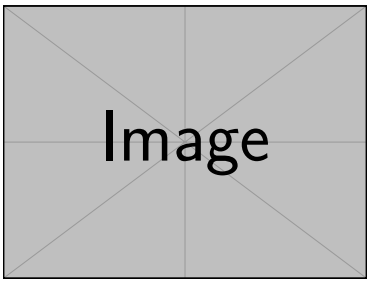
Notatki z wykładów

Matematyka – Analiza I



Twoje Imię i Nazwisko

Semestr zimowy 2025/2026

Hochschule Emden/Leer

Contents

1 Relationale Algebra

1 Relationsale Algebra

Relationale Algebra to zbiór operacji, które przyjmują jedną lub więcej relacji (tabel) jako dane wejściowe i zwracają nową relację jako wynik. Wszystko w niej opiera się na zbiorach i operacjach matematycznych.

Die wichtigste operationen:

Table 1.1: Wichtige Operationen der Relationalen Algebra

Operation	**Symbol**	**Beschreibung**
Selektion	σ	Wählt Tupel aus, die eine gegebene Bedingung erfüllen.
Projektion	π	Wählt bestimmte Attribute (Spalten) einer Relation aus.
Vereinigung	$\cup$	Kombiniert die Tupel zweier Relationen mit gleicher Struktur (wie `UNION` in SQL).
Differenz	$-$	Liefert die Tupel, die in der ersten, aber nicht in der zweiten Relation vorkommen.
Kartesisches Produkt	$\times$	Bildet alle möglichen Kombinationen von Tupeln aus zwei Relationen.
Join (Verbund)	$\bowtie$	Verknüpft zwei Relationen über gleiche Attribute oder Bedingungen.

1.1 Relationen

Eine Relation R (Tabelle) ist eine Teilmenge des Kreuzproduktes $Att_1 \times ... \times Att_n$. Dies wird $R \subseteq Att_1 \times ... \times Att_n$ geschrieben.

Hinweis

Relacja to mattematyczny model tabeli w relacynej bazie danych. Jest to zbiór tuples, które mają taką samą struktórę atrybutów:
$R \subseteq Att_1 \times ... \times Att_n$

gdzie:

R - nazwa relacji (np. Student)

$A_1, A_2, ..., A_n$ nazy atrybutów(Matrikellnummer, Name, Fachrichtung).

1.2 Vereinigungsverträglichkeit

2 Relationen sind Vereinigungsverträglich, wenn sie:

1. denselben Anzahl an Attributen haben
2. die entsprechenden Attribute A_i in R und B_i in S denselben Datentyp oder einen gemeinsamen Obertyp besitzen.

Hinweis

$$R \subseteq Att_1 \times \cdots \times Att_n$$
$$S \subseteq Btt_1 \times \cdots \times Btt_n$$
$$mitTyp(A_i) = Typ(B_i)$$

1.2.1 Beispiel

Table 1.2: Beispiel: Vereinigungsverträgliche Relationen

Relation R

Matr-nummer	Name
101	Anna
102	Ben

Relation S

Matr-nummer	Name
103	Carla
104	David

Relation C

Name	Alter
Anna	21
Ben	22

Die relationen R und S sind verträglich, da Sie in einer tabelle dargestellt werden (verbunden). Die relationen R und C oder S und C sind net kompatibel, da die Attribute nicht gleich sind (siehe subsection 1.3)

1.3 Mengenoperationen für Relationen

Seien R und S vereinigungsverträglich, dann kann man neue Relationen berechnen:

1. **Schnittmenge** $R \cap S = \{ r \mid r \in R \wedge r \in S \}$ - Einträge, die in beiden Relationen vorkommen
2. **Vereiguung** $R \cup S = \{ r \mid r \in R \vee r \in S \}$ - Zusammenfassung aller Einträge der Relationen
3. **Differenz** $R - S = \{ r \mid r \in R \wedge r \notin S \}$ - Suchen nach EInträgen, die nur in der ersten, aber nicht in der zweiten Relation vorkommen

Hinweis

Bei Relationen handelt es sich um Mengen, daher keine Zeile kommt doppelt vor!

1.3.1 Beispiel

VK

Verkäufer	Produkt	Käufer
Meier	Hose	Schmidt
Müller	Rock	Schmidt
Meier	Hose	Schulz

VK2

Verkäufer	Produkt	Käufer
Müller	Hemd	Schmidt
Müller	Rock	Schmidt
Meier	Rock	Schulz

VK ∪ VK2

Verkäufer	Produkt	Käufer
Meier	Hose	Schmidt
Müller	Rock	Schmidt
Meier	Hose	Schulz
Müller	Hemd	Schmidt
Meier	Rock	Schulz

VK ∩ VK2

Verkäufer	Produkt	Käufer
Müller	Rock	Schmidt

VK - VK2

Verkäufer	Produkt	Käufer
Meier	Hose	Schmidt
Meier	Hose	Schulz

1.4 Projektion

Sei $R \subseteq Att_1 \times \cdots \times Att_n$ eine Relation und $B_1, \ldots, B_j$ verschiedene Attribute aus der Menge $\{Att_1, \ldots, Att_n\}$.

Dann ist die **Projektion** von R auf $B_1, \ldots, B_j$, geschrieben als

$$\text{Proj}(R, [B_1, \ldots, B_j]),$$

die Relation, die entsteht, wenn man aus R alle Spalten entfernt, die nicht in $B_1, \ldots, B_j$ enthalten sind.

Die Reihenfolge der Attribute $B_1, \ldots, B_j$ bestimmt zugleich die Reihenfolge der Spalten in der Ergebnisrelation.

Hinweis

Projekcja służy do wyboru określonych kolumn (atrybutów) z relacji. Odrzuca wszystkie pozostałe atrybuty i często też usuwa duplikaty, ponieważ relacja w matematycznym sensie to zbiór (a zbiór nie zawiera powtórzeń).

1.4.1 Beispiel

Proj$(VK, [Käufer, Product])$

Käufer	Produkt
Schmidt	Hose
Schmidt	Rock
Schulz	Hose

Proj$(VK, [Verkäufer])$

Verkäufer
Meier
Müller

Proj$(VK \cap VK2, [Produkt])$

Produkt
Rock

1.5 Umbennenung (Rename)

Sei R eine Relation, dann bezeichnet $Ren(R, T)$ eine Relation mit gleichem Inhalt wie R, die T gennant wird.

