt nur den Vergleichsoperator $=$, das Vergleichsattribut ist aber zweimal in der Ergebnisrelation enthalten
- Im **Self-Join** werden Tupel einer Relation miteinander verbunden
- Der **Semi-Join** (spezielle Form des Equi-Join) dient der Verknüpfung verteilter Datenbanken
- Alle bisherigen Joins sind **Inner-Joins** – Es werden nur Tupel übernommen, die die Bedingung erfüllen



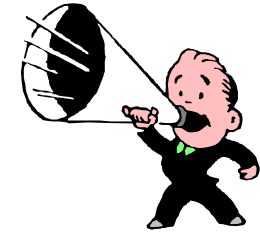
- Im **Outer-Join** werden auch Tupel übernommen, für die kein entsprechendes Tupel der anderen Relation vorhanden ist

Beispiel:

- Im eben verwendeten Beispiel würde zusätzlich auch noch der Datensatz des (an keinem Projekt beteiligten) Mitarbeiters mit der PersonalNr 0001 einmal auftauchen
- Die Attribute ProjektNr, Tätigkeit und Stunden erhalten den <null> Wert
- Drei Formen des Outer-Join
 - ◆ **Left-Outer-Join:** Es werden auf jeden Fall alle Tupel der linken Relation übernommen
 - ◆ **Right-Outer-Join:** dito für die rechte Relation
 - ◆ **symmetrischer Outer-Join:** Vereinigung von Left- und Right-Outer-Join



JETZT



Relationenkalkül

DM
3IB/3IMB/3UIB

342

WS 2017/18



- Bisher: Relationenalgebra
 - ◆ durch Operationen wird eine Ergebnismenge berechnet
 - ◆ prozedural aufgebaut
- Jetzt: Relationenkalkül
 - ◆ Deskriptive Angabe
 - ◆ ohne Angabe anzuwendender Operation
- Zwei Ausprägungen
 - ◆ Tupelkalkül (Datensatz-orientiert)
 - ◆ Domainenkalkül (Werte-orientiert)



■ **Tupelkalkül:**

$$\{ t \mid P(t) \}$$

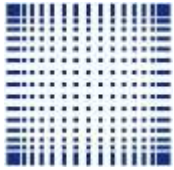
Dabei ist t die Tupelvariable und $P(t)$ das Prädikat, das erfüllt sein muss, damit t in die Ergebnisrelation aufgenommen werden soll

■ **Domainenkalkül:**

$$\{ [d_1, d_2, \dots, d_n] \mid P(d_1, d_2, \dots, d_n) \}$$

d_i ist eine Domainenvariable, die für ein Attribut steht und P wiederum ein Prädikat, das erfüllt sein muss

- ☹ sieht kompliziert aus – ist es am Anfang auch
- ☺ Beispiele folgen gleich und in der Übung



Prädikate sind logische Ausdrücke,

- verwenden neben Vergleichsoperatoren ($<$, $>$, $=$, \neq , etc.) auch folgende Zeichen
- Logisches UND (AND)
 $a \text{ AND } b \rightarrow$ wahr, wenn a und b wahr sind
- Logisches ODER (OR)
 $a \text{ OR } b \rightarrow$ wahr, wenn a oder b wahr sind
- Negation (NOT)
 $\text{NOT } a \rightarrow$ wahr, wenn a falsch ist
- Allquantor ("für alle ...")
 $(\forall x) b \rightarrow$ für alle Werte x ist der Ausdruck b wahr
- Existenzquantor ("es existiert (mindestens) ein ...")
 $(\exists x) b \rightarrow$ es existiert mindestens ein Wert x , sodass der Ausdruck b wahr ist



- **Gesucht:** Alle Personalnummern in der Relation ARBEITET_AN, die an dem Projekt mit der Nummer 1 mitarbeiten
 $\{ \text{ARBEITET_AN.PersonalNr} \mid \text{ARBEITET_AN.ProjektNr} = 1 \}$
- **Gesucht:** Alle Namen der Personen, die an Projekt Nr. 2 mitarbeiten.
Beteiligte Relationen: MITARBEITER1, ARBEITET_AN
 $\{ \text{MITARBEITER1.Name} \mid \exists (\text{ARBEITET_AN}(\text{ARBEITET_AN.ProjektNr} = 2) \text{ AND } (\text{ARBEITET_AN}(\text{ARBEITET_AN.PersonalNr} = \text{MITARBEITER1.PersonalNr}))) \}$
- **Gesucht:** Alle Namen der Personen, die an keinem Projekt arbeiten
 ARBEITET_AN as A (A wird Tupelvariable, Typ ARBEITET_AN)
 $\{ \text{MITARBEITER1.Name} \mid \forall A(A.\text{PERSONALNr} \neq \text{MITARBEITER1.PersonalNr}) \}$



- **Gesucht:** Alle Namen der Personen, die an einem Projekt mitarbeiten, das von Mitarbeiter mit der Personal Nr. 0004 geleitet wird:

ARBEITET_AN as A (A wird Tupelvariable, Typ ARBEITET_AN)

ARBEITET_AN as B (B wird Tupelvariable, Typ ARBEITET_AN)

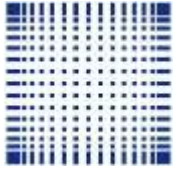
{ MITARBEITER1.Name |

\exists A(A.PersonalNr = MITARBEITER1.PersonalNr) AND

\exists B(B.Tätigkeit = 'Leitung' AND B.PersonalNr = 0004 AND

A.ProjektNr = B.ProjektNr) }

- Ist der Name 'Richter' in der Ergebnisrelation enthalten oder nicht?
 - ◆ Begründung?
 - ◆ Semantisch richtig? (mit anderen Worten: Entspricht die Ergebnisrelation der Fragestellung?)

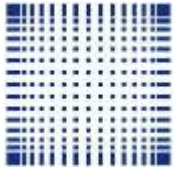


Welche Fragen gibt es?

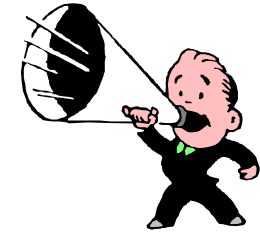


Zum Nachdenken:

*Warum funktioniert eine
Datenbankabfragesprache so
komplett anders als Java, C, C# ?
Warum gibt es keine Schleifenkonstrukte?
Warum gibt es keine Möglichkeit
nur den ersten Treffer einer Anfrage zu listen?*



JETZT



Kapitel 10

Transaktionen

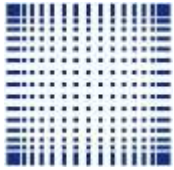




- nicht ursprünglich ein Begriff, der mit dem Relationalen Modell definiert wurde, aber ein (etwas eingehender zu betrachtender) Grundbegriff
- *Transaktion: Mehrere Operationen als zusammenhängend betrachten (in: Anforderungen an ein DBS)*
- DBS erlauben den gleichzeitigen Datenbankzugriff mehrere Benutzer
- lesend → kein Problem
- schreibend → DBMS muss für die Konsistenz der Daten sorgen



- Konsistenz muss auch gewährleistet bleiben bei Hardwareausfällen, Programmfehlern etc.
- DBMS muss daher in der Lage sein, die betroffenen Daten in den letzten konsistenten Zustand (zurück-)zu versetzen
- Transaktionenverwaltung als Komponente des DBMS
- Eine Gruppe von logisch zusammenhängenden Datenbankoperationen (für unsere Betrachtungen: in SQL), die nur gemeinsam sinnvoll (im Sinne der Wahrung von Konsistenz) ausgeführt werden können, heißt **Transaktion**



- Der Begriff Transaktion ist ein Kunstwort, der Ursprung soll in der engl. Phrase "Transformation Action" liegen. Damit beschreibt Transaktion den Vorgang, bei dem eine DB von einem konsistenten Zustand in einen anderen (nächsten) konsistenten Zustand überführt wird
- Kann – durch einen Fehler – die Transaktion nicht korrekt abgeschlossen werden, dann wird **keine** der Operationen in der DB ausgeführt und der (gesamte) Datenbestand in den Ursprungszustand (vor der Transaktion) zurückgesetzt



ACID-Eigenschaften – 1

- Das DBMS garantiert bei der Ausführung einer Transaktion die Einhaltung von vier grundsätzlichen Eigenschaften
 - Diese Eigenschaften werden auch als **ACID-Eigenschaften** bezeichnet
1. **Atomicity (Atomarität, "Alles oder Nichts")**
 - ◆ Transaktion, bestehend aus einer oder mehreren Operationen, wird entweder vollständig oder gar nicht ausgeführt
 - ◆ Tritt während der Ausführung ein Fehler auf, werden alle bisherigen Operationen der Transaktion rückgängig gemacht



2. Consistency (Konsistenz)

- ◆ Ist die Transaktion abgeschlossen, befindet sich die DB in einem konsistenten Zustand
 - ◆ Es gelten die def. Integritätsbedingungen (Wertebereiche, Schlüsseleigenschaften, Fremdschlüssel etc.), die die logische Konsistenz sichern
 - ◆ Verletzungen von Integritätsbedingungen führen zum Zurücksetzen der Transaktion
 - ◆ Während einer Transaktion ist allerdings temporäre Inkonsistenz gestattet, muss aber am Ende der Transaktion behoben sein
- So eine "verzögerte Integritätsbedingung" tritt beispielsweise bei der Umbuchung von Geldern zwischen Bankkonten auf



3. Isolation

- ◆ Transaktionen laufen isoliert ab
- ◆ Mehrere gleichzeitig ablaufende Transaktionen stören und beeinträchtigen sich nicht gegenseitig
- ◆ DBMS muss geeignet synchronisieren
- ◆ Dies ist eine der wichtigsten (und nicht gerade leicht zu realisierenden) Voraussetzungen zur Sicherung der Konsistenz der DB

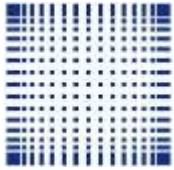


4. Durability (Dauerhaftigkeit)

- ◆ Das Ergebnis einer erfolgreichen Transaktion ist dauerhaft (persistent)
- ◆ Bedeutet, dass selbst beim Auftreten von Fehlern bei der Übertragung des **Transaktionsergebnis** in die DB die Änderungen vollständig durchgeführt werden.



