Kapitel 4: Inhaltsübersicht

- In Kapitel 3 wurden Abfragen auf Basis einer formalen Sprache spezifiziert, die auch die theoretische Grundlage für die Datenbankabfragesprache SQL (Structured Query Language) bilden.
- Im Kapitel 4 wird SQL, ein im DB-Umfeld Defacto-Standard vorgestellt.
- Nach einem kurzen geschichtlichen Überblick, werden Systax und Anwendungsbeispiele detailliert vorgestellt.
- Das Kapitel 4 schließt mit einer praxisorientierten Rechnerübung im Kontext des Übunsbeispiels Duale Hochschule.

Historisches zu SQL (1)

- Nachdem das relationale Modell 1970 von Codd eingeführt wurde, diente in der ersten Phase die bereits behandelte relationale Algebra und das Relationenkalkül als Grundlage für Abfragen.

- Auf dieser Basis wurde 1974 von einem IBM-Forschungsprojekt eine erste Datenbanksprache **SEQUEL** (Structured English Query Language) entwickelt und im Rahmen des Projektes System R (das war der erste Prototyp eines relationalen Datenbanksystems) 1976 zu **SEQUEL2** erweitert.

- In den ersten kommerziellen DB-Systemen Anfang der 80er Jahre (übrigens von Oracle) wurde eine Untermenge dieser Sprache implementiert und SQL genannt.

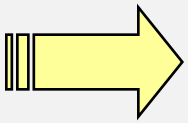
Historisches zu SQL (2)

- SQL wurde dann in den Jahren 1982 bis 1986 von der ANSI genormt. Diese Normierung wird heute mit **SQL1** bezeichnet.
- 1989 erfolgte eine zweite Normierung (**SQL-89**), bei der zusätzlich eine neue Teilsprache zur Integritätssicherung eingeführt wurde.
- 1992 erschien mit **SQL-92** (in der Praxis als **SQL2** bezeichnet) eine dritte Normierung, bei der der Sprachumfang deutlich erweitert wurde.
- **SQL3** ist das letzte vollständig abgeschlossene Normungsprojekt der ANSI, wobei sich die Veröffentlichung von 1999 bis ins Jahr 2003 hingezogen hat. Dieser Standard wird heute von nahezu allen am Markt verfügbaren DB-Systemen abgebildet.
- **SQL-2006** definiert Erweiterungen im Zusammenhang mit XML; im Jahr 2008 erfolgte eine umfangreiche Revision von SQL-2006.

Zusammenhang zwischen SQL und relationaler Algebra

- Der folgende Ausdruck der relationalen Algebra gibt eine Relation im DH-Umfeld an, aus der Daten durch eine Selektion mit folgender Projektion extrahiert werden sollen:

$$\pi_{\text{PersNr, Name}} (\sigma_{\text{Rang = 'C4'}} (\text{Professoren}))$$



1571	**Keppler**
1643	**Newton**
1901	**Heisenberg**

- Das Äquivalent auf Basis von SQL lautet

SELECT *PersNr, Name*
FROM *Professoren*
WHERE *Rang = 'C4'*;

- Hinweis: Die Beispiele in diesem Kapitel beziehen sich auf unser E/R-Modell der Dualen Hochschule und sind überwiegend Kem-09, Seite 106 bis 134 entnommen.

Vereinbarungen zur Notation der SQL-Syntax (1)

- **GROSSBUCHSTABEN und FETT**
  Schlüsselwörter, Funktionen und Operatoren von SQL, wobei viele RDBMS nicht case sensitiv sind.
- *klein und kursiv*
  Bezeichnungen, die der Anwender selbst bestimmen kann und die nicht weiter definiert werden, z.B. Tabellennamen
- x ::= y
  Definitionsanweisung mit der Bedeutung, x wird durch y definiert.
- ,…
  Wiederholung, d.h. der zuvor geschriebene Ausdruck kann, muss aber nicht beliebig oft wiederholt werden.

Vereinbarungen zur Notation der SQL-Syntax (2)

- **[ ]**
  Option, d.h. der in der eckigen Klammer enthaltene Ausdruck kann, muss aber nicht verwendet werden.
- **{ }**
  Gruppierung, d.h. der in der geschweiften Klammer enthaltene Ausdruck muss verwendet werden.
- **|**
  Alternative, d.h. entweder der Ausdruck links oder der Ausdruck rechts vom Symbol
- Beispiele
  - [A | B] bedeutet entweder A oder B oder keins von beiden
  - {A | B} bedeutet entweder A oder B

Einfache Datentypen in SQL (1)

- Zahlenformate können wie folgt näher spezifiziert werden:
  - **INTEGER** (auch **INT**): Zahlen ohne Nachkommastellen mit fester Länge und einem vom jeweiligen DBMS abhängigen Wertebereich.
  - **SMALLINT**: In einigen DBMS verfügbare Darstellung von ganzen Zahlen mit stark eingeschränktem Wertebereich. Analog **BIGINT**
  - **FLOAT** (auch **REAL**): Gleitkommazahl; teilweise auch als **DOUBLE** mit größerem Wertebereich implementiert.
  - **NUMBER(n)**: Numerisches Feld mit einer variablen Länge von maximal n Stellen
  - **NUMBER (n,m)** definiert einen numerischen Datentyp mit der Gesamtlänge n (ohne Dezimalpunkt), wovon allerdings m Stellen als Nachkommastellen reserviert sind.