Hinweis

Tego używa się gdy tabela sama ze sobą musi być zestawiona. Jeśli mamy 2 razy nazwę tej samej tabeli i potem chcemy operowac na Atrybutach tej tabeli to SQL nie wie o ktorą tabekę nam chodzi. Dlatego robimy TAB1 i $Ren(TAB1, TAB2)$ i terazpod TAB1 i TAB2 mamy tą samą tabelę i moeby operowac na jej kolumnach

1.6 Auswahl (Select)

Sei R eine Relation, dann bezeichnet $Sel(R, Bed)$ eine Relation, die alle Zeilen aus R beinhaltet, die die Bedingung Bed erfüllen.

Syntax

Syntax der Bedingungen Bed: $Att_1 \;\; OPERATOR \;\; KONSTANTE$

- OPERATOR - =, <>, <=, >=, <, >
- KONSTANTE - muss ein Wert des zum Attribut gehörenden Datentyps sein. Es kann auch ein Attribut aus anderer Spalte sein - hierbei muss der Typ des Attibunts gleich sein, oder sie müssen einen Gemeinsamen Obertyp besitzen

Es besteht auch die möglichkeit mehrere Bedingungen einzuführen:

- $Bed_1 \;\; AND \;\; Bed_2$ - beide Bedingungen sollen erfüllt sein
- $Bed_1 \;\; OR \;\; Bed_2$ - mindenstens eine der Bedingungen soll erfüllt sein
- $NOT \;\; Bed_1$ - die Bedingung soll nicht erfüllt sein
- (Bed_1) - die Bedingung in Klammern werden zuerst ausgewertet

1.6.1 Beispiel: Alle Berkäufe, die Meier gemacht hat

$Sel(VK, \; VK.Verkäufer =$ 'Meier')

Verkäufer	Produkt	Käufer
Meier	Hose	Schmidt
Meier	Hose	Schluz

1.6.2 Beispiel: Alle Käufer, die bei Meier gekauft haben

$Proj(Sel(VK,\ VK.Verkäufer = \text{'Meier'}), [\text{'Käufer'}])$

Käufer
Schmidt
Schluz

1.6.3 Beispiel: Alle Verkäufr, die Meier gemacht hat und die nicht den Kunden Schulz betreffen

2 Ein Verknüpfungsoperator für Relationen

Bislang beziehen sich operationen auf einzelne Tabellen. Durch das kreuzprodukt können mehrerer, auch verschiedene Tabellen miteinander Verknüpft.

2.1 Verknüpfung von Tupeln (Konkatenation)

Seien R und S Relationen mit $r = \{r_1, \ldots, r_n\} \in R$ und $s = \{s_1, \ldots, s_n\} \in S$. Dann ist die Verknüpfung oder Konkatenation von r mit s, geschrieben $r \circ s$, definiert als $\{r_1, \ldots, r_n, s_1, \ldots, s_n\}$.

2.2 Kreuzprodukt

Seien R und S Relationen, dann ist das kreuzprodukt von R und S, geschrieben $R \times S$, sefiniert durch $R \times S = \{r \circ s | r \in R \text{ und } s \in S\}$

Hinweis

Konkatenacja to operacja na pojedynczych elementach, łączy je w jeden dłuszy element. Kreuzprodukt to operacja na zbiorach elementów, która generuje nową relację, która zawiera wszystkie moliwe kombinacje krotek z R i S

Konkatenation vs Kreuzprodukt

Seien

$$R = \{(a_1), (a_2)\} \quad \text{und} \quad S = \{(b_1), (b_2)\}.$$

Dann ist die **Konkatenation** einzelner Tupel definiert als:

$$(a_1) \circ (b_1) = (a_1, b_1)$$

Das **Kreuzprodukt** der Relationen R und S besteht aus allen möglichen Konkatenationen von Tupeln aus R und S:

$$R \times S = \{\, r \circ s \mid r \in R,\, s \in S \,\}$$

Konkret ergibt sich hier:

$$R \times S = \{(a_1, b_1), (a_1, b_2), (a_2, b_1), (a_2, b_2)\}$$

2.3 Übung

Relationen

Projekt

ProNr	Name
1.	Schachtel
2.	Behang

Aufgabe

AufNr	Arbeit	ProNr
1.	knicken	1
2.	kleben	1
3.	knicken	2
4.	färben	2

Maschine

Mname	Dater	AufNr
M1	2	1
M2	3	1
M1	3	2
M3	2	3
M1	1	4
M4	3	4

1. Geben Sie die Namen aller möglichen Arbeiten an

$$Proj(Aufgabe, [Arbeit])$$

2. Geben Sie zu jedem Projektnamen die zugehörigen Arbeiten an. Das Ergebnis ist eine Relation mit den Attributen „Name" und „Arbeit".

$$\text{Proj}\Big(\text{Sel}(Projekt \times Aufgabe,\ Projekt.ProNr = Aufgabe.ProNr), [\text{Name, Arbeit}]\Big)$$

Przykład dla Kreuzprodukt $Projekt \times Aufgabe$

$Projekt \times Aufgabe$

ProNr	Name	AufNr	Arbeit	ProNr
1	Schachtel	1	knicken	1
1	Schachtel	2	kleben	1
1	Schachtel	3	knicken	2
1	Schachtel	4	färben	2
2	Behang	1	knicken	1
2	Behang	2	kleben	1
2	Behang	3	knicken	2
2	Behang	4	färben	2

3. Welche Maschinen werden zum Knicken genutzt?

$$\text{Proj}\big(\text{Sel}(Aufgabe \times Maschine,\ Aufgabe.AufNr = Maschine.AufNr \text{ AND } Aufgabe.Arbeit = \text{'knicken'}),\ [\text{Mname}]\big)$$

4. Geben Sie zu jedem Projektnamen die Maschinen aus, die genutzt werden

$$\text{Proj}\big(\text{Sel}(Projekt \times Aufgabe \times Maschine, Projekt.ProNr = Aufgabe.ProNr \text{ AND } Aufgabe.AufNr = Maschine.AufNr),\ [ProjName, Mname]\big)$$

5. Geben Sie alle Projekte (deren Namen) aus, bei denen geknickt und gefärbt wird

$$\text{Proj}\big(\text{Sel}(Projekt \times Aufgabe \times \text{Ren}(Aufgabe, A2),\ Projekt.ProNr = Aufgabe.ProNr \text{ AND } Projekt.ProNr = A2.ProNr \text{ AND } Aufgabe.Arbeit = \text{'knicken' AND } A2.Arbeit = \text{'färben'}),\ [\text{Name}]\big)$$

2 Formalisierung von Tabellen in SQL

1 Tabellendefinition in SQL

Verkäufer

Vnr	Name	Status	Gehalt
1001	Udo	Junior	1500
1002	Ute	Senior	1900
1003	Uwe	Senior	2000

Kunde

Knr	Name	Betreuer
1	Egon	1001
2	Erwin	1001
3	Erna	1002

Hinweis: Durch eine Fremdschlüsselbeziehung (Kunde.Betreuer → Verkaeufer.Vnr) wird festgelegt, dass jeder Kunde einen existierenden Verkäufer als Betreuer haben muss.