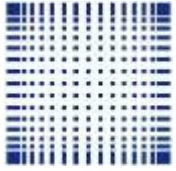
- Konnte eine Transaktion erfolgreich abgeschlossen werden, wird dies als **Commit** bezeichnet. Die Transaktion wurde committed (Denglish ☹)
- Musste die Transaktion – bedingt durch einen Fehler – zurückgesetzt werden, spricht man von einem **Rollback**
- Nicht nur Fehler können ein Rollback auslösen, Anwendungen und Benutzer ebenso
- d.h. es gibt entsprechende SQL-Anweisungen für Commit und Rollback



- Kontoverwaltung bei einer Bank
 - ◆ Von einem Girokonto A soll eine Rechnung beglichen werden
 - ◆ Zwei Vorgänge:
 1. Girokonto A belasten
 2. dem Empfänger-Girokonto B gleichen Betrag gutschreiben
 - ◆ Was passiert bei Systemabsturz nach dem ersten Schritt? Geld weg?
 - ◆ Transaktionsverwaltung stellt in diesem Fall Urzustand her
- Transaktionsverwaltungen stellen für SQL-Server eine Selbstverständlichkeit dar – für Desktop-DBS nicht unbedingt



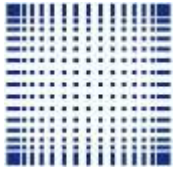
Welche Fragen gibt es?



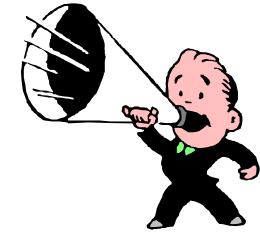
Zum Nachdenken:

***Wie könnte man die Probleme
bei der Banküberweisung lösen?***

***Hinweis: Wie überzeugen Sie sich
„in der realen Welt“, dass eine Nachricht
angekommen ist?***



JETZT



Kapitel 11

Java Database Connectivity (JDBC)

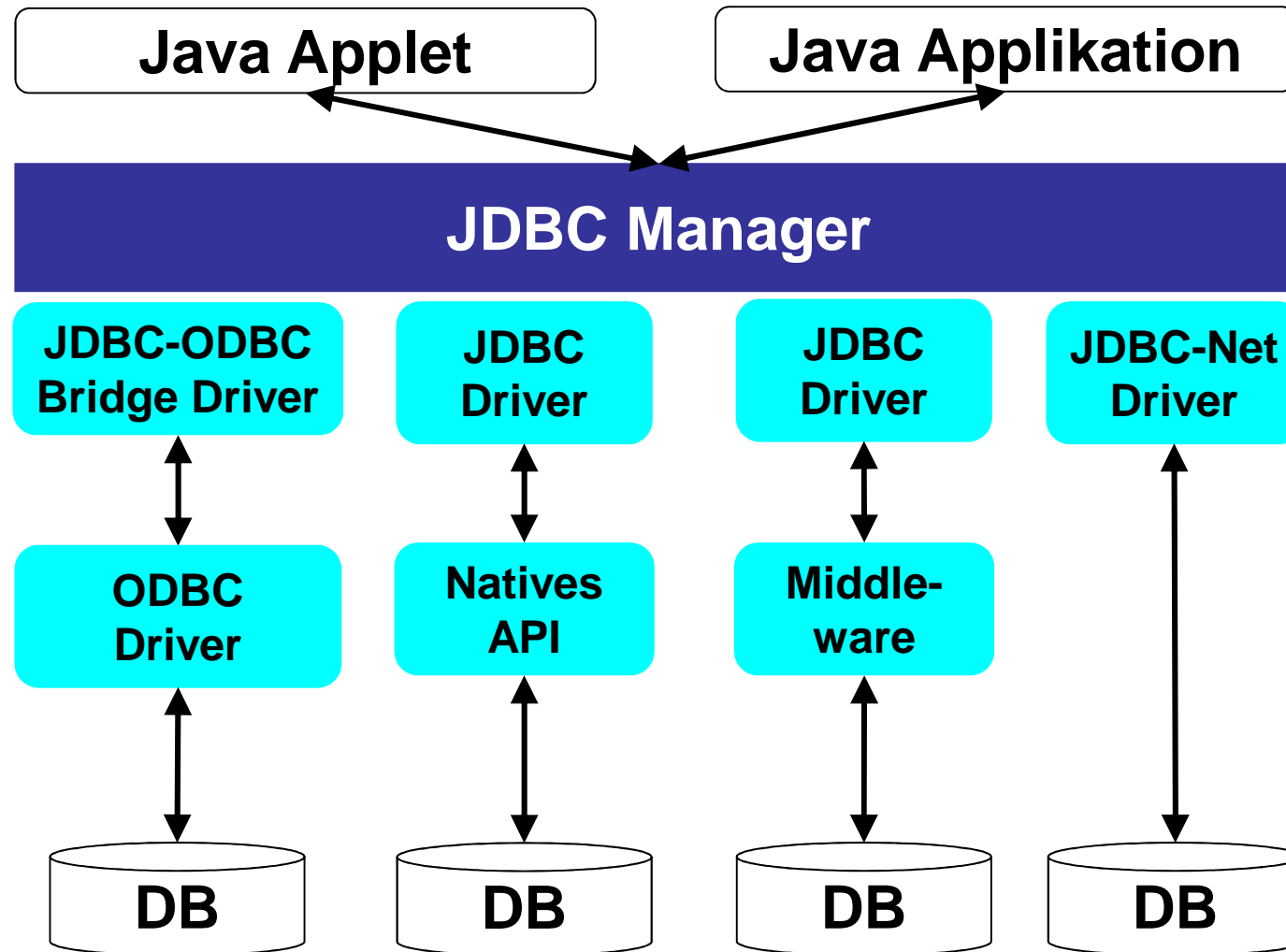


Ein paar Begriffe noch vorweg ...

- **Trigger** – eine Art Eventsteuerung für Datensätze
- **Stored Procedures** – das Pendant für Makros in MS Word oder anderen Office Applikationen
- **Dump** – das Back Up einer Datenbank
- **Replikation** – Spiegelung einer Datenbank zur Performancesteigerung und/oder Datensicherung

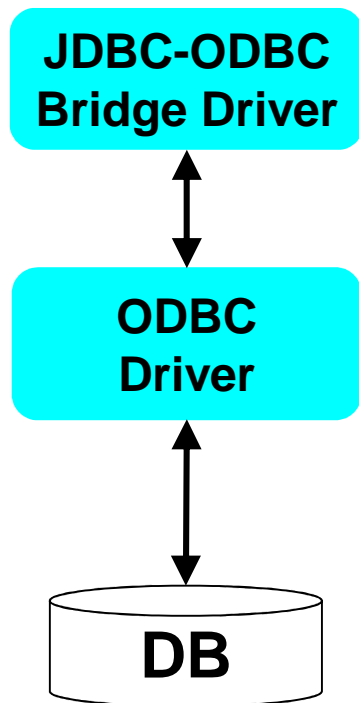


- JDBC steht für Java Database Connectivity
- wurde 1996 von JavaSoft entwickelt
- Idee von JDBC ähnelt Open Database Connectivity (ODBC) sehr stark
- JDBC basiert ebenfalls auf dem CLI von X/Open und der SQL Access Group
- JDBC ermöglicht die Datenbankbindung für Java Applets und Java Applikationen gleichermaßen
 - ◆ JDBC bietet ebenfalls eine "Brücke" zu ODBC
- Das JDBC API ist Bestandteil des "Standard" Java SDK





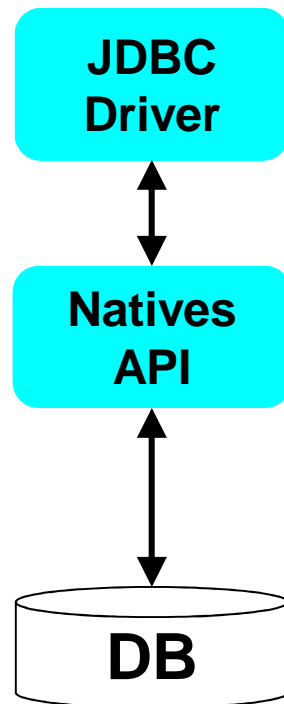
JDBC Driver – Typ 1



- Der JDBC Driver Typ 1 steht für die JDBC-ODBC Bridge
- Er war als erster verfügbar
- Die Bridge übersetzt die JDBC-Calls in ODBC-Calls
- Die JDBC-ODBC Bridge ist damit nichts als anderes als eine "Java-" Fassade für eine ODBC-Datenquelle
- **Bis heute im Einsatz und manchmal „die einzige Möglichkeit“!**