Einfache Datentypen in SQL (2)

- Alphanumerische Zeichenketten
  - **CHAR[(n)]:** Zeichenkette (String) mit einer festen Länge n d.h. nicht benötigte Zeichen werden automatisch mit Leerzeichen aufgefüllt. Anmerkung: Falls n weggelassen wird, beträgt die Länge 1
  - **VARCHAR(n):** Zeichenkette mit variabler Länge, wobei n die maximale Länge des Strings angibt. Kürzere Zeichenketten sind kürzer!
- Datumsformate
  - **DATE**: Datumsfeld
  - **TIME**: Uhrzeiten
  - **TIMESTAMP**: Datumsfeld mit Uhrzeit
- Besondere Datentypen sind z.B.
  - **BLOB** (von binary large object) für sehr große binäre Daten
  - **BOOLEAN** zur Darstellung von Wahrheitswerten

(Nicht vollständiger) Vergleich der wichtigsten Datentypen

Datentyp	Beschreibung	ANSI	Oracle	MySQL
SMALLINT	Ganze Zahl zwischen -32.767 und 32767	x	x	x
BIGINT	Ganze Zahl zwischen -2^{63} und $2^{63}-1$	x		x
INTEGER INT	Ganze Zahl zwischen -2.147.483.647 und 2.147.483.647	x	x	x
FLOAT REAL	Gleitkommazahl zwischen -4.10E79 und 7.210E75	x	x	x
DOUBLE	Gleitkommazahl mit 8 Byte	x	x	x
DECIMAL(p,q) NUMERIC(p,q)	Festkommazahl mit p Ziffern und q Nachkommastellen, p zwischen 1 und 38, q zwischen -38 und 38	x	x	x
CHARACTER(n) CHAR(n)	Wort mit fester Zeichenlänge von n Zeichen, n ≤ 2000 Zeichen	x	x	x
VARCHAR(n) VARCHAR2(n)	Wort mit variabler Zeichenlänge von n[5] Zeichen	x	x	x
RAW	Binärer Datentyp bis zu 2000 Byte		x	
LONG RAW	Binärer Datentyp bis zu 2 GByte (z.B. für Fotos, Grafiken und andere Binärdaten)		x	
DATE	Datumsfelder, teilweise mit Uhrzeit	x	x	x
DATETIME	Datumsfelder mit Uhrzeit			x
TIME	Uhrzeiten	x		x
TIMESTAMP	Zeitstempel mit Datum und Uhrzeit	x	x	x
ROWID	Hexadezimale Adresse des Datensatzes		x	(x)[6]
INTERVAL	Zeitintervalle	x		x
BLOB	Für Binärdaten wie Fotos, Grafiken und Ton	x	x	x
CLOB	Für lange Textobjekte		x	
BFILE	Zeiger auf eine Datei, in der ein Text oder Bildobjekt gespeichert ist		x	

Quelle:Fae-07, Seite 197

Definitionen der Basistabellen (1)

- Zunächst muss eine (leere) Datenbank generiert werden:
  CREATE SCHEMA *schemaname*
- Basierend auf einem Relationenschema, welches z.B. aus einem E/R-Modell abgeleitete wurde, werden im nächsten Schritt die einzelnen Tabellen erzeugt.
- Die entsprechende SQL-Anweisung hat folgende Syntax:
  CREATE TABLE *tabellenname*
  (Spaltenname Datentyp [Spaltenbedingung [,...]]
  | Tabellenbedingung [,...])
  - Anmerkung: Spaltenbedingungen gelten immer für einzelne Attribute, während sich Tabellenbedingungen auf Attributkombinationen beziehen oder Merkmale der referenziellen Integrität festlegen.

Definitionen der Basistabellen (2)

- Einfaches Anwendungsbeispiel:

```
CREATE TABLE Professoren
        (PersNr INTEGER PRIMARY KEY,
        Name VARCHAR (20) NOT NULL,
        Vorname VARCHAR (20),
        Rang CHARACTER (2));
```

- Mit der optionalen Einschränkung "not null" wird erzwungen, dass alle in der Tabelle eingetragenen Tupel für dieses Attribut einen definierten Wert haben. Bezogen auf das Beispiel ist es also nicht möglich, Professoren ohne Personalnummer und/oder ohne Namen in die Tabelle einzutragen.
- Die Angabe "not null" ist eine sogenannte Integritätsbedingung und gilt (natürlich) impliziert auch für alle Schlüsselattribute.