Listing 2.1: Tabellendefinition in SQL

```
CREATE TABLE Verkaeufer(
    Vnr INTEGER,
    Name VARCHAR(6),
    Status VARCHAR(7),
    Gehalt INTEGER,
    PRIMARY KEY (Vnr)
);
CREATE TABLE Kunde(
    Knr INTEGER,
    Name Varchar(6),
    Betreuer INTEGER,
    PRIMARY KEY (Knr),
    CONSTRAINT FK_Kunde
        FOREIGN KEY (Betreuer)
        REFERENCES Verkaeufer(Vnr)
)
```

- Eigenschaften (z.B. Attribute) werden durch Kommata getrennt
- Zeilenumbrüche zur Übersichtlichkeit, aber nicht zwingend erforderlich

- Trennung von mehreren Befehlen üblicherweise durch Semikolon

1.1 Primärschlussel

Klucz główny (Primärschlüssel) to mechanizm, który zapewnia, że każdy wiersz w tabeli jest unikalny i da się go jednoznacznie zidentyfikować. Ale oprócz „weryfikacji różności" ma też kilka dodatkowych, bardzo ważnych funkcji.

- Kady wpis w tej kolumnie musi być unikalny - numery nie mogą się powtarzać
- Blokuje wartości NULL - Kolumna będąca kluczem głównym musi zawsze zawierać jakąś wartość — nie może być pusta (NULL).
- Umożliwia tworzenie relacji z innymi tabelami - Klucz główny jest punktem odniesienia dla kluczy obcych (Foreign Keys).

Beispiel

Linia:

```
PRIMARY KEY (Vnr)
```

oznacza, że kolumna **Vnr** to klucz główny (Primärschlüssel).

1.2 Constraint

Constraint (ograniczenie) to reguła określająca zasady, jakie wartości mogą występować w kolumnach tabeli. **Baza danych automatycznie sprawdza ich poprawność przy dodawaniu, usuwaniu lub modyfikowaniu danych.** Dzięki ograniczeniom możliwe jest zapewnienie integralności i spójności danych pomiędzy tabelami.

- Ograniczenia mogą dotyczyć pojedynczej kolumny lub relacji między tabelami.
- Każde ograniczenie może posiadać nazwę — ułatwia to późniejsze modyfikacje lub usunięcie.
- Do najczęściej używanych ograniczeń należą:
  - `PRIMARY KEY` – gwarantuje unikalność i brak wartości `NULL`.
  - `FOREIGN KEY` – definiuje relację między tabelami (klucz obcy) - To kolumna w jednej tabeli, która odwołuje się do klucza głównego w innej tabeli.
  - `REFERENCES` – wskazuje, do której tabeli i kolumny odnosi się klucz obcy.

Beispiel

```
CREATE TABLE Kunde(
    Knr INTEGER,
    Name VARCHAR(6),
    Betreuer INTEGER,
    PRIMARY KEY (Knr),
    CONSTRAINT FK_Kunde
        FOREIGN KEY (Betreuer)
        REFERENCES Verkaeufer(Vnr)
);
```

Powyższy przykład definiuje ograniczenie o nazwie **FK_Kunde**, które ustala relację pomiędzy kolumną **Betreuer** w tabeli **Kunde** a kolumną **Vnr** w tabeli **Verkaeufer**. Dzięki temu baza danych zapewnia, że każda wartość w kolumnie **Betreuer** odpowiada istniejącemu sprzedawcy w tabeli **Verkaeufer**.

Hinweis

`CONSTRAINT` definiert eine benannte Regel (Einschränkung), die von der Datenbank automatisch überprüft wird. `FOREIGN KEY` gibt die Spalte in der aktuellen Tabelle an, während `REFERENCES` bestimmt, auf welche Tabelle und Spalte sich diese Spalte bezieht.
Bei jedem Einfügen eines neuen Datensatzes in die Tabelle `Kunde` überprüft das Datenbanksystem automatisch, ob der in der Spalte `Betreuer` eingetragene Wert in `Verkaeufer.Vnr` existiert. Ist dies der Fall, wird der Datensatz gespeichert, andernfalls wird der Vorgang abgelehnt.

Hinweis

`FOREIGN KEY` zawsze musi odnosić się do `PRIMARY KEY` (lub innego klucza unikalnego) w innej tabeli.

Więcej do tego tematu w punkcie section 5

2 Einfügen, Löschen und Ändern von Daten

2.1 Einfügen

```
INSERT INTO Verkaeufer VALUES(1001,'Udo','Junior',1500);
INSERT INTO Verkaeufer VALUES(1002,'Ute','Senior',1900);
INSERT INTO Verkaeufer VALUES(1003,'Uwe','Senior',2000)
```

Eingabe der Werte muss der Reihenfolge der Attribute in der Tabelle entsprechen. Es muss für jede Spalte ein Wert angegeben werden

2.1.1 Einfügen mit Angabe der Spalten

Es können auch eingtäge in explicite Spalten gemacht werden. Dann müssen wir erstmal die Spalten nennen und dann entsprechend Values übergeben

```
INSERT INTO Verkaeufer(Vnr,Name,Status) VALUES(1004,'Ulf','Junior');
INSERT INTO Verkaeufer(Vnr,Gehalt,Name) VALUES(1005,1300,'Urs')
```

2.1.2 weitere Beispiele

Einfügen mit Nullwerten

```
INSERT INTO Kunde VALUES(4,'Edna',NULL);
```

2.2 Update

Dieser Befehl ändert die Werte eines bestehenden Datensatzes in der Tabelle **Kunde**. Mit `SET` werden die neuen Werte für die angegebenen Spalten festgelegt, und die `WHERE`-Bedingung bestimmt, welcher Datensatz aktualisiert wird. Alle definierten `CONSTRAINTS` (z. B. `FOREIGN KEY`) müssen dabei erfüllt bleiben – das bedeutet, dass der neue Wert in **Betreuer** nur gesetzt werden kann, wenn dieser auch in **Verkaeufer.Vnr** existiert.

```
UPDATE Kunde
    SET Name='Edwina', Betreuer=1002
    WHERE Name='Edna'
```

Der Befehl sucht alle Datensätze in der Tabelle **Kunde**, bei denen der Name **'Edna'** ist, und ändert diesen Namen zu **'Edwina'** sowie den Betreuer auf **1002**. Vor dem Speichern prüft die Datenbank alle `CONSTRAINTS` – die neue Betreuer-Nummer muss daher bereits in **Verkaeufer.Vnr** existieren, ansonsten wird die Aktualisierung verweigert.