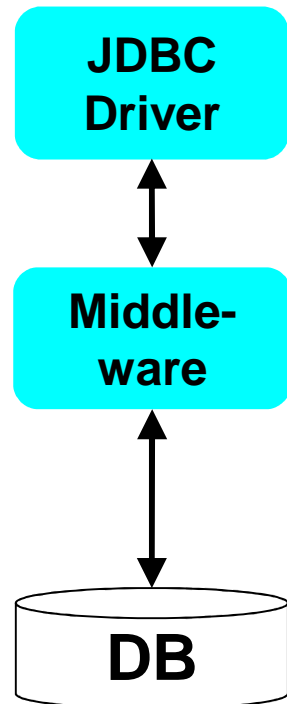
JDBC Driver – Typ 2



- JDBC Driver vom Typ 2 verwenden Programmierschnittstellen (API), die vom Hersteller der Datenbank definiert wurden
- Dieses native API ist in der Regel plattformabhängig, während der JDBC Driver in Java programmiert ist
- Zwischen DB und nativem API wird ein proprietäres Protokoll verwendet
- **VERALTET!**



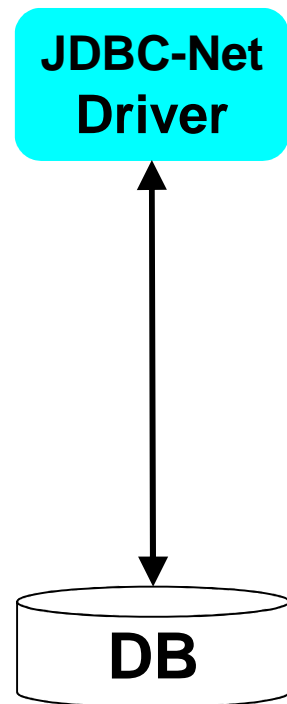
JDBC Driver – Typ 3



- JDBC Driver vom Typ 3 greifen per Netzwerkprotokoll auf eine sog. Middleware (auch: Database Access Server) zu
- Die Middleware kommuniziert dann mit der DB über eigene Treiber
- Der Treiber als solcher ist damit plattformunabhängig in Java entwickelt
- Dieses Verfahren bringt häufig Nachteile in der Performance
- **VERALTET!**



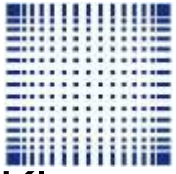
JDBC Driver – Typ 4



- Der Zugriff bei JDBC Drivern vom Typ 4 erfolgt über Sockets bzw. das Netzprotokoll der Datenbank selbst
- Der Treiber kommuniziert direkt im proprietären Protokoll des DBMS, ist aber komplett plattformunabhängig in Java entwickelt
- Durch die Auslassung aller Zwischenstufen ist dies die schnellste Verbindungsart
- **OPTIMALE Lösung!**



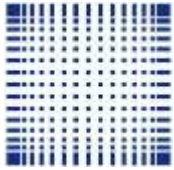
- Java Software Development Kit (SDK), Standard Edition (SE)
- Den JDBC-Treiber für MySQL (JConnector)
<http://dev.mysql.com/downloads>
- Eine MySQL-Datenbank mit mindestens einer Tabelle darin, die Datensätze enthält
- Nicht zwingend notwendig, aber "realitätsnäher":
In MySQL sollten Benutzer eingerichtet sein
- Vorteilhaft: eine Java-Entwicklungsumgebung wie z.B. Eclipse



Beispiel – 1

```
// Klassen für JDBC importieren  
import java.sql.*;
```

```
public class MySQLJDBC {  
    public static void main(String[] args) throws Exception {  
        // Treiber laden  
        Class.forName("com.mysql.jdbc.Driver");  
        // Verbindung aufbauen  
        Connection con =  
            DriverManager.getConnection(  
                "jdbc:mysql://codd/uebungXX", "dbabscXX", "abd");  
        // Statement-Objekt erzeugen  
        Statement stmt = con.createStatement();  
        // Abfrage ausführen  
        ResultSet res = stmt.executeQuery("SELECT * FROM t_ma_dt");  
    }  
}
```



Beispiel – 2

```
// Ergebnis ausgeben
while(res.next()) {
    String vorname = res.getString("vname");
    String name = res.getString("name");
    String strasse = res.getString("str");
    String plz = res.getString("plz");
    String wohnort = res.getString("ort");
    int alter = res.getInt("alt");
    System.out.print(vorname+" "+name+" (" +alter+ ")"+"; "+strasse);
    System.out.println("; "+plz+" "+wohnort);
}
// Verbindung schliessen
con.close();
}
}
```

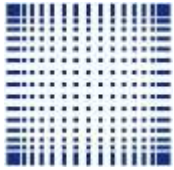


Beispiel – 3

```
Command Prompt
C:\pv3\Block5>java MySQLJDBC
Elsbeth Haas <42>; Berliner Str. 223; 63067 Offenbach
Hans Richter <32>; Frankfurter Str. 61; 63067 Offenbach
Irmgard Friedrich <40>; Goethestr. 61; 63067 Offenbach
Jochen Hartmann <29>; Berliner Str. 223; 60528 Frankfurt
Martin Goldbach <35>; Frankfurter Str. 61; 60529 Frankfurt
Norbert Naumann <38>; Goethestr. 61; 60594 Frankfurt
Tanja Haas <36>; Berliner Str. 223; 30323 Hannover
Martin Neppe <43>; Goethestr. 61; 30324 Hannover
```

Der Vorteil der JDBC Normung:

- Nur die Aufrufe zum Treiber laden und zur Herstellung der Verbindung müssen bei anderem Treiber bzw. anderem DBMS angepasst werden



- Sowohl
 - ◆ den ODBC-Driver als auch
 - ◆ den JDBC-Driverfür MySQL finden Sie auf dev.mysql.com unter der Überschrift Connector zum Download
- Es gibt weitere (JDBC, ODBC; in der Regel kommerzielle) Treiberprodukte für MySQL
- Die Verbindung Java Applet, Applikation \Leftrightarrow MySQL DB kann entweder via
 - ◆ JDBC-ODBC Bridge (Typ 1) oder
 - ◆ JDBC-Driver (Typ 4; diese Lösung ist vorzuziehen ☺)realisiert werden.



- Um die Verbindung herstellen zu können, werden Interfaces und Klassen benötigt, welche die JDBC-Technologie zur Verfügung stellt
- Über den Treibermanager wird ein bestimmter JDBC-Treiber geladen, bzw. ein Treiber registriert sich bei ihm
- JDBC stellt hierfür lediglich ein Interface API zur Verfügung
 - ◆ d.h. es besitzt keinen Programmcode
 - ◆ sondern der JDBC-Treiber muss die entsprechenden Interfaces implementieren
- Durch die Verwendung dieser Interfaces wird der entsprechende Aufruf in den Treiber umgemappt



- Die JDBC-Interfaces und die Klasse zur Arbeit mit dem Treibermanager sind im Package `java.sql` zu finden:
- **DriverManager**
Diese Klasse verwaltet die geladenen JDBC-Treiber für die Datenbanksysteme und stellt die Verbindung zu einer Datenbank her
- **Connection**
Objekte des Typs des Interfaces `Connection` repräsentieren eine physische Verbindung zu einer bestimmten Datenbank über einen festen Treiber



■ **Statement**

Mit den Methoden des Interfaces Statement können SQL-Anfragen an die Datenbank über eine bestehende Datenbankverbindung gesendet werden

■ **ResultSet**

Objekte vom Typ dieses Interfaces repräsentieren tabellarisch das Ergebnis einer SQL-Datenbankanfrage

- Alle diese Bestandteile bauen innerhalb einer JDBC-Anwendung untereinander Beziehungen auf.



Aufbau einer DB-Verbindung – 1

Einmalig:

- Installation eines Datenbankservers (lokal oder im Netz)
- Installation eines JDBC-Treibers

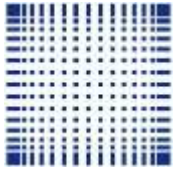
In der Anwendung:

- Laden des Treibermanagers mit dem entsprechenden JDBC-Treiber
- Herstellen einer Verbindung zur Datenbank