Spaltenbedingungen

- Spaltenbedingung::=
  NOT NULL
  | { **PRIMARY KEY** | **UNIQUE** }
  | **CHECK (***Bedingung***)**

- **NOT NULL**: Erzwingt eine Dateneingabe für dieses Attribut

- **PRIMARY KEY**: Bestimmt eine oder mehrere Spalten als Primärschlüssel

- **UNIQUE**: Verhindert dass in dieser Spalte Duplikate auftreten. In einigen Systemen wird für solche Attribute automatisch ein Index generiert.

- **CHECK:** Legt eine Bedingung fest, die jeder Attributwert erfüllen muss und erlaubt daher eine Prüfung hinsichtlich des Wertebereichs.

Tabellenbedingungen

- Tabellenbedingung::=
  {PRIMARY KEY | UNIQUE} (Spaltenname [, …])
  FOREIGN KEY (Spaltenname [, …])
  REFERENCES Tabellenname (Spaltenname [, …])
  [,ON DELETE {RESTRICT | NO ACTION | CASCADE | SET NULL | SET DEFAULT}]
  [,ON UPDATE {RESTRICT | NO ACTION | CASCADE | SET NULL | SET DEFAULT}]

- Primary key und unique gelten analog der Definition für Spaltenbedingungen, aber eben für mehrere Attribute.

- Ausführliche Erläuterungen zum Foreign Key und den damit verbundenen Integritätsbedingungen folgen in Kapitel 5.

Schemaveränderungen

- Um nachträglich ein Attribut einzufügen (z.B. die zunächst vergessene Zimmernummer), wird folgender Befehl benötigt:

  ALTER TABLE *Professoren*
  ADD (*ZimmerNr* **INTEGER**);

- Wenn sich rausstellt, dass einige Professoren Namen mit mehr als 20 Zeichen haben, wird die Definition nachtäglich modifiziert:

  ALTER TABLE *Professoren*
  MODIFY (*Name* **VARCHAR** (30) **NOT NULL**);

- Um nicht mehr benötigte Spalten zu entfernen, formuliert man:

  ALTER TABLE *Professoren* **DROP** *Raum*;

- Befehl, um die Relation aus der Datenbank zu löschen:

  DROP TABLE *Tabellenname* { **RESTRICT | CASCADE** };

Änderungs-Operationen (1)

- Um die soeben angelegten Tabellen mit Daten zu füllen, sind Änderungs-Operationen erforderlich. Man unterscheidet
  - **Einfügen** von Tupeln in Basistabellen
  - **Ändern** von Tupeln in Basistabellen
  - **Löschen** von Tupeln aus Basistabellen
- Diese Operationen sind jeweils möglich als
  - **Eintupel-Operationen** (etwa das Erfassen einer neuen Vorlesung)
  - **Mehrtupel-Operationen** (z.B. Gehaltserhöhung für alle um 3,5%)
- In kommerziellen Systemen werden diese Befehle normalerweise menü- bzw. fromulargestützt angeboten, was den Komfort für die Anwender signifikant erhöht.

Änderungs-Operationen (2)

- Zum Einfügen von Tupeln in Basistabellen dient der Befehl insert. Hierbei gibt es die beiden zuvor angesprochenen Möglichkeiten: Das Einfügen von konstanten Tupeln sowie das Einfügen von (aus anderen Relationen) berechneten Tupeln.

- Syntax für den ersten Fall:

  INSERT
  INTO b*asistabellenname* [(Spaltenname[,spaltenname]…]
  VALUES (konstante [,konstante]…);

- Anwendungsbeispiele:
  - **INSERT INTO** *Professoren*
    VALUES (1473,'Kopernikus','C3',121);
  - **INSERT INTO** *Vorlesungen* (*Titel, VorlNr*)
    VALUES (*'Das heliozentrische Weltbild',2001);*

Änderungs-Operationen (3)

- Etwas komplizierter, aber auch wesentlich effizienter, ist die zweite Variante, die folgende Syntax hat:

  INSERT
  INTO B*asistabellenname* [(*Spaltenname*[,*Spaltenname*]…]
  SQL-Anweisung;

- Praxisbeispiel: Angenommen Professor Heisenberg ist der Meinung, dass alle Studenten unbedingt seine Vorlesung über die Unschärferelation hören sollten, so kann er dies – zumindest auf Datenbankebene – relativ leicht bewirken:

Änderungs-Operationen (4)

- Syntaktisch wird die update-Anweisung wie folgt beschrieben:
  UPDATE *Basistabellenname*
  SET *Spaltenname* = <Ausdruck> [,*Spaltenname* = <Ausdruck>]…
  [**WHERE** (*Bedingung*)];

- Damit werden in allen Tupeln der Basistabelle, welche die Bedingung erfüllen, die Attributwerte wie angegeben ersetzt.