Weiterer Beispiel

```
UPDATE Verkaeufer SET Gehalt=Gehalt * 1.05
```

2.3 Überprüfung des Inhalts einer Tabelle

Dieser Befehl zeigt alle gespeicherten Datensätze einer Tabelle an. Das Sternchen `*` steht dabei für „alle Spalten“. So können sämtliche Inhalte der angegebenen Tabelle überprüft werden, z. B. um zu kontrollieren, ob Einträge korrekt eingefügt oder aktualisiert wurden.

```
SELECT * FROM <Tabellenname>
```

2.4 Delete

```
DELETE FROM <Tabellenname> WHERE <Bedingung>
```

Mit dem `DELETE`-Befehl können Datensätze aus einer Tabelle gelöscht werden, die eine bestimmte Bedingung erfüllen. Wird keine Bedingung angegeben, löscht der Befehl `DELETE` alle Datensätze der Tabelle.

Beispiel

```
DELETE FROM Kunde WHERE Knr = 3;
```

In diesem Beispiel wird der Datensatz mit der Kundennummer `3` aus der Tabelle `Kunde` entfernt.

2.4.1 Delete on Cascade

Die Anweisung `ON DELETE CASCADE` definiert das Verhalten der Datenbank beim Löschen von Datensätzen in einer übergeordneten Tabelle. Sie legt fest, dass alle abhängigen Datensätze in der untergeordneten Tabelle automatisch gelöscht werden, sobald der zugehörige Datensatz in der übergeordneten Tabelle entfernt wird.

Im folgenden Beispiel bezieht sich die Tabelle **Kunde** über den Fremdschlüssel **Betreuer** auf die Tabelle **Verkaeufer**. Wird ein Verkäufer gelöscht, so werden alle Kunden, die diesem Verkäufer zugeordnet sind, automatisch mit gelöscht.

```
CREATE TABLE Verkaeufer (
    Vnr INTEGER PRIMARY KEY,
    Name VARCHAR(20)
);

CREATE TABLE Kunde (
    Knr INTEGER PRIMARY KEY,
    Name VARCHAR(20),
    Betreuer INTEGER,
    CONSTRAINT FK_Kunde
        FOREIGN KEY (Betreuer)
        REFERENCES Verkaeufer(Vnr)
        ON DELETE CASCADE
);
```

2.4.2 Verhalten beim Löschen abhängiger Datensätze

Der SQL-Standard definiert mehrere Möglichkeiten, wie eine Datenbank mit abhängigen Datensätzen umgeht, wenn ein referenzierter Eintrag aus der übergeordneten Tabelle gelöscht wird. Diese Optionen werden in der `FOREIGN KEY`-Definition mit der Klausel `ON DELETE` angegeben:

- **NO ACTION** — Standardverhalten. Das Löschen eines übergeordneten Datensatzes ist nicht erlaubt, solange abhängige Datensätze in der untergeordneten Tabelle existieren. (NO ACTION nie pozwala usunąć rekordu z tabeli nadrzędnej, jeśli w tabeli podrzędnej istnieją powiązane dane.)
- **CASCADE** — Entspricht der vorgestellten „Löschfortpflanzung“. Wird ein Datensatz in der übergeordneten Tabelle gelöscht, werden alle zugehörigen abhängigen Datensätze automatisch mit entfernt.
- **SET NULL** — Beim Löschen eines referenzierten Datensatzes wird der Fremdschlüsselwert in allen abhängigen Datensätzen automatisch auf `NULL` gesetzt.
- **SET DEFAULT** — Wenn für den Fremdschlüssel ein Standardwert (`DEFAULT`) definiert ist, wird dieser Wert in den abhängigen Datensätzen gesetzt, sobald der referenzierte Datensatz gelöscht wird.

```sql
CREATE TABLE Kunde (
    Knr INTEGER PRIMARY KEY,
    Name VARCHAR(20),
    Betreuer INTEGER DEFAULT 1000,
    CONSTRAINT FK_Kunde
        FOREIGN KEY (Betreuer)
        REFERENCES Verkaeufer(Vnr)
        ON DELETE SET DEFAULT
);
```

Tutaj Każdy klient (Kunde) ma przypisanego opiekuna (Betreuer), Jeśli ten opiekun zostanie usunięty, to kolumna Betreuer klienta zostanie automatycznie ustawiona na wartość domyślną (1000).

2.4.3 Löschen von Tabellen

Das Löschen ganzer Tabellen kann bei bestehenden Fremdschlüsselbeziehungen zu Problemen führen. Je nach gewählter Option wird das Verhalten des Befehls `DROP TABLE` vom Datenbankmanagementsystem (DBMS) unterschiedlich gehandhabt.

Beispielvarianten

- **DROP TABLE <Tabellenname> RESTRICT**
  Löscht die Tabelle nur, wenn keine abhängigen Tabellen existieren. Dadurch wird verhindert, dass Fremdschlüsselbeziehungen ins Leere zeigen.
- **DROP TABLE <Tabellenname> CASCADE**
  Löscht die angegebene Tabelle und entfernt in abhängigen Tabellen die entsprechenden Fremdschlüsselbeziehungen. Die eigentlichen Daten in diesen abhängigen Tabellen bleiben jedoch bestehen.
- **Kurzform: DROP TABLE <Tabellenname>**
  Führt – abhängig vom verwendeten DBMS – automatisch entweder die `RESTRICT`- oder `CASCADE`-Variante aus.

Hinweis

Das vorschnelle Löschen von Tabellen sollte vermieden werden, insbesondere in produktiven Datenbanken. Fremdschlüsselabhängigkeiten sollten zuvor sorgfältig geprüft werden, um Dateninkonsistenzen zu verhindern.

3 Datentypen in SQL

Obwohl ein allgemeiner SQL-Standard existiert, unterscheiden sich die Datentypen zwischen verschiedenen Datenbankmanagementsystemen (DBMS) erheblich. Die folgenden Ausführungen beziehen sich auf PostgreSQL und beschränken sich auf die wichtigsten Datentypen.