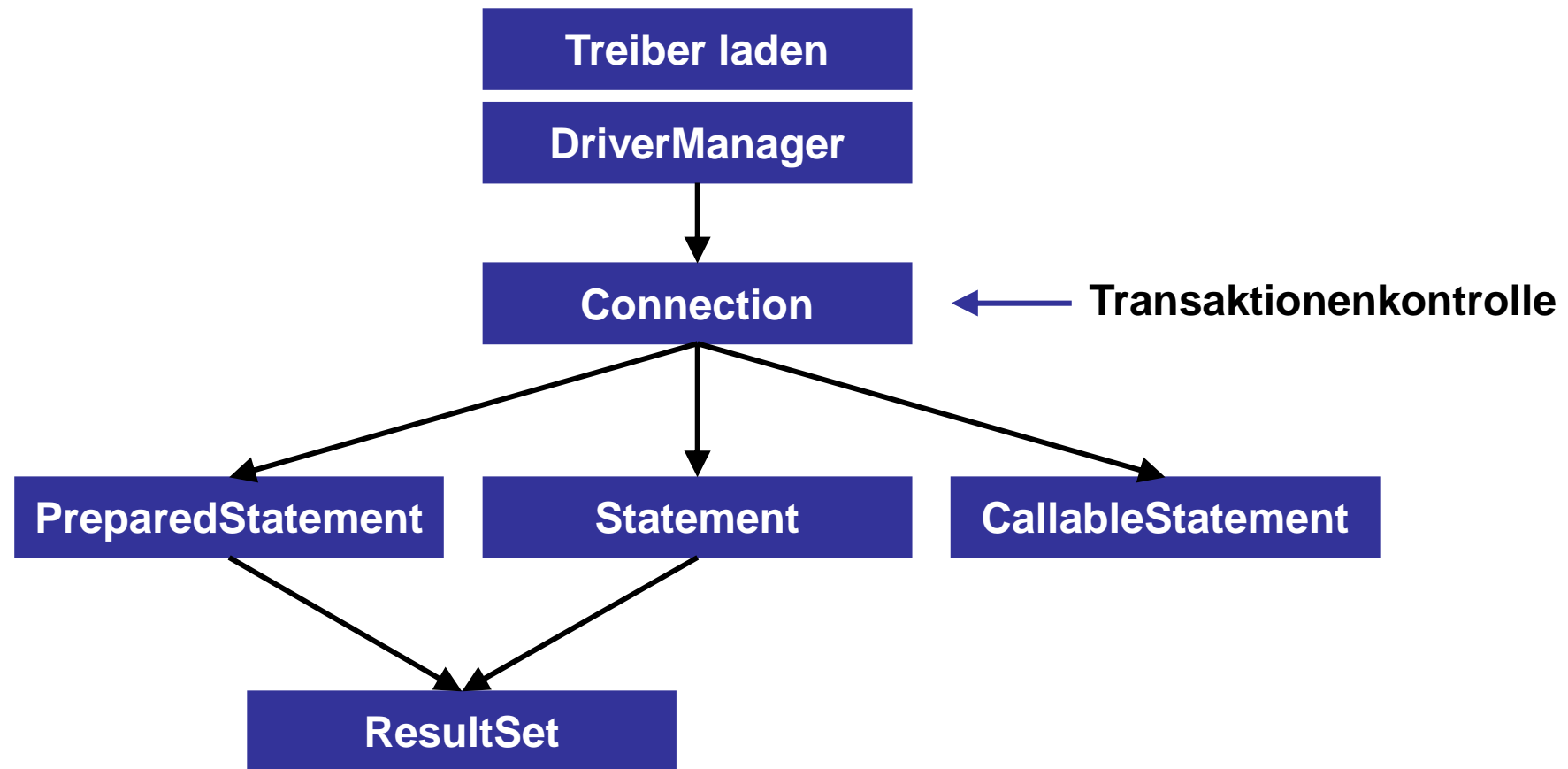


Aufbau einer DB-Verbindung – 2

- SQL-Anweisung an die Datenbank richten
 - ◆ zum Bearbeiten der Struktur der Datenbank (Tabellen anlegen, DDL, ...). Es wird keine Ergebnismenge geliefert
 - ◆ zum Bearbeiten von Daten (einfügen, ändern, löschen). Es wird keine Ergebnismenge geliefert
 - ◆ zum Abfragen/Selektieren von Daten. Es wird eine Ergebnismenge geliefert
- Eine ggf. zurück gelieferte Ergebnismenge wird ausgewertet bzw. bearbeitet



Aufbau einer DB-Verbindung – 3

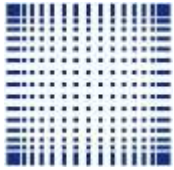




Vorsicht *FALLE*



- Über das JDBC API können Sie **keine Datenbanken** erzeugen
- Der Versuch ein CREATE DATABASE Statement abzusetzen wird bewusst vom JDBC Treiber verhindert. Warum?
- Temporäre Daten müssen daher als (ggf. temporäre) Tabellen in einer bestehenden Datenbank angelegt werden



Laden des Treibers

```
try {  
    Class.forName("com.mysql.jdbc.Driver");  
}  
catch (ClassNotFoundException e) {  
    e.printStackTrace();  
    System.exit(1);  
}
```

- Der JDBC Treiber ist eine ganz "normale" Java-Klasse
- Der Pfad zum Treiber muss im Klassenpfad (Umgebungsvariable CLASSPATH) bekannt gemacht worden sein
- Der Treiber wird – sofern er gefunden wird – über die Klasse `java.lang.Class` geladen und automatisch beim Treibermanager registriert



Verbindung zur DB – 1

- Nach der Registrierung des Treibers kann eine Verbindung zur Datenbank aufgebaut.
- Dazu besitzt der Treibermanager die Methode `getConnection()`
- Parameter:
 - ◆ Der Pfad zur Datenbank (ggf. weitere Parameter für die DB)
optional:
 - ◆ Anmeldenamen
 - ◆ Passwort
- Kann keine Verbindung hergestellt werden, wird eine `SQLException` ausgelöst
- Wird die Verbindung nicht mehr benötigt, sollte sie explizit mit der Methode `close()` geschlossen werden



Verbindung zur DB – 2

```
Connection con;  
con = DriverManager.getConnection(String url, String name, String passwort);  
// oder ...  
con = DriverManager.getConnection(String url);
```

- Der Aufbau des Parameters url ist Treiber-abhängig ...
- ... folgt aber dem Aufbau jdbc:<subprotocol>:<subname>
- <subname> enthält die Informationen (Name, Pfad, Parameter) zur gewünschten DB

Beispiele:

- jdbc:odbc:Uebungen
- jdbc:interbase://localhost/D:/daten/examples/employee.gdb
- jdbc:mysql://127.0.0.1/uebungen



Verbindung zur DB – 3

```
Connection con;
// An dieser Stelle Treiber registrieren ...
// ... Jetzt die Verbindung zur Datenbank aufbauen:
try {
    con = DriverManager.getConnection(
        "jdbc:mysql://localhost/uebungen","dbabsc00","abd");
    System.out.println("Verbindung hergestellt!");
    // Datenabfragen etc.
    con.close();
    System.out.println("Verbindung beendet!");
}
catch (SQLException e) {
    System.out.println("Konnte keine Verbindung herstellen!");
    System.exit(1);
}
```



- Zur Ausführung von SQL-Anweisungen stehen drei Interfaces zur Verfügung:
 - ◆ **Statement**
Über dieses Interface sollten Sie einmalige Anweisungen ausführen
 - ◆ **PreparedStatement**
Wenn Sie eine SQL-Anweisung mehrfach ausführen oder Parameter in einer SQL-Anweisung verwenden, sollten Sie dieses Interface nutzen
 - ◆ **CallableStatement**
Dieses Interface wird verwendet, um in mySQL abgelegte Stored Procedures zu nutzen.



- Anweisungen werden in SQL-Syntax geschrieben ...
- ... und beim Methodenaufruf als String übergeben
 - ◆ **Vorteil** 👍: Einfaches Handling
 - ◆ **Nachteil** 👎: Syntax wird erst zur Laufzeit geprüft – also auch hier mit Exceptionbehandlung arbeiten!

Beispiel:

```
Connection con;
```

```
// ...
```

```
Statement sqlAnw = con.createStatement();
```

```
sqlAnw.executeQuery("SELECT * FROM t_ma_dt");
```

Wo sind die
Datensätze?



- Zuerst Statement-Objekt erzeugen:
Methode `createStatement()` – gebunden an ein `Connection` Objekt!
- Dann über die Methoden
 - ◆ `execute`,
 - ◆ `executeQuery` oder
 - ◆ `executeUpdate`die eigentliche SQL-Anweisung ausführen.
- **`ResultSet executeQuery(String sql)`**
Diese Methode dient zum Abfragen von Daten (SELECT-Anweisungen) und liefert das Ergebnis in einem `ResultSet` zurück



■ **int executeUpdate(String sql)**

Auf diesem Weg können Strukturen in der Datenbank definiert und Daten manipuliert werden (CREATE, INSERT, UPDATE und DELETE-Anweisungen). Der Rückgabewert gibt die Anzahl der Zeilen an, die von der Änderung betroffen waren, oder 0.

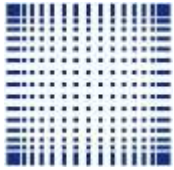
■ **boolean execute(String sql)**

boolean getMoreResults()

ResultSet getResultSet()

int getUpdateCount()

Beim dynamischen Ausführen von SQL-Anweisungen oder bei Stored Procedures können SQL-Anweisungen zu mehreren oder unterschiedlichen Ergebnistypen (ResultSet oder int) führen. Diese Anweisungen können mit execute ausgeführt werden. Ist der Rückgabewert true, liegt als Ergebnis ein ResultSet vor, ansonsten ein Wert vom Typ int.



■ **void addBatch(String sql)**
int[] executeBatch()

Über die Methode `addBatch` können Sie SQL-Anweisungen zum Bearbeiten von Datensätzen einer Liste von Anweisungen hinzufügen. Sie dürfen keine Anweisungen verwenden, die eine Ergebnismenge zurückliefern. Die Anweisungen werden über die Methode `executeBatch` ausgeführt, wobei im zurück gelieferten `int`-Array pro Anweisung ein Wert steht, der die entsprechende Information zur Auslieferung liefert.