- Um beispielsweise alle Studenten beim Semesterwechsel "upzudaten", formuliert man
  UPDATE *Studenten*
  SET *Semester = Semester + 1;*

Änderungs-Operationen (5)

- Zum Löschen von Tupeln wird die Delete-Anweisung verwendet, die folgende Syntax hat:

  DELETE FROM *Basistabellenname*
  [**WHERE** (*Bedingung*)];

- Beispiel sei folgende Anweisung, mit der datenbanktechnisch alle Langzeitstudenten exmatrikuliert werden:

  DELETE FROM *Studenten*
  WHERE *Sem > 8*;

- **Vorsicht**! Wenn keine Bedingung angegeben wird, löscht die Anweisung - absolut folgerichtig - sämtliche Tupel, z.B.

  DELETE FROM *Professoren;*

Der SFW-Block

- Durch den SFW-Block wird die Standardform einer SQL-Anfrage beschrieben. Er besteht aus
  - **SELECT**-Klausel
    - Projektion, welche das Ergebnisschema beschreibt
    - Integration von arithmetischen Operationen und Funktionen.
  - **FROM**-Klausel
    - Spezifikation der verwendeten Relationen (Tabellen),
    - bei Verwendung mehrerer Relationen deren kartesisches Produkt
    - Gegebenenfalls erforderliche Umbenennungen.
  - **WHERE**-Klausel
    - Festlegung der Selektionsbedingungen,
    - sofern erforderlich, Spezifikation von Verbundbedingungen, um aus dem kartesischen Produkt einen Join zu machen,
    - kann wiederum (geschachtelte) SQL-Anfragen enthalten.

Syntax des SELECT-Befehls

SELECT [DISTINCT] { ***** | *Attributliste* | *arithmetischer Ausdruck* | *Aggregat-funktion* }
FROM *Basistabellenname [, Basistabellenname ...]*
[WHERE *Bedingung*]
[[GROUP BY *Attributliste*]
[HAVING *Bedingung*]]
[ORDER BY *Attributname* **[ASC** | **DESC]** ,
Attributname **[ASC** | **DESC]];**

- **SELECT :** Auswahl von Attributen (Projektion);Operationen, Funktionen
- **FROM** : Angabe der Tabellen, die in der Abfrage benötigt werden
- **WHERE :** Auswahl von Zeilen (Selektion)
- **GROUP BY :** Zusammenfassung von Zeilen zu Gruppen
- **HAVING :** Auswahl von bestimmten Gruppen
- **ORDER BY :** Sortierung der Ergebnistabelle anhand bestimmter Spalten

Die SELECT-Klausel (1)

- Ausgabe aller Inhalte einer ganzen Tabelle:
  SELECT *
  FROM *Professoren;*

- Beispiel für Projektion mit einfacher Bedingung:
  SELECT *PersNr, Name*
  FROM *Professoren*
  WHERE *Rang = 'C4';*

  ermittelt eine Liste mit Personalnummer und Name der Professoren, die den Rang C4 haben.

- Beispiel für die Unterdrückung von Duplikaten:
  SELECT DISTINCT *Rang*
  FROM *Professoren;*

Die SELECT-Klausel (2)

- Beispiel für die Verwendung von arithmetischen Ausdrücken
  SELECT *ArtNr, ArtBezeichnung, Nettopreis x 1.19*
  FROM *Artikel;*

  erzeugt eine Bruttopreisliste für den Verkauf an Endkunden.

- Beispiel für die Sortierung der Ergebnisliste
  SELECT *PersNr, Name, Rang, Gehalt*
  FROM *Professoren*
  ORDER BY *Rang* **DESC**, *Name* **ASC**;

  ermittelt eine sortierte Liste aller Professoren mit Personalnummer, Name und Rang, wobei die Sortierung absteigend nach Rang und bei gleichem Rang aufsteigend nach dem Namen erfolgt.

Die SELECT-Klausel (3) / Aggregatfunktionen

- Im Gegensatz zu skalaren Operationen arbeiten Aggregat-funktionen "tupelübergreifend" und berechnen Eigenschaften von ganzen Tupelmengen, sogenannte Aggregate.