3.1 Ganze Zahlen

- `SMALLINT`: Wertebereich von `-32.768` bis `+32.767`
- `INTEGER`: Wertebereich von `-2.147.483.647` bis `+2.147.483.646`
- `BIGINT`: für sehr große Ganzzahlen (bis ca. $\pm 9{,}22 \times 10^{18}$)

3.2 Kommazahlen

- `DECIMAL(p, q)` oder `NUMERIC(p, q)`: ermöglicht die Angabe von Genauigkeit `p` (Gesamtzahl der Stellen) und `q` (Nachkommastellen); bis zu 131.072 Stellen vor und 16.383 Stellen nach dem Komma.
- `REAL`: Gleitkommazahlen mit mindestens sechs signifikanten Stellen (häufig auch als `FLOAT` bezeichnet).
- `DOUBLE PRECISION`: Gleitkommazahlen mit mindestens 15 signifikanten Stellen (häufig als `DOUBLE` bezeichnet).

3.3 Zeichenketten

- `CHAR(q)`: speichert immer exakt `q` Zeichen, gegebenenfalls mit Leerzeichen aufgefüllt.
- `VARCHAR(q)`: speichert bis zu `q` Zeichen.

3.4 Datum und Zeit

- `DATE`: repräsentiert ein Datum, Ausgabeformat z. B. `'YYYY-MM-DD'` (ISO 8601).
- `TIME`: repräsentiert eine Uhrzeit im Format `'HH:MM:SS'` (ISO 8601).
- `TIMESTAMP`: kombiniert Datum und Uhrzeit, z. B. `'YYYY-MM-DD HH:MM:SS'`.
  - Die Funktion `CURRENT_TIMESTAMP()` liefert den aktuellen Zeitpunkt zurück.
  - Beispiel: `INSERT INTO tab VALUES (..., CURRENT_TIMESTAMP, ...);`

3.5 Spezielle Datentypen für umfangreiche Daten

- `TEXT`: Speicherung sehr langer Zeichenfolgen (bis zu 1 GB); im Standard-SQL als `CLOB` (Character Large Object) bezeichnet.
- `BYTEA`: Speicherung von Binärdaten; im Standard-SQL als `BLOB` (Binary Large Object) bezeichnet.

Hinweis

Die Unterstützung und Einschränkungen dieser Datentypen hängen stark vom jeweiligen DBMS ab. Insbesondere große Text- und Binärfelder (`TEXT` , `BYTEA`) sind oft nur eingeschränkt in Abfragen nutzbar.

4 Wartości `NULL` i logika trójwartościowa

W języku SQL każda kolumna może przyjmować specjalną wartość `NULL`, która nie oznacza zera, pustego tekstu ani wartości fałszywej. `NULL` oznacza po prostu *„brak danych”* lub *„wartość nieznana”*. Z tego powodu w SQL stosowana jest nie klasyczna logika dwuwartościowa, lecz rozszerzona logika trójwartościowa (*three-valued logic*).

4.1 Logika dwuwartościowa

W klasycznej logice występują tylko dwa stany: *prawda (t)* i *fałsz (f)*.

X	Y	`NOT X`	`X AND Y`	`X OR Y`
t	t	f	t	t
t	f	f	f	t
f	t	t	f	t
f	f	t	f	f

4.2 Problem wartości `NULL`

Dla kolumn, które mogą zawierać `NULL`, porównania takie jak `=`, `<>`, `>`, `<` nie mogą zostać jednoznacznie rozstrzygnięte. Jeżeli jedna z wartości jest `NULL`, baza danych nie wie, czy warunek jest prawdziwy, czy fałszywy. Dlatego wprowadzono trzeci stan logiczny — *nieznany (u)*.

4.3 Logika trójwartościowa

W logice trójwartościowej każdy warunek może przyjmować jedną z trzech wartości: **prawda (t)**, **fałsz (f)** lub **nieznany (u)**.

X	Y	`NOT X`	`X AND Y`	`X OR Y`
t	t	f	t	t
t	f	f	f	t
t	u	f	u	t
f	t	t	f	t
f	f	t	f	f
f	u	t	f	u
u	t	u	u	t
u	f	u	f	u
u	u	u	u	u

4.4 Przykład praktyczny

```
SELECT * FROM Mitarbeiter WHERE Gehalt > 2000;
```

Jeśli w tabeli **Mitarbeiter** znajdują się następujące dane:

Name	Gehalt
Anna	3000
Bob	NULL
Eva	1500

to wynik zapytania będzie zawierał jedynie rekord **Anna**, ponieważ:

- **Anna:** $3000 > 2000 \rightarrow$ *prawda (t)*
- **Bob:** `NULL` $> 2000 \rightarrow$ *nieznany (u)*
- **Eva:** $1500 > 2000 \rightarrow$ *fałsz (f)*

SQL zwraca tylko te wiersze, dla których warunek jest **prawdziwy (w)**. Wiersze z wynikiem *nieznany (u)* są pomijane, ale nie są traktowane jako fałszywe.

4.5 Porównania z `NULL`

Aby uwzględnić wartości `NULL` w zapytaniach, należy używać operatorów:

- `IS NULL`
- `IS NOT NULL`

```sql
SELECT * FROM Mitarbeiter WHERE Gehalt IS NULL;
```

Porównania w stylu `= NULL` lub `<> NULL` zawsze zwrócą wynik *nieznany (u)*.

Podsumowanie

- Klasyczna logika: **prawda** / **fałsz**.
- Logika SQL: **prawda** / **fałsz** / **nieznany**.
- Porównanie z `NULL` $\rightarrow$ wynik **nieznany**.
- Aby sprawdzić brak wartości, używaj `IS NULL` i `IS NOT NULL`.

5 Ograniczenia integralności danych (`CONSTRAINTS`)

Ograniczenia integralności (`CONSTRAINTS`) służą do kontrolowania poprawności danych wprowadzanych do tabeli. Dzięki nim można automatycznie wymusić określone reguły, np. minimalne lub maksymalne wartości, unikalność, czy relacje między tabelami.

5.1 Przykład z warunkiem `CHECK`

Załóżmy, że sprzedawcy o statusie `Junior` mogą zarabiać maksymalnie 2000.

```
CONSTRAINT GehaltsgrenzeJunior
    CHECK (NOT(Status='Junior') OR Gehalt <= 2000)
```

Zasada działania:

- Dla wszystkich rekordów, w których `Status='Junior'`, sprawdzany jest dodatkowy warunek `Gehalt <= 2000`.
- Wiersze, które nie spełniają tego warunku, nie zostaną zaakceptowane przez bazę danych.
- Dla pozostałych rekordów (nie-`Junior`) warunek `NOT(Status='Junior')` jest prawdziwy, więc całe wyrażenie w `CHECK` również daje wynik *true*.