Ohne Ergebnismenge

- executeUpdate für INSERT-, UPDATE-, DELETE-, CREATE-, ALTER-, DROP-Anweisungen
- Also allgemein: DDL- und DML- Sprachkonstrukte, die keine Ergebnismenge zurückliefern

Beispiele:

- `sqlAnw.executeUpdate("CREATE TABLE t_ku (nr INTEGER, name VARCHAR(50))");`
- `sqlAnw.executeUpdate("INSERT INTO t_ku (nr, name) VALUES (15, 'Fritz Froehlich')");`



Vorbereitete SQL-Anweisungen – 1

- Vorbereitete SQL-Anweisungen können durch das DBMS "vorbehandelt" (parsen, vorcompilieren, optimieren) werden
→ Performancegewinne

Beispiel:

```
PreparedStatement stmt = con.prepareStatement("SELECT *  
                                           FROM t_ma_dt);  
stmt.executeQuery();
```

- Bereits beim Anlegen des PreparedStatement-Objektes wird die SQL-Anweisung übergeben, ...
- ... dafür wird sie beim Aufruf der Methoden executeQuery() bzw. executeUpdate weggelassen
- Interessanter sind natürlich parametrisierbare Anweisungen 😊



Vorbereitete SQL-Anweisungen – 2

Die Erstellung läuft wie folgt ab:

- Erstellen Sie eine SQL-Anweisung
- Ersetzen Sie alle Parameter durch ein Fragezeichen. Nur die Datenwerte in einer Anweisung (also nicht Tabellennamen, Spaltennamen etc.) dürfen als Parameter genutzt werden

Vor jeder Abfrage:

- Ersetzen Sie die Platzhalter durch die gewünschten Parameter
- Führen Sie die Abfrage aus



Vorbereitete SQL-Anweisungen – 3

Parameterersetzung:

- durch spezielle Methoden des Interfaces PreparedStatement, können gesetzt und gelesen werden
- Dazu steht für jeden SQL-Datentyp jeweils eine setXXX- und eine getXXX-Methode zur Verfügung
- Der erste Wert steht für den Index des Platzhalters (1..n), der zweite (bei setXXX) ist vom Datentyp des betreffenden Attributes

Beispiele:

```
void setString(int parameterIndex, String value)
void setDouble(int parameterIndex, double value)
void setInt(int parameterIndex, int value)
double getDouble(int parameterIndex)
```



```
Connection con;  
// Treiber registrieren, Datenbankverbindung herstellen  
// ...
```

```
PreparedStatement vorbSQL = con.prepareStatement(  
    "INSERT INTO t_ku (nr, name) VALUES (?, ?)");  
for (int i=1; i < 11; i++) {  
    vorbSQL.setInt( 1 , i );  
    vorbSQL.setString(2, "Kunde" + i );  
    vorbSQL.executeUpdate();  
}  
con.close();  
// ...
```

**Was passiert
hier?**



Mit Ergebnismenge

- Die Ergebnismenge wird in einem Objekt vom Typ ResultSet abgelegt
- Allerdings nicht die gesamte Datenmenge in diesem Objekt abgelegt. Das Objekt ist vielmehr ein Cursor, der den Zugriff auf jeweils einen (den aktuell betrachteten) Datensatz erlaubt
- Dieser Cursor kann durch die gesamte Ergebnismenge bewegt werden

Beispiel:

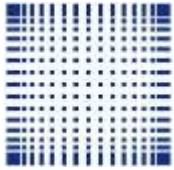
```
ResultSet rs = sqlAnw.executeQuery("SELECT * FROM Kunden");
while (rs.next()) {
    // ... hier wird irgendwas mit dem jeweils aktuellen Datensatz
    // ... gemacht, z.B. ausgelesen und auf der Console ausgegeben
}
```



- Standardmäßig steht der Cursor nach der Ausführung einer SELECT-Anweisung **vor** dem ersten Datensatz der Ergebnismenge
- Der Cursor kann mit den folgenden Methoden durch die Ergebnismenge navigiert werden:
 - ◆ **boolean next()**
Der Datensatzzeiger wird auf dem nächsten Datensatz positioniert. Ist kein Datensatz mehr vorhanden, wird false zurückgeliefert
 - ◆ **boolean previous()**
Der Datensatzzeiger wird auf dem vorigen Datensatz positioniert. Ist kein Datensatz mehr vorhanden, wird false zurückgeliefert



- ◆ **boolean absolute(int row)**
Bewegt den Datensatzzeiger auf den entsprechenden Datensatz (1..n). Negative Zahlen positionieren den Cursor relativ zum Ende der Ergebnismenge
- ◆ **void afterLast()**
Der Cursor befindet sich hinter dem letzten Datensatz
- ◆ **void beforeFirst()**
Der Cursor steht vor dem ersten Datensatz
- ◆ **boolean first()**
Der Cursor wird auf den ersten Datensatz positioniert



- ◆ **int getRow()**
Die aktuelle "Zeilennummer" in der Ergebnismenge wird geliefert
- ◆ **boolean last()**
Der Datensatzzeiger wird auf dem letzten Datensatz positioniert
- ◆ **boolean relative(int rows)**
Ausgehend von der aktuellen Position wird der Cursor relativ in die entsprechende Richtung (positiver Parameter vorwärts, ansonsten rückwärts) bewegt



- ◆ `boolean isAfterLast()`
`boolean isBeforeFirst()`
`boolean isFirst()`
`boolean isLast()`

Über diese Methoden können Sie die aktuelle Position des Cursors erfragen. Diese Methoden können ebenfalls geeignet zum Durchlaufen einer Ergebnismenge eingesetzt werden.



Ergebnismenge konfigurieren – 1

- Die schlechte Nachricht ☹
Ergebnismengen können per Standardeinstellung **nur vorwärts** durchlaufen werden!
- Rückwärts durchlaufen geht doch 😊
indem man die Ergebnismenge konfiguriert.
- Das geht über einen (überladenen) Aufruf von createStatement
**Statement createStatement(int resultSetType,
int resultSetConcurrency)**
- Die erlaubten Parameter sind als Konstanten in der Klasse ResultSet abgelegt ...
- ... für resultSetType
 - ◆ **ResultSet.TYPE_FORWARD_ONLY**
(Standardeinstellung) Nur vorwärts durchlaufen möglich



Ergebnismenge konfigurieren – 2

- ◆ **ResultSet.TYPE_SCROLL_INSENSITIVE**

Die Ergebnismenge kann in beiden Richtungen durchlaufen werden. Änderungen an der Ergebnismenge, z.B. durch andere Benutzer werden nicht sofort sichtbar

- ◆ **ResultSet.TYPE_SCROLL_SENSITIVE**

Auch hier können Sie die Ergebnismenge in beiden Richtungen durchlaufen. Änderungen an der Ergebnismenge werden allerdings sofort sichtbar



Ergebnismenge konfigurieren – 3

- ... für resultSetConcurrency
 - ◆ **ResultSet.CONCUR_READ_ONLY**
(Standardeinstellung) Auf die Ergebnismenge ist ausschließlich lesender Zugriff möglich
 - ◆ **ResultSet.CONCUR_UPDATEABLE**
Sie können die Ergebnismenge bearbeiten (*später ...*)

WICHTIG:

- Das Setzen dieser Parameter hat nur dann eine Auswirkung, wenn **sowohl**
 - ◆ das DBMS **als auch**
 - ◆ der JDBC-Treiber diese Funktionalitäten unterstützen!



Ergebnisse auslesen – 1

- Ist der Cursor auf einen Datensatz positioniert, so erlaubt JDBC die Werte Spaltenweise auszulesen.
- Allerdings müssen Sie wissen welcher Datentyp in welcher Spalte verwendet wird!
- Das Interface ResultSet stellt für eine große Anzahl Datentypen getXXX-Methoden zur Verfügung

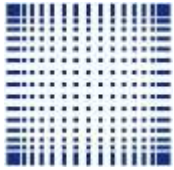
Es gibt zwei Möglichkeiten zum Auslesen der Werte:

- `double getDouble (int columnIndex)`
Liefert den Wert der angegebenen Spalte (1..n) zurück. Schneller aber schlechter lesbar im Quelltext.
- `double getDouble(String columnName)`
Liefert den Wert der über ihren Namen angegebenen Spalte zurück. Besser lesbarer aber langsamer.



Ergebnisse auslesen – 2

- `boolean getBoolean(String columnName)`
 - `byte getByte(String columnName)`
 - `byte[] getBytes(String columnName)`
 - `Date getDate(String columnName)`
 - `float getFloat(String columnName)`
 - `int getInt(String columnName)`
 - `long getLong(String columnName)`
 - `short getShort(String columnName)`
 - `String getString(String columnName)`
 - `Time getTime(String columnName)`
 - `Timestamp getTimestamp(String columnName)`
- ... alle Methoden ebenfalls mit `(int columnIndex)` als Parameter



- Nullwerte sind keine Datenwerte, sondern symbolische Werte, die einen nicht vorhandenen Wert anzeigen
- Eine leere Zeichenkette oder der Integer Wert 0 sind keine Nullwerte
- Die getXXX() Methoden können also keine Nullwerte zurückliefern!
- Die Abfrage funktioniert über die Methode wasNull():

```
while (result.set.next()) {  
    String s = resultSet.getString("name");  
    if (resultSet.wasNull())  
        s = null;  
}
```

immer bezogen auf die
letzte getXXX()
Methode!



Ergebnismenge modifizieren – 1

- Einige DBMS erlauben das bearbeiten der Ergebnismenge
 - ◆ Wertänderung in Tupeln
 - ◆ Löschen von Datensätzen
 - ◆ aber auch das Hinzufügen von Datensätzen zu einer Ergebnismenge
 - Vergleiche hierzu die Folien Ergebnismenge konfigurieren
 - Mit den Methoden
 - ◆ `int getType()` bzw.
 - ◆ `int getConcurrency()`
- können die für das jeweilige ResultSet gesetzten Parameter abgefragt werden



Ergebnismenge modifizieren – 2

Folgende Methoden dienen zur Bearbeitung der Ergebnismenge:

- **void deleteRow()**

Löscht die aktuelle Zeile (= Datensatz), auf der der Cursor steht

- **void moveToInsertRow()**

Bewegt den Cursor auf eine sog. Einfügezeile, in der ein neuer Datensatz angelegt werden kann

- **update<type>(int columnIndex, <Type> value)**
update<type>(String columnName, <Type> value)

In Analogie zu den getXXX-Methoden können so die Werte in der Einfügezeile bzw. dem aktuellem Datensatz gesetzt werden.