- In SQL gibt es folgende Aggregatfunktionen:

COUNT	berechnet die Anzahl der Werte einer Spalte oder alternativ (bei count (*)) die Anzahl der Tupel einer Relation.
SUM	berechnet die Summe der Werte einer Spalte (natürlich nur bei numerischen Wertebereichen der Attribute)
AVG	berechnet den arithmetischen Mittelwert der Werte einer Spalte (ebenfalls nur bei numerischen Wertebereichen)
MAX	ergibt den größten Wert einer Spalte
MIN	ergibt den kleinsten Wert einer Spalte

Die SELECT-Klausel (4) / Aggregatfunktionen

- Zulässige Argumente der Aggregatfunktionen sind
  - ein Attribut der durch die FROM-Klausel spezifizierten Relation(en)
  - ein gültiger skalarer Ausdruck
  - im Spezialfall der Count-Funktion das Symbol *
- Abhängig davon wird die gewählte Funktion angewendet auf
  - die Menge der angegeben Attributwerte
  - die Menge der Ergebniswerte eines skalaren Ausdrucks
  - die Menge aller Tupel der Relation
- Anmerkungen
  - Außer bei COUNT (*) können vor dem Argument noch DISTINCT und ALL (Default-Einstellung) verwendet werden.
  - Nullwerte werden grundsätzlich vorab eliminiert (nicht bei COUNT (*))

Die SELECT-Klausel (5) / Aggregatfunktionen

- Die Anzahl der Professoren ermittelt man mit

  SELECT COUNT (*)
  FROM *Professoren*;

- Die Anzahl der prüfenden Professoren ermittelt man mit

  SELECT COUNT (**DISTINCT** *PerNr*)
  FROM *prüfen*;

- Die durchschnittliche Semesterzahl aller Studenten:

  SELECT AVG (**ALL** *Semester*)
  FROM *Studenten*;

- Die Durchschnittsnote aller Studenten in einem Fach:

  SELECT AVG (*Note*)
  FROM *prüfen, Vorlesungen*
  WHERE *prüfen.VorlNr = Vorlesungen.VorlNr* **AND**
  Vorlesungen.Titel = 'Die Unschärferelation';

Die FROM-Klausel

- Die FROM-Klausel ist die Basis für die Abfragebearbeitung, da hier die im Statement verwendeten Basistabellen bzw. Views spezifiziert werden.

- Beispielsweise wird mit dem Statement

  SELECT *
  FROM Professoren, Assistenten;

  das kartesische Produkt der beiden Basistabellen gebildet.

- Falls gewünscht, können hier Umbenennungen erfolgen, z.B.

  SELECT *Name, Titel*
  FROM *Professoren p, Assistenten a*
  WHERE *……*

Die WHERE-Klausel (1)

- Mit der WHERE-Klausel werden aus denen bei „FROM“ angegebenen Relationen diejenigen Tupel **selektiert**, die den hier formulierten Bedingungen genügen.

- Bedingungen könne beispielsweise sein
  - **Verbundbedingung** (Join), z.B. relation1.attribut1 = relation2.attribut2
  - **Vergleich** eines Attributs mit einer Konstanten
  - **Prüfung**, ob ein Attribut in einem bestimmten Intervall liegt
  - **Vergleich zweier Attribute** mit kompatiblen Wertebereichen
  - **Teilstringsuche** in Strings in der Form attribut LIKE zeichenkette
  - **Null-Selektion** (attribut IS NULL bzw. IS NOT NULL)
  - Quantifizierende Bedingungen mit einem **Quantor**, z.B: EXISTS.

Die WHERE-Klausel (2)

Symbol	Bezeichnung
=,<,<=,>,>=,<>	arithmetische Vergleichsoperatoren
[NOT] LIKE	Teilstringsuche (Ähnlichkeitsoperator)
[NOT] BETWEEN	Intervallsuche
ALL, ANY, SOME	Quantoren (Auswahloperatoren)
IS [NOT] NULL	Nullbedingung
[NOT] IN	Mengenoperator
[NOT] EXISTS	Existenzoperator
UNIQUE	Eindeutigkeitsoperator

- Verknüpfung mehrerer Operatoren mit AND und OR möglich

Die WHERE-Klausel (3)

- **SELECT** *PerNr, Name*
  FROM *Professoren*
  WHERE *Name* **LIKE** 'K%';

  Liefert eine Liste aller Professoren, deren Name mit K anfängt.

- **SELECT** *ArtNr, ArtText, Nettopreis x 1.19*
  FROM *Artikel*
  WHERE *ArtNr* **BETWEEN** *1000* **AND** *1999*
  ORDER BY *Nettopreis*;

  Gibt eine nach dem Preis sortierte Verkaufsliste mit Artikelnummer, Bezeichnung und Bruttopreis der Artikel aus, deren Artikelnummern im Bereich von 1000 bis 1999 liegen.

Die WHERE-Klausel (4)

- **SELECT** *ArtNr, ArtText, Gewicht*
  FROM *Artikel*
  WHERE *Gewicht* **IN** (*5,10,25*) **AND** *ArtText like '%kleber%'*;

  Erzeugt eine Liste aller Artikel, in deren Bezeichnung das Wort „kleber“ vorkommt, sofern das Gewicht 5, 10 oder 25 kg beträgt. Ergibt u.a. alle Fliesenkleber in den entsprechenden Verpackungsmengeneinheiten.