Interpretacja logiczna

Wyrażenie `NOT(Status='Junior') OR Gehalt <= 2000` odpowiada zasadzie: *„Jeśli pracownik ma status Junior, to jego pensja musi być mniejsza lub równa 2000."*

Przykład: Ograniczenie warunkowe dla wartości płacy

Załóżmy, że programiści o statusie **Junior** mogą zarabiać maksymalnie 2000. Aby to wymusić, można zdefiniować ograniczenie `CHECK`:

```
CONSTRAINT GehaltsgrenzeJunior
    CHECK (NOT(Status='Junior') OR Gehalt <= 2000)
```

Interpretacja

To wyrażenie odpowiada zasadzie logicznej:

„Jeśli pracownik ma status Junior, to jego pensja musi być mniejsza lub równa 2000."

W logice formalnej warunek ten można zapisać jako implikację:

$$Status = \text{'Junior'} \Rightarrow Gehalt \leq 2000$$

Ponieważ SQL nie posiada operatora „implikacji" ($\Rightarrow$), stosuje się równoważne logicznie wyrażenie:

$$\text{NOT}(Status = \text{'Junior'}) \text{ OR } Gehalt \leq 2000$$

Zasada działania

Warunek `CHECK` musi być prawdziwy (*TRUE*), aby rekord został przyjęty przez bazę danych. Operator `NOT` odwraca wynik porównania `Status='Junior'`, a operator `OR` zwraca wartość *TRUE*, jeśli przynajmniej jeden z warunków jest spełniony.

Status	Gehalt	`Status='Junior'`	`NOT(...)`	`Gehalt<=2000`	Wynik końcowy
Junior	1800	t	f	t	**TRUE**
Junior	2500	t	f	f	**FALSE**
Senior	2500	f	t	f	**TRUE**
Senior	1500	f	t	t	**TRUE**

Wniosek

- Jeśli pracownik **nie jest Juniorem**, warunek jest zawsze spełniony.
- Jeśli **jest Juniorem**, musi spełniać `Gehalt <= 2000`.
- W przeciwnym razie baza danych odrzuci wiersz przy próbie wstawienia lub aktualizacji.

5.2 Dodatkowe ograniczenia integralności w SQL

Poza kluczami głównymi (`PRIMARY KEY`) i obcymi (`FOREIGN KEY`) w SQL można definiować dodatkowe ograniczenia — tzw. *Constraints*, które zapewniają spójność danych i kontrolują poprawność wartości w tabelach.

5.2.1 Definiowanie ograniczeń

Przykład tabeli ze zdefiniowanymi constraintami:

```
CREATE TABLE Verkaeufer (
    Vnr INTEGER,
    Name VARCHAR(6) NOT NULL,
    Status VARCHAR(7) DEFAULT 'Junior'
        CONSTRAINT StatusHatWert
        CHECK (Status IS NOT NULL),
    Gehalt INTEGER,
    PRIMARY KEY (Vnr),
    CONSTRAINT GehaltImmerAngegeben
        CHECK (Gehalt IS NOT NULL)
);
```

Opis

- `NOT NULL` – kolumna nie może zawierać wartości pustych.
- `DEFAULT 'Junior'` – przypisuje domyślną wartość, jeśli nie zostanie podana przy wstawianiu danych.
- `CHECK(...)` – warunek logiczny, który musi być spełniony dla każdego rekordu.
- `CONSTRAINT <nazwa>` – nadaje ograniczeniu nazwę, dzięki czemu pojawia się ona w komunikatach błędów.

5.2.2 Rodzaje constraintów

- **Spalten-Constraint** — dotyczy jednej kolumny (np. `Name VARCHAR(6) NOT NULL`).
- **Tabellen-Constraint** — odnosi się do całego wiersza tabeli (np. `CHECK(Gehalt IS NOT NULL)`)
- **UNIQUE** — gwarantuje unikalność wartości w kolumnie lub kombinacji kolumn.

```
-- Przyklady unikalnosci
Name VARCHAR(6) UNIQUE
CONSTRAINT EindeutigerName UNIQUE(Name)
Name VARCHAR(6) NOT NULL UNIQUE
UNIQUE(X, Y) -- unikalnosc kombinacji kolumn
```

Uwagi praktyczne

- `NOT NULL` nie jest wymagane dla kluczy głównych – wynika to z ich definicji.
- Jeśli baza danych wykryje, że warunek `CHECK` zwraca `FALSE`, operacja (INSERT/UPDATE) zostanie przerwana.
- W niektórych DBMS operacja jest przerywana także wtedy, gdy warunek zwróci `UNKNOWN` (np. z powodu `NULL`).
- Ograniczenia dotyczące wielu kolumn muszą być definiowane jako **Tabellen-Constraints**.

6 Änderungen in Tabellenstrukturen

Hinweis

Die Änderungen von Tabellenstrukturen sollen vermieden werden - möglichst nur Ergänzungen für existirende Tabellen oder Hinzufühen neuer Tabellen.

Die Änderungen können über Befehl `ALTER` vorgenommen werden

```
ALTER TABLE Verkaeufer ADD Klasse VARCHAR(1);

ALTER TABLE Verkaeufer ADD
    CONSTRAINT Klassenwerte CHECK (Klasse='A'
    OR Klasse ='B' OR Klasse ='C');
```

- Bei Spaltenergänzung kann ein Default-Wert angegeben werden, sonst wird die Spalte mit NULL-Werten gefüllt.
- Löschen von Constraints:

  ```
  ALTER TABLE <Tabellenname> DROP CONSTRAINT <Constraintname>
  ```

- Constraints können de- und aktiviert werden, um kurzzeitig erforderliche inkonsistente Zustände zu erlauben

3 Anfragen in SQL

Gehenge

Gnr	GName	Flaeche
1	Wald	20
2	Feld	10
3	Weide	9

Tier

Gnr	TName	Gattung
1	Laber	Bär
1	Sabber	Bär
2	Klopfer	Hase
3	Bunny	Hase
2	Harald	Schaf
3	Walter	Schaf

Art

Gattung	Min Fläche
Bär	8
Hase	2
Schaf	5

1 Ausgabe der Informationen

```
SELECT Gname FROM Gehege
```

GName
Wald
Feld
Weide

Ergebniss

- `FROM` - Tabellen mit relevanten Informationen
- `SELECT` - Attribute für die Ausgabe
- Ergebniss ist eine Tabelle, die nicht gespeichert wird