<Type> steht für den jeweiligen Datentyp des Attributes



Ergebnismenge modifizieren – 3

- **void insertRow()**

Fügt die Einfügezeile in die Ergebnismenge und in die Datenbank ein

- **void updateRow()**

Schreibt die Änderung an einem Datensatz fest in die Datenbank

- **void cancelRowUpdates()**

Widerruft alle Änderungen an einem Datensatz der Ergebnismenge

- **void moveToCurrentRow()**

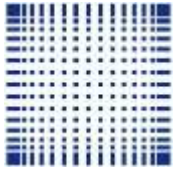
Bewegt den Cursor auf die Position zurück, die er vor dem Einfügen hatte



- Transaktionen: Mehrere SQL-Anweisungen als eine, unzertrennbare Einheit ansehen
- Defaulteinstellung von JDBC ist, dass jede Anweisung für sich sofort durch das DBMS ausgeführt und als abgeschlossen betrachtet wird.
- Man nennt das auch Auto-Commit. Dieser Auto-Commit-Modus kann aber auch deaktiviert werden und Sie erhalten so die Möglichkeit Transaktionen aus mehreren Anweisungen zu steuern
- Das DBMS muss Transaktionskonzepte unterstützen!



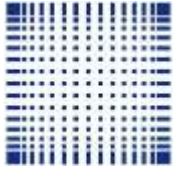
- Das Objekt vom Typ Connection kennt die folgenden Methoden zur Verwaltung von Transaktionen:
 - ◆ **void setAutoCommit(boolean autoCommit)**
Schaltet den Auto-Commit-Modus ein (true) oder aus
 - ◆ **void commit()**
Schreibt die Änderungen, die durch eine Transaktion in die Datenbank durchgeführt wurden, fest
 - ◆ **void rollback()**
Macht die Änderungen, die durch die SQL-Anweisungen einer Transaktion in der Datenbank durchgeführt wurden, rückgängig



```
Connection con;  
// ...  
Statement stmt = con.createStatement();  
try {  
    con.setAutoCommit(false);  
    stmt.executeUpdate("UPDATE t_ku SET name = 'Lukas Lehmann'  
                        WHERE name = 'Markus Maier'");  
    stmt.executeUpdate("UPDATE t_ku SET nr = 13 WHERE nr = 12);  
    con.commit(); // Transaktion durchführen  
}  
catch (SQLException e) {  
    con.rollback(); // Abbruch der Transaktion  
}
```



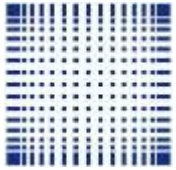
Welche Fragen gibt es?



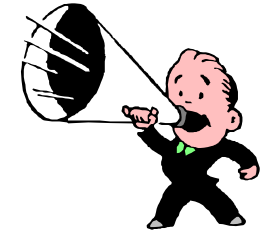
Zum Nachdenken:

***Wieso konzipiert man für JDBC
eine ODBC Bridge?***

***Wieso sieht JDBC die
Erzeugung einer Datenbank nicht vor?***



JETZT



Kapitel 12

Big Data und MongoDB



Was ist passiert?

- Die Datenmengen wurden immer größer aus immer mehr (ggf. nicht eigenen Quellen)
- „Noch einen Server“ dazuzustellen brachte bald nicht mehr die gewünschten Steigerungen
- Immer mehr Web Applikationen → Skalierbarkeit?
- Ursache war die **JOIN Problematik**

- Erste DBMS mit In-Memory Technik
- Storage Engines, Key Value Stores
- Schließlich die **Document Stores**



Ein häufig benutzter Begriff

- Big Data: What is it?
- Big Data is like teenage sex:
 - ◆ Everyone talks about it.
 - ◆ Nobody really knows how to do it.
 - ◆ Everyone thinks everyone else is doing it.
 - ◆ So, everyone claims they are doing it.

Quelle: 9gag.com



Was ist *Big Data*?

- 2012 zum Trend durch die BITKOM erhoben
- 2014 Leitthema der CeBIT

in den Schlagzeilen ...

- Wahlkampfkampagne Obama
- NSA Skandal
- Big Data umfasst i.W. vier Facetten,
die vier „V“s



Die vier „V“s von Big Data – 1

Datenmenge (Volume)

- Anzahl von Datensätzen und Dateien
- Yottabytes, Zettabytes, Exabytes, Petabytes, Terabytes

Geschwindigkeit (Velocity)

- Datengenerierung in hoher Geschwindigkeit
- Übertragung der konstant erzeugten Daten
- Echtzeit
- Millisekunden vs. Stunden



Die vier „V“s von Big Data – 2

Datenvielfalt (Variety)

- Fremddaten (Web, Soziale Netze etc.), Firmendaten
- unstrukturierte, semi-strukturierte, strukturierte Daten: Präsentationen, Texte, Videos, Bilder, Tweets, Blogs etc.
- Kommunikation zwischen Maschinen

Analyse (damit es passt: Value)

- Bedeutungen, Muster, Vorhersagemodelle
- Data Mining, Text Mining, neuronale Netze, Bildanalyse
- Visual Analytics
- Echtzeit



Wie alles begann ...

- Den Anstoß lieferten die Web 2.0 Anwendungen wie Amazon, Google, Facebook, Twitter
- Schnelle Auswertungen und Antwortzeiten, die unabhängig von der Anzahl der Nutzer sind
- Verteilte DBMS
- 2010 SAP HANA, Abfragen mit SQL
- Besinnung auf die *eigentliche* Aufgabe eines DBMS: Speichern!
- ... zu Lasten der Integritätsprüfung und Redundanz!
- Konzept NoSQL (Not only SQL)



Fakten

- MongoDB ist wahrscheinlich inzwischen der verbreitetste Vertreter unter den NoSQL Datenbanken
- MTV, CERN, New York Times, Source Forge und viele andere große Unternehmen verwenden MongoDB

Zum Nachdenken

- Wenn Sie an eine Drei-Schicht-Architektur für eine Webanwendung denken: Wo sitzt die „externe Ebene“ in diesem Modell?
- Formulieren Sie die Kernidee für NoSQL!



- Bei etwa 3 GHz ist Schluss mit der Taktung von CPUs
- Daten fallen in immer größerer Mengen bei immer kürzeren Zyklen an
- Die objektorientierte Programmierung muss für relationale DBMS das Datenmodell übersetzen

Bekannteste Ansätze aus der NoSQL Welt:

- Objektorientierte Datenbanken
- Objektrelationale Mapper
- Key-Value-Stores
- Dokumentendatenbanken
(Key-Value, Objekte oder Graphen als „Dokumente“)



- MongoDB ist eine verteilte Dokumentendatenbank

Ziele

- **Flexibilität:** schemalosen Datenmodell, dass mit der Datenbankanwendung mitwachsen kann
- **Mächtigkeit:** hoher Funktionsumfang == mächtige Anfragesprache, Index-Strukturen etc.
- **Geschwindigkeit/Skalierbarkeit:** Konzentration auf die eigentlichen Datenbankaufgaben, Architekturnähe zu den Key-Value-Stores, Sharding Konzept
- **Einfache Benutzbarkeit:** Syntax von JavaScript



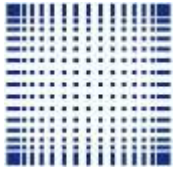
- **Datenbank** ist der physischer Container für Collections. Jede Datenbank wird in einer eigenen Datei abgelegt. Jeder MongoDB Server arbeitet üblicherweise auf mehreren Datenbanken.
- **Collection** ist eine Gruppe von MongoDB Dokumenten. Collections sind genau einer Datenbank zugeordnet, geben aber den Dokumenten kein Schema vor. Normalerweise sollten aber alle Dokumente in einer Collection zum gleichen Themenbereich gehören.
- **Dokument** ist eine Menge von Schlüssel-Wert-Paaren. Es gibt keine Strukturvorgabe, selbst identische Schlüssel können in unterschiedlichen Dokumenten unterschiedliche Datentypen liefern.