- Angenommen man möchte diejenigen Studenten auflisten, deren Wohnort noch nicht in der Datenbank erfasst wurde, so formuliert man

  SELECT *
  FROM *Studenten*
  WHERE *Wohnort* **IS NULL**

Vorsicht bei Nullwerten!

- Generell sollte man durch geeignete Integritätsbedingungen und Konsitenzprüfungen Nullwerte möglichst vermeiden. Manchmal sind jedoch bestimmte Merkmale zu dem Zeitpunkt t (noch) nicht bekannt, z.B. das Semester bei einem Studenten, der von einer anderen Hochschule wechselt.

- Es kann in solchen Fällen vorkommen, dass Nullwerte das Ergebnis von Abfragen beeinflussen, obwohl man dies nicht vermuten würde, z.B.

```
SELECT COUNT (*)
FROM Studenten
WHERE Semester < 13 OR Semester >= 13;
```

- Merke: Bei arithmetischen Ausdrücken ist das Ergebnis grundsätzlich NULL, wenn einer der Operanden NULL ist.

Auswertungsregeln bei Nullwerten

- SQL hat eine dreiwertige Logik, die neben „true" und „false" auch den Wert „unknown" kennt. Logische Ausdrücke werden dabei gem. folgender Tabelle berechnet:

not	
true	false
unknown	unknown
false	true

and	true	unknown	false
true	true	unknown	false
unknown	unknown	unknown	false
false	false	false	false

or	true	unknown	false
true	true	true	true
unknown	true	unknown	unknown
false	true	unknown	false

entnommen aus Kem-09, Seite 125

  - In der WHERE-Klausel werden nur Tupel berücksichtigt, für die die Bedingung „TRUE" gilt, d.h. Tupel mit "unknown" werden nicht in die Ergebnismenge aufgenommen.
  - Bei einer Gruppierung dagegen wird "null" als ein eigenständiger Wert betrachtet und daher in einer separaten Gruppe zusammengefasst.

Abfragen über mehrere Tabellen (1)

- Um festzustellen, welche Assistenten (namentlich) für welche Professoren arbeiten, müssen Informationen aus zwei Tabellen zusammengeführt werden:

```
SELECT    Assitenten.Name , Professoren.Name
FROM      Assistenten, Professoren
WHERE     Boss = Professoren.Name;
```

- Um festzustellen, welche Studenten welche Vorlesungen hören, muss die Abfrage 3 Tabellen verknüpfen („joinen“) :

```
SELECT    Name, Titel
FROM      Studenten, hören, Vorlesungen
WHERE     Studenten.MatrNr = hören.MatrNr AND
          hören.VorlNr = Vorlesung.VorlNr;
```

Abfragen über mehrere Tabellen (2)

- Alternativ können auch neue Variablen eingeführt werden:

```
SELECT  Name, Titel
FROM    Studenten s, hören h, Vorlesungen v
WHERE   s.MatrNr = h.MatrNr AND
        h.VorlNr = v.VorlNr;
```

- Und nochmals zur Erinnerung

$$\pi_{\text{Name, Titel}} (\sigma_{\text{Studenten.MatrNr=hören.MatrNr} \wedge \text{hören.VorlNr=Vorlesung.VorlNr}} (\text{Studenten} \times \text{hören} \times \text{Vorlesungen}))$$

- Solche Abfragen werden in drei Schritten abgearbeitet:
  - Zunächst wird das Kreuzprodukt der beteiligten Tabellen gebildet.
  - Dann wird jede Zeile dieses Kreuzproduktes auf Erfüllung der Bedingungen geprüft, die in der WHERE-Klausel definiert sind, und gegebenenfalls eliminiert.
  - Anschließend erfolgt eine Projektion auf die in der SELECT-Klausel angegeben Spalten.

Gruppierung (1)

- Mit der GROUP BY-Klausel entfernt man sich etwas von dem normalen Relationenmodell, da dieses Statement zumindest virtuell eine geschachtelte Relation erzeugt. Im Prinzip wird bei einer gegebenen Relation R anhand einer Attributmenge G die Relation "geschachtelt", d.h. für gleiche G-Werte werden die Resttupel zusammengefasst.

- Es soll die Gesamtzahl der von den einzelnen Professoren gelesenen Semesterwochenstunden ermittelt werden:

  SELECT *gelesenVon*, **SUM** (*sws*)
  FROM *Vorlesungen*
  GROUP BY *gelesenVon*

  Man beachte die (elegante) Kombination mit der Aggregatfunktion SUM.

Gruppierung (2)

- Zuerst werden alle Zeilen der Tabelle Vorlesungen, die den gleichen Wert im Attribut gelesenVon haben, zusammengefasst und für jede dieser so entstandenen Gruppen die Summe der Semesterwochenstunden berechnet.