Ausgabe mit Angabe der Tabelle beim Attribut

```
SELECT Gehege.Gname FROM Gehege
```

1.1 DISCTINCT

Bei der Ausgabe aller Tiergattungen über den Befehl

```
SELECT Tier.Gattung FROM Tier
```

werden alle Tierarten ausgegeben:

Gattung
Bär
Bär
Hase
Hase
Schaf
Schaf

Daher muss den Befehl `DISTINCT` verwendet werden, um alles auszufiltern:

```
SELECT DISTINCT Tier.Gattung FROM Tier
```

Ergebnisstabelle:

Gattung
Bär
Hase
Schaf

1.2 Ausgabe des gesamten Inhalts einer Tabelle

```
SELECT * FROM Gehenge
```

Ergebnis:

Gnr	GName	Flaeche
1	Wald	20
2	Feld	10
3	Weide	9

1.3 Ausgabe mit mathematischen Operation

```
SELECT Gehege.Gname, (Gehege.Flaeche/50.0)*100.0 FROM Gehege
```

Ergebnis:

GName	?COLUMN?
Wald	40
Feld	20
Weide	18

Hinweis

übliche Operatoren möglich: + - * /

1.4 Konkatenation von Zeichen

1.4.1 `||` - Operator

```
SELECT Tier.Gattung || '::' || Tier.Tname FROM Tier
```

1.4.2 `CONCAT` - Operator

```
SELECT CONCAT(Tier.Gattung, '::', Tier.Tname) FROM Tier
```

Ergebnis:

CONCAT
Bär::Laber
Bär::Sabber
Hase::Klopfer
Hase::Bunny
Schaf::Harald
Schaf::Walter

Hinweis

Spaltenüberschrift entspricht der Berechnungsvorschrift, evtl. aber verkürzt (oder auch `?COLUMN?`)

1.5 Umbenennung von Spalten

Spaltenüberschriften können im Ergebnis einer `SELECT`-Abfrage umbenannt werden. Dies ist besonders nützlich, wenn:

- die Spaltennamen zu technisch oder zu lang sind,
- Berechnungsausdrücke im Ergebnis erscheinen,
- mehrere Tabellen ähnliche Spaltennamen enthalten.

Die Umbenennung erfolgt durch die Vergabe eines *Aliasnamens* für eine Spalte. Dabei gibt es zwei Varianten:

1. **Kurzform (oft in älteren Systemen):**

```
SELECT Gehege.Gname Gatter
FROM Gehege;
```

2. **SQL-Standard (empfohlen):**

```
SELECT Gehege.Gname AS Gatter
FROM Gehege;
```

Beide Varianten führen zum gleichen Ergebnis.

Beispiel mit Berechnung

Auch berechnete Spalten können mit einem Alias versehen werden:

```
SELECT Gehege.Gname AS Gatter,
    Gehege.Flaeche * 10000 AS Quadratzentimeter
FROM Gehege;
```

Ergebnis:

GATTER	QUADRATZENTIMETER
Wald	200000
Feld	100000
Weide	90000

Beispiel mit festen Werten

Neue Spalten können auch mit festen Werten erzeugt werden:

```
SELECT 'Unser Zoo' AS Zooname,
    Tier.Tname AS Tiername,
    2005 AS Einzug,
    42 AS Beispielwert
FROM Tier;
```

Ergebnis:

ZOONAME	TIERNAME	EINZUG	BEISPIELWERT
Unser Zoo	Klopfer	2005	42
Unser Zoo	Bunny	2005	42

1.6 Reihenfolde bei der Ausgabe

Reihenfolge der Daten bei der Ausgabe muss nicht der Reihenfolge beim Eintragen entsprechen. Ausgabereihenfolge kann über `ORDER` BY gesteuert werden

- `ASC` ergibt aufsteigende (Default)
- `DESC` ergibt absteigende Sortierung

1.6.1 Ausgabe der Gehege nach aufsteigender Größe

```
SELECT Gehege.Gname, Gehege.Flaeche
    FROM Gehege
    ORDER BY Gehege.Flaeche ASC
```

1.6.2 Ausgabe der Gattungen absteigend nach Flächenbedarf

```
SELECT Art.Gattung, Art.MinFlaeche
    FROM Art
    ORDER BY Art.MinFlaeche DESC
```

Sortierung nach zwei Kriterien

Eine Ergebnistabelle kann nach mehreren Spalten sortiert werden. Dabei wird zunächst nach der ersten Spalte sortiert, und innerhalb gleicher Werte nach der zweiten Spalte.

```
SELECT *
    FROM Tier
    ORDER BY Tier.Gattung DESC, Tier.Tname ASC;
```

In diesem Beispiel erfolgt die Sortierung **absteigend** nach `Gattung` und **aufsteigend** nach `Tname`.

2 Auswahlkriterien mit `WHERE`

Mit der `WHERE`-Klausel können gezielt nur jene Datensätze ausgewählt werden, die bestimmte Bedingungen erfüllen. Ohne `WHERE` werden alle Zeilen ausgegeben.

Beispiel 1: Alle Tiere der Gattung Schaf

```
SELECT Tier.Tname
    FROM Tier
    WHERE Tier.Gattung = 'Schaf';
```

Beispiel 2: Gattungen mit einer Mindestfläche von mindestens 4, die jedoch keine Bären sind

```
SELECT Art.Gattung
    FROM Art
    WHERE Art.MinFlaeche >= 4 AND Art.Gattung <> 'Baer';
```

Hinweis

- Mehrere Bedingungen können mit `AND`, `OR` und `NOT` kombiniert werden.
- Für Textvergleiche wird häufig das Schlüsselwort `LIKE` verwendet.

2.1 Mustervergleich mit `LIKE`:

- `%` steht für beliebig viele Zeichen.

- `_` steht für genau ein Zeichen.

Beispiel 1: Ausgabe aller Tiernamen, die mit 's' beginnen

```
SELECT *
    FROM Tier
    WHERE Tname LIKE 'S%';
```

Dies gibt alle Tiere aus, deren Name mit `S` beginnt (z. B. Sabber oder Susi).