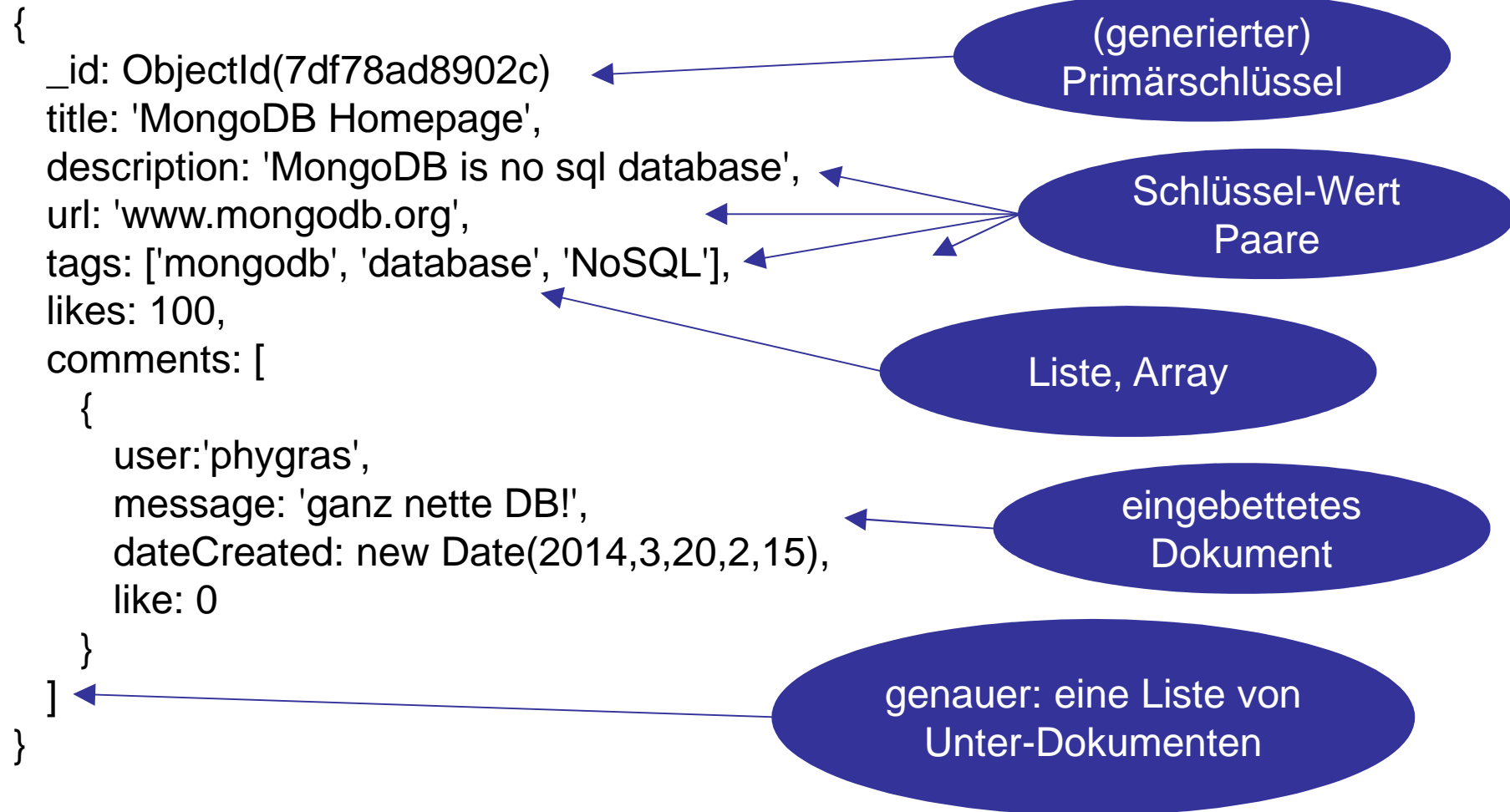
Entsprechungen

| RDBMS | MongoDB |
|-----------------------------------|--------------------------------------|
| Datenbank | Datenbank |
| Tabelle | Collection, Sammlung |
| Datensatz, Tupel, Objekt, Entität | Dokument |
| Spalte, Attribut | Feld, Schlüssel-Wert-Paar |
| Join (von Tabellen) | Eingebettete (embedded) Dokumente |
| Primärschlüssel | _id Schlüssel-Wert-Paar |

- Im Unterschied zu reinen Key-Value-Stores werden die Schlüssel-Wert-Paare also in Dokumenten gebündelt, die im Gegensatz zu den OODB keinerlei Strukturvorgabe besitzen.



Beispieldokument - 1





JavaScript Object Notation (JSON)

- Dokumente werden mit { } geklammert.
- Listen werden mit [] geklammert, einzelne Listenelemente werden mit Komma voneinander getrennt.
- Der Primärschlüssel (_id) ist eine 12 Byte Zahl, hexadezimal notiert, die für jedes Dokument individuell ist.
 - ◆ Ein Wert kann übergeben werden, muss aber nicht.
 - ◆ MongoDB kann diesen Wert selbst generieren:
 - 4 Bytes Timestamp,
 - 3 Bytes Kennung dieses Computers,
 - 2 Bytes Prozesskennung des MongoDB Serverprozeß und
 - 3 Bytes autoinkrementeller Wert.



Referenzielle Integrität

- Fremdschlüsselbeziehungen gibt es nicht.
- Stattdessen: Einbettung oder Verlinkung auf Dokumentenebene.
- Ein Link ist dabei ein Verweis auf ein anderes Dokument,
 - ◆ das getrennt von der Anwendungslogik eingelesen
 - ◆ und verwaltet werden muss.

Dokument, das eine CD beschreibt, enthält Unterdokumente für jeden Track auf der CD, welche dann die Eigenschaften (Tracknr., Titel etc.) enthalten.

→ Die Daten liegen „unnormalisiert“ vor.



Atomarity

- Nur die Änderung eines Dokumentes ist atomar.

Consistency

- Es gibt keine referenzielle Integrität.
- Es gibt keine Constraints.

Isolation

- Keine Lese- oder Schreibsperrern setzbar.



Durability

- Dauerhaftigkeit kann zugesichert werden, zumindest bei Replikation.
- Atomarität und Isolation können begrenzt durch die Änderungsoptionen nachgebildet werden.
- Für MongoDB ist eine Einführung von referenzieller Integrität und Isolation geplant.



Sicherheit

- Voreingestellt bei MongoDB ist es, keinerlei Autorisierungsmechanismen zu verwenden.
- Jeder kann sich erstmal einloggen.
- Das Benutzerkonzept kennt nur zwei Benutzerprofile: Auf einer Datenbank gibt es entweder
 - ◆ nur Lese-Rechte oder
 - ◆ komplette Lese-/Schreibrechte.

Und der Preis für all diese Einschränkungen?

- Die Verwendung von MongoDB bringt einen *immensen* Geschwindigkeitsvorteil!



Geschwindigkeitsvergleich

- Durchgeführt wurde eine SELECT-Anweisung auf einem MySQL- und einem MongoDB-Server bei baugleicher Hardware.
- Die Tabelle umfasste insgesamt 10.000.000 Datensätze.
- Die Abfrage lieferte davon 500.000 Datensätze als Ergebnis.
- Die Ausführungsdauer auf dem MySQL-Server variierte zwischen 533 und 1433 ms.
- Die Ausführungsdauer auf dem MongoDB-Server betrug lediglich maximal 3 ms!
- Dabei wurde die CPU durch den MongoDB-Server nur zu etwa einem Viertel ausgelastet im Vergleich zum MySQL-Server!

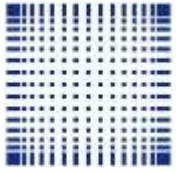


Kriterien für die Verwendung von NoSQL

- Performancesteigerung des Systems?

Besser:

- Viele Schreibzugriffe
- Wenig Lesezugriffe
- Wenig referentielle Integrität
- Es gibt eine Programmebene, die den Zugriff auf die Daten regelt
- Das ER-Diagramm hat Baumcharakter



Beispiel – 1

```
// Hello World für Mongo DB, DBA
```

```
// 2014-12-17, Sven Klaus
```

```
import java.net.UnknownHostException;
```

```
// Die "Abkürzung" in der folgenden Zeile ist rein didaktisch.
```

```
import com.mongodb.*;
```

```
public class MongoDBBeispiel {
```

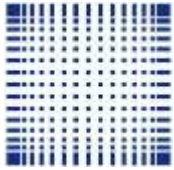
```
    // Beispieldaten anlegen
```

```
    public static BasicDBObject[] createSeedData() {
```

```
        BasicDBObject seventies = new BasicDBObject();
```

```
        seventies.put("decade", "1970s");
```

```
        seventies.put("artist", "Debby Boone");
```



Beispiel – 2

```
seventies.put("song", "You Light Up My Life");  
seventies.put("weeksAtOne", 10);
```

```
BasicDBObject eighties = new BasicDBObject();  
eighties.put("decade", "1980s");  
eighties.put("artist", "Olivia Newton-John");  
eighties.put("song", "Physical");  
eighties.put("weeksAtOne", 10);
```

```
BasicDBObject nineties = new BasicDBObject();  
nineties.put("decade", "1990s");  
nineties.put("artist", "Mariah Carey");  
nineties.put("song", "One Sweet Day");  
nineties.put("weeksAtOne", 16);
```



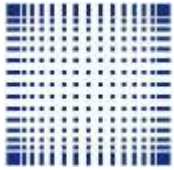
Beispiel – 3

```
final BasicDBObject[] seedData = { seventies, eighties, nineties };

return seedData;
}

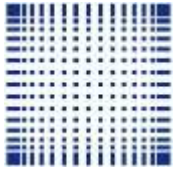
public static void main(String[] args) throws UnknownHostException {
    // Beispieldaten übernehmen
    final BasicDBObject[] seedData = createSeedData();

    // Verbindung herstellen
    // Standard URI format:
    // mongodb://[dbuser:dbpassword@]host:port/dbname
    MongoClientURI uri = new MongoClientURI(
        "mongodb://dba00:blubb@codd.2clever4you.net:27017/dba00");
    MongoClient client = new MongoClient(uri);
    DB db = client.getDB(uri.getDatabase());
}
```



Beispiel – 4

```
// Zuerst die Beispieldaten eintragen.  
// Eine Collection muss, genau wie die Datenbank, *nicht*  
// extra angelegt werden. Dies passiert automatisch.  
DBCollection songs = db.collection("songs");  
  
// Beim Einfügen kann ein einzelnes Dokument, aber auch ein  
// ganzes Array eingefügt werden.  
songs.insert(seedData);  
  
// An dieser Stelle sollten wir fairerweise Boyz II Men für ihren  
// Beitrag an "One Sweet Day" würdigen.  
// Ein Update ist notwendig geworden.  
BasicDBObject updateQuery = new BasicDBObject("song",  
        "One Sweet Day");
```

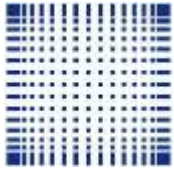


Beispiel – 5

```
songs.update(updateQuery, new BasicDBObject("$set", new  
BasicDBObject("artist", "Mariah Carey ft. Boyz II Men")));
```

```
// Abschließend fragen wir unseren Datenbestand dahingehend  
// ab: Welche Titel belegten 10 oder mehr Wochen den  
// Platz 1 der Billboards?
```

```
BasicDBObject findQuery = new  
BasicDBObject("weeksAtOne", new BasicDBObject("$gte", 10));  
BasicDBObject orderBy = new BasicDBObject("decade", 1);  
DBCursor docs = songs.find(findQuery).sort(orderBy);  
while (docs.hasNext()) {  
    DBObject doc = docs.next();  
    System.out.println("In the " + doc.get("decade") + ", "  
        + doc.get("song") + " by " +  
        doc.get("artist") + " topped the charts for " +  
        doc.get("weeksAtOne") +  
        " straight weeks.");  
}
```



Beispiel – 6

```
// Neu: Der Cursor sollte aus Performancegründen ebenfalls
```

```
// geschlossen werden
```

```
docs.close();
```

```
// Da dies ein Beispiel ist: Aufräumen!
```

```
// Die Collection songs wird wieder gelöscht.
```

```
songs.drop();
```

```
// Das Schließen der Verbindung an sich
```

```
// nur am Ende der Applikation notwendig
```

```
client.close();
```

```
}
```

```
}
```




Wie soll man damit programmieren?