- Sollen bei der Auswertung nur die Professoren berücksichtigt werden, die überwiegend umfangreiche Vorlesungen halten, kommt die optionale HAVING-Klausel zum Einsatz, die sich in der Auswertungsreihenfolge an GROUP BY anschließt.

```
SELECT gelesenVon, SUM (sws)
FROM Vorlesungen
GROUP BY gelesenVon
HAVING AVG (sws) > 4;
```

Gruppierung (3)

- Zur Verdeutlichung des Unterschieds zwischen der WHERE- und der HAVING-Klausel, soll das vorherige Beispiel erweitert werden um:
  - Es sollen nur C4-Professoren betrachtet werden.
  - Zusätzlich ist der Name des Professors auszugeben.

SELECT *gelesenVon, Name,* **SUM** *(sws)*
FROM *Vorlesungen, Professoren*
WHERE *gelesenVon = PersNr* **AND** *Rang = 'C4,*
GROUP BY *gelesenVon, Name*
HAVING AVG *(sws) > 4*;

Gruppierung (4)

- Die Abarbeitung kann man sich wie folgt vorstellen:
  - Zuerst wird das Kreuzprodukt Vorlesungen x Professoren gebildet.
  - Anschließend werden aus dieser Relation die Tupel ausgewählt, die die WHERE-Bedingung erfüllen.
  - Die so entstandene (temporäre) Ergebnismenge wird anhand der gleichen Werte in den beiden Gruppierungsattributen gelesenVon und Name zusammengefasst.
  - Jede einzelne Gruppe wird mittels der Bedingung in der HAVING-Klausel überprüft und, bei Erfüllen der Bedingung, in die Ergebnismenge übernommen.
- Anmerkung: Da in der Ausgaberelation jede Gruppe nur durch ein Tupel repräsentiert wird, können in der SELECT-Klausel nur Attribute oder Aggregatfunktionen vorkommen, nach denen auch gruppiert wurde. Daher ist in der GROUP BY-Klausel das Attribut Name ebenfalls zwingend anzugeben.

Abstrakte Übungsaufgabe zur Gruppierung

- Gegeben sei folgende Relation „Beispieltabelle“:

A	B	C	D	E
1	2	3	4	5
1	2	4	5	7
2	3	3	4	9
2	2	2	2	3
3	3	4	5	6
3	3	6	7	7

und diese SQL-Anweisung. Wie lautet das Ergebnis?

SELECT *A*, **SUM** *(D)*
FROM *Beispieltabelle*
WHERE *E > 4*
GROUP BY *A,B*
HAVING *A < 4* **AND**
SUM(D) < 10 **AND MAX** *(C) = 4*;

Geschachtelte Anfragen (1)

- In SQL können SELECT-Anweisungen auf vielfältige Art geschachtelt werden. Grundsätzlich unterscheidet man dabei Unteranfragen die
  - höchstens ein Tupel
  - eine beliebig große Menge von Tupeln

  an die übergeordnete SELECT-Klausel zurückliefern.

- Im ersten Fall kann die Unteranfrage dort eingesetzt werden, wo ein skalarer Wert erforderlich ist. Z.B. ermittelt folgendes SQL-Statement alle Prüfungen, die exakt durchschnittlich verlaufen sind:

```
SELECT *
FROM prüfen
WHERE Note = (SELECT AVG (Note) FROM prüfen);
```

Geschachtelte Anfragen (2)

- Um beispielsweise die zeitliche Belastung aller Professoren zu ermitteln, formuliert man:

  SELECT *PersNr, Name*, (**SELECT** *SUM* (sws) *AS Belastung*
  FROM *Vorlesungen*
  WHERE *gelesenVon = PersNr*)
  FROM *Professoren*;

- Interessant an diesem Beispiel ist, dass die Unterabfrage sich auf ein Attribut der übergeordneten Abfrage (nämlich PersNr) bezieht. Daher sagt man auch, dass die Unteranfrage mit der äußeren (übergeordneten) Anfrage **korreliert**.

- Eine nicht korrelierte Unteranfrage (wie auf der vorherigen Folie) wird nur einmal berechnet, während korrelierte Unteranfragen i. A. für jedes Tupel der übergeordneten Anfrage neu berechnet werden müssen. → Perfomanceverlust!

Geschachtelte Anfragen (3)

- Angenommen die Relationen Studenten und Professoren enthalten als weiteres Attribut das jeweilige Geburtsdatum und man möchte alle Studenten ermitteln, die älter sind als der jüngste Professor.

```
SELECT s.*
FROM   Studenten s
WHERE  EXISTS
       ( SELECT p.*
         FROM Professoren p
         WHERE p.GebDatum < s.GebDatum);
```

- Die korrelierte SQL-Anfrage hat eine schlechte Performance, da die Unteranfrage für jeden Studenten neu ausgeführt wird.

Geschachtelte Anfragen (4)

- Besser ist es in diesem Fall, eine nicht korrelierte Anfrage zu formulieren:

```
SELECT s.*
FROM Studenten s
WHERE s.GebDatum < (SELECT MAX (p.GebDatum)
                    FROM Professoren p );
```

- Welche Professoren halten überhaupt keine Vorlesungen?

```
SELECT Name
FROM Professoren
WHERE NOT EXISTS (SELECT *
                  FROM Vorlesungen
                  WHERE gelesenVon = PersNr);
```

Mengenoperationen mit SQL

- Die klassischen Operationen der Mengenlehre heißen in SQL
  - Union (für Vereinigungsmenge)
  - Intersect (für Schnittsmenge)
  - Except bzw. minus bei Oracle (für Differenzmenge)
- Um die Anzahl aller Angestellten zu bestimmen, schreibt man:

```
SELECT COUNT (*)
FROM ((SELECT PerNr FROM Professoren)
        UNION
       (SELECT PerNr FROM Assistenten));
```

- Der Operator "in" (bzw. "not in") testet auf Mitgliedschaft in einer Menge und liefert einen Wahrheitswert. Beispiel:

```
SELECT Name FROM Professoren
WHERE PersNr NOT IN (SELECT gelesenVon
                     FROM Vorlesungen);
```

Der Selbst-Verbund

- Eine auf den ersten Blick etwas kuriose Konstruktion ist der Selbst-Verbund (self join) – also eine FROM-Klausel, in der dieselbe Relation mehrfach vorkommt.

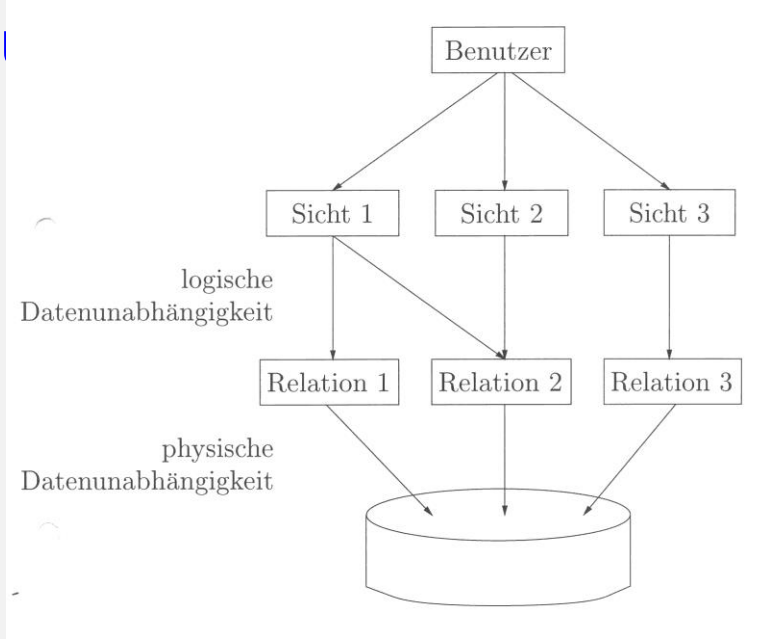
- Suche Paare von Professoren, die in der derselben Stadt wohnen, z.B. zwecks Bildung von Fahrgemeinschaften:

```
SELECT x.Name, y.Name
FROM   Professoren x, Professoren y
WHERE  x.Wohnort = y.Wohnort
AND    x.PersNr < y.PersNr;
```

- Verständnisfrage: Was passiert wenn man auf die zweite Bedingung (*x.PersNr < y.PersNr)* verzichtet?

Sichten (1)

- Bereits in Kapitel 1 wurden temporäre Tabellen (als Sichten bzw. Views bezeichnet) angesprochen, die ein Mittel zur Anpassung des Datenbanksystems an die Bedürfnisse der unterschiedlichen Benutzergruppen (
  Gewährleisten des Datenschutzes darstellen. Zur Erinnerung:



- Um z.B. sicherzustellen, dass nicht alle Anwender das Ergebnis von Prüfungen einsehen können, formuliert man

```
CREATE VIEW prüfenSicht AS
    SELECT MatrNr, VorlNr, PersNr
    FROM prüfen;
```

Sichten (2)

- Folgende Sicht verbindet die Studenten mit den Professoren, bei denen sie Vorlesungen haben:

```
CREATE VIEW StudProf (sName, Semester, Titel, pName) AS
    (SELECT s.Name, s.Semester, v.Titel, p.Name
     FROM Studenten s, hören h, Vorlesungen v, Professoren p
     WHERE s.MatrNr = h.MatrNr
        AND h.VorlNr = v.VorlNr
        AND v.gelesenVon = p.PersNr);
```

- Auf Basis dieser Sicht kann ein neue Abfrage recht einfach formuliert werden. Beispielsweise in welchem Semester sich die Studenten von Professor Keppler gerade befinden:

```
SELECT DISTINCT Semester
FROM StudProf
WHERE pName = 'Keppler';
```