- Umdenken!
- Spätestens in der OOD erfolgt die Trennung in:
 - ◆ (Laufzeit-) Objekte, deren Werte nur zur Laufzeit *dieses* Prozesses benötigt werden und
 - ◆ **Entitäts-Objekte**, deren Werte über die Lebensdauer *dieses* Prozesses erhalten bleiben müssen
- Idee: Die Entitäts-Objekte persistieren sich selbst!
- Hint: Nutzen Sie vorhandene Konzepte, Mechanismen wie das Interface Serializable in Java.



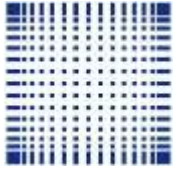
Selbstopersistierende Objekte – 1

- Jeder Konstruktor Aufruf löst ein INSERT Kommando mit den entsprechenden Attributen aus. Bereits vorher gespeicherte Objekte werden in die Anwendung mit einem Factory Pattern geladen.
- Jeder Getter Aufruf ist ein SELECT Kommando. Dies gilt nur für Anwendungen mit konkurrierenden Schreibzugriffen.
- Jeder Setter Aufruf ist ein UPDATE Kommando.
- Bei konkurrierenden Schreibzugriffen, muss der Setter zuerst das betreffende Dokument für den Schreibzugriff durch sich sperren, z.B. durch ein Tag mit der eigenen Prozessnummer. Dieses Tag muss der Setter am Ende wieder löschen.



Selbstpersistierende Objekte – 2

- Referentielle Integrität wird als Programmlogik in den Settern implementiert.
- Genau gegen die Anforderungen prüfen!
- 1:1 und 1:n Beziehungen werden (sofern es die Performance erlaubt) als eingebettete Dokumente umgesetzt.
- m:n Beziehungen werden als Dokumentverlinkung umgesetzt.
- Abwandlungen sind möglich z.B. bei Einzelplatz- oder Admin-Szenarios.



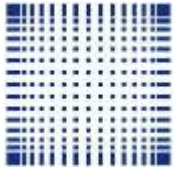
Welche Fragen gibt es?



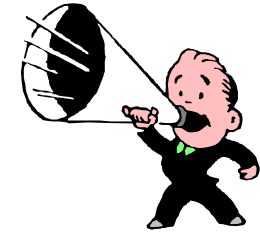
Zum Nachdenken:

*Wieso hat der NoSQL Ansatz
einen größeren Erfolg als die
rein objektorientierten Datenmodelle?*

*In welchen Szenarien spielt es
keine Rolle, wenn Fehler beim
Schreiben auftreten können?*



JETZT



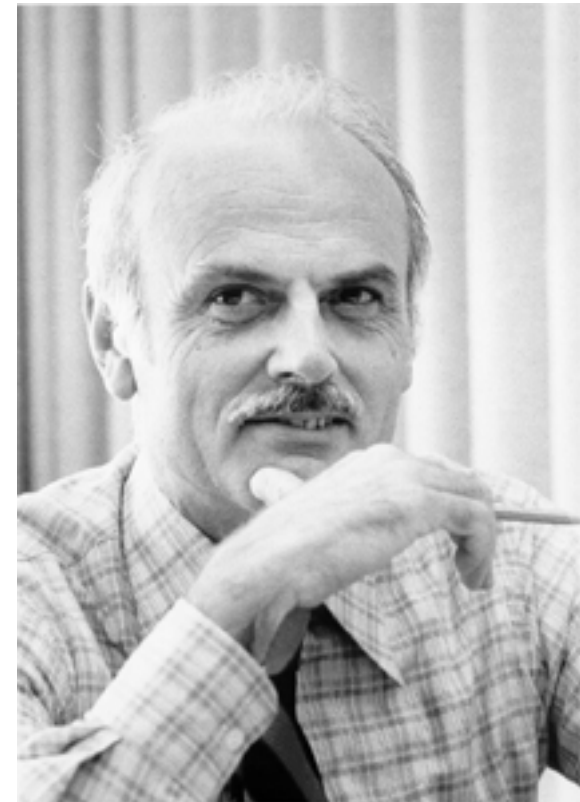
Addendum

***Mal zusammengefasst:
Das Relationale Datenmodell***



Relationales Datenmodell – 1

- **Edgar Frank "Ted" Codd**
(1923 – 2003), IBM, 1970
- Heute immer noch am weitesten verbreitet!
- Daten, aber auch ihre Beziehungen werden in Relationen gespeichert
- Relation = Tabelle
- Feste Anzahl Spalten
- Jede Tabellenzeile ist ein Datensatz





Relationales Datenmodell – 2

- Reihenfolge der Zeilen ist **nicht** festgelegt
- Reihenfolge der Spalten ist **nicht** festgelegt
- Datensätze werden über die Werte der Attribute identifiziert
- Minimale identifizierende Attributmenge ist ein Schlüssel
- Beziehungen sind ebenfalls Tabellen, die Schlüssel der an der Beziehung beteiligten Datensätze speichern
- Operationen im Relationalen Modell: relationale Algebra, Relationenkalkül



- Ergebnisse von Operationen sind wieder Tabellen

Vorteile 👍

- (spätere) leichte Änderungen an der Struktur (Tabellen entfernen, hinzufügen)
- Logische Sicht auf die Daten steht im Vordergrund. Anwender braucht für Abfragen keine Kenntnisse der Speicherorganisation, sondern nur Tabellen- und Spaltennamen



Nachteile 👎

- Nur atomare Werte können gespeichert werden
- Übliche Abfragen über mehrere Tabellen kosten Ausführungszeit
- Manche Informationen können nur mit Redundanz gespeichert werden
- Standardsprache: SQL



Schlüssel im relationalen Datenmodell – 1

- Aus der Mengendefinition folgt, dass jedes Tupel eindeutig über ein oder mehrere (ggf. auch alle) Attribute eindeutig identifizierbar sein.
- Die Menge der Attribute, die ein Tupel eindeutig identifizieren heißen **Schlüsselkandidaten**
- An Schlüssel wird die **Minimalitätsforderung** gestellt, d.h. ein Schlüssel muss in Bezug auf die Mächtigkeit der umfassenden Attributmenge so kurz wie möglich sein



Tabelle Kunde:

KUNDE(KuNr, Name, PLZ, Wohnort, Strasse)

Als Schlüssel kommen in Frage:

- KuNr (die Kundennummer) oder
- Name, PLZ, Strasse oder
- Name, Wohnort, Strasse
(sofern nicht zwei Kunden *exakt gleichen Namens in exakt der gleichen Strasse im gleichen Wohnort* wohnen)



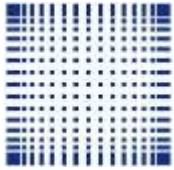
Schlüssel im relationalen Datenmodell – 2

- Ein möglichst kurzer Schlüssel wird als **Primärschlüssel** ausgezeichnet (im Beispiel KuNr)
- Ggf. muss ein zusätzliches Attribut der Tabelle hinzugefügt werden zur eindeutigen Identifikation der Datensätze. Dieses Attribut wird automatisch Primärschlüssel
- ein derartiges zusätzliches Attribut wird in der Literatur auch als **Identifikations-Attribut** bezeichnet, häufig ist es eine ID-Nummer o.ä.
- Jede Tabelle kann nur einen Primärschlüssel besitzen



Schlüssel im relationalen Datenmodell – 3

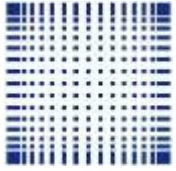
- Ein Attribut A_i (bzw. eine Menge von Attributen) in einer Tabelle R_1 , welches eine Beziehung zu einem Schlüsselattribut einer anderen Relation R_2 herstellt, heißt **Fremdschlüssel**
- Beispielsweise wird eine Tabelle Aufträge in jedem Datensatz die Kundennummer (KuNr) des auslösenden Kunden vermerken. Diese KuNr identifiziert den Kunden eindeutig. In der Tabelle Aufträge ist die KuNr ein Fremdschlüssel = "fremder" Schlüssel in einer anderen Tabelle



- Häufig sollen die Datensätze einer Tabelle in einer anderen Reihenfolge durchlaufen werden, als sie erfasst wurden (z.B. nach Postleitzahlen)
- Dazu muss die gesamte Tabelle vorher sortiert werden (nach dem Kriterium, Attribut Postleitzahl)
- Umso mehr Datensätze in der Tabelle enthalten sind, desto länger dauert dieser Vorgang
- Zur Beschleunigung können zusätzlich zum Primärschlüssel *weitere Indizes* (nach einem bestimmten Kriterium sortierte Zugriffslisten) angelegt werden
- Diese heißen auch **Sekundärindizes**
- Dieser Geschwindigkeitsvorteil wird mit einem Geschwindigkeitsnachteil bei der Erfassung der Datensätze erkauft, da für jeden Datensatz alle Indizes aktualisiert werden müssen
- Viele Systeme erlauben eine (Gesamt-)Aktualisierung aller Indizes nach Abschluss der Eingabe mehrerer Datensätze



Welche Fragen gibt es?



Zum Nachdenken:

Warum „hält“ sich das relationale Datenmodell trotz „neuerer“ Datenmodelle so „hartnäckig“ im industriellen Einsatz?



Vielen Dank für Ihre Aufmerksamkeit!