Beispiel 2: Ausgabe aller Tiernamen, die ein 'a' enthalten

```
SELECT Tier.Tname
    FROM Tier
    WHERE Tier.Tname LIKE '%a%'
```

Beispiel 3: Ausgabe aller Tiernamen, deren dritter Buchstabe ein 'n' ist

```
SELECT Tier.Tname
    FROM Tier
    WHERE Tier.Tname LIKE '__n%'
```

Beispiel 4: Ausgabe aller Tiernamen, die ein % enthalten

```
SELECT Tier.Tname
    FROM Tier
    WHERE Tier.Tname LIKE '%/%%' ESCAPE '/'
```

Hinweis

Folgenze Zeilen sind ävuivalent

```
Art.Gattung<>'Baer'
Art.Gattung NOT LIKE 'Baer'
NOT (Art.Gattung LIKE 'Baer')
```

Verwendung erster Variante bevorzugt, um kenntlich zu machen, dass hier ein exakter Textvergleich erfolgt

2.2 Umwandlung von Texten mit `LOWER` und `UPPER`

Mit den Funktionen `LOWER()` und `UPPER()` können Textwerte in Klein- bzw. Großbuchstaben umgewandelt werden. Dies ist besonders nützlich bei Vergleichen, da SQL-Systeme oft zwischen Groß- und Kleinschreibung unterscheiden.

Beispiel 1: Ausgabe aller Tiere der Gattung **Schaf**, unabhängig von der Schreibweise

```
SELECT LOWER(Tier.Tname)
    FROM Tier
    WHERE LOWER(Tier.Gattung) = 'schaf';
```

Beispiel 2: Ausgabe der Tiernamen in Großbuchstaben

```
SELECT UPPER(Tier.Tname) AS Gross
    FROM Tier;
```

Hinweis

- `LOWER()` – wandelt alle Buchstaben in Kleinbuchstaben.
- `UPPER()` – wandelt alle Buchstaben in Großbuchstaben.

2.3 Auswertung einer SQL-Anfrage und Umgang mit `NULL`

Auswertung von Bedingungen in SQL

Bei der Auswertung einer SQL-Anfrage werden nur jene Zeilen in das Ergebnis aufgenommen, für die die `WHERE`-Bedingung den Wert `TRUE` liefert. Zeilen mit dem Ergebnis `FALSE` oder `UNKNOWN` (z. B. durch `NULL`) werden nicht berücksichtigt.

Beispiel: Erstellung und Befüllung einer Tabelle

```
CREATE TABLE Person(
    Pnr INTEGER,
    Name VARCHAR(5),
    Gehalt INTEGER,
    PRIMARY KEY (Pnr)
);

INSERT INTO Person VALUES (1, 'Eddy', 2500);
INSERT INTO Person VALUES (2, 'Egon', NULL);
INSERT INTO Person VALUES (3, 'Erna', 1700);
```

Beispiel 1: Personen mit einem Gehalt unter 2000

```
SELECT Person.Name
    FROM Person
    WHERE Person.Gehalt < 2000;
```

Ergebnis:

NAME
Erna

Beispiel 2: Einbeziehen von Personen ohne Gehaltsangabe

```
SELECT Person.Name
    FROM Person
    WHERE Person.Gehalt < 2000 OR Person.Gehalt IS NULL;
```

Ergebnis:

NAME
Egon
Erna

Wichtig: Vergleiche mit `NULL` liefern nie `TRUE`, sondern `UNKNOWN`. Daher müssen Prüfungen auf fehlende Werte immer mit `IS NULL` oder `IS NOT NULL` erfolgen.

2.4 Nutzung von Aggregatsfunktionen

SQL stellt verschiedene Funktionen für einfache statistische Auswertungen bereit. Aggregatsfunktionen fassen mehrere Zeilen zu einem einzigen Ergebniswert zusammen.

Wichtige Aggregatsfunktionen:

- `MAX()` – größter Wert einer Spalte
- `MIN()` – kleinster Wert einer Spalte
- `SUM()` – Summe aller Werte
- `AVG()` – Durchschnittswert
- `COUNT()` – Anzahl der Zeilen

Beispiele:

```sql
-- find the largest enclosure area
SELECT MAX(Gehege.Flaeche)
FROM Gehege;

-- find the smallest enclosure area with a custom column name
SELECT MIN(Gehege.Flaeche) AS KleinsteFlaeche
FROM Gehege;

-- calculate the total area of all enclosures
SELECT SUM(Gehege.Flaeche) AS Gesamtflaeche
FROM Gehege;

-- count the total number of animals
SELECT COUNT(*) AS Tieranzahl
FROM Tier;
```

Hinweis zu `COUNT`

- `COUNT(*)` zählt alle Zeilen, auch wenn einzelne Werte `NULL` sind.
- `COUNT(Attribut)` zählt nur Zeilen, in denen das Attribut **nicht NULL** ist.

Hinweis

Funkcja `COUNT()` służy do zliczania wierszy w tabeli lub w ramach grupy utworzonej przez `GROUP BY`. W zależności od argumentu działa w nieco inny sposób.

1. `COUNT(*)` Liczy **wszystkie wiersze**, niezależnie od tego, czy w kolumnach występują wartości `NULL`.

```
SELECT Gattung, COUNT(*) AS Tieranzahl
    FROM Tier
    GROUP BY Gattung;
```

Wynik pokazuje liczbę wszystkich zwierząt dla każdego gatunku. `COUNT(*)` nie wymaga podania kolumny jako argumentu — odnosi się do całych wierszy tabeli.

2. `COUNT(kolumna)` Liczy tylko te wiersze, w których dana kolumna nie ma wartości `NULL`.

```
SELECT COUNT(GName) AS AnzahlNamen
    FROM Gehege;
```

Jeśli w niektórych wierszach kolumna `GName` ma wartość `NULL`, nie zostaną one uwzględnione w zliczaniu.

3. `COUNT(DISTINCT kolumna)` Zlicza tylko **unikalne wartości** w kolumnie (również pomijając `NULL`).

```
SELECT COUNT(DISTINCT Gattung) AS AnzahlGattungen
    FROM Tier;
```

To zapytanie zwróci liczbę różnych gatunków w tabeli `Tier`.

Wniosek:

- `COUNT(*)` — gdy chcemy policzyć wszystkie rekordy (np. wszystkie zwierzęta),
- `COUNT(kolumna)` — gdy interesują nas tylko niepuste dane w konkretnej kolumnie,
- `COUNT(DISTINCT kolumna)` — gdy chcemy poznać liczbę różnych wartości (np. gatunków).

Beispiel: Äquivalente Zählung über ein bestimmtes Attribut

```sql
-- count all non-null values in the column "Gattung"
SELECT COUNT(Tier.Gattung) AS Tieranzahl
FROM Tier;
```

2.5 Anfragen über mehrere Tabellen

Bisherige SQL-Anfragen bezogen sich nur auf eine einzelne Tabelle. In realistischen Szenarien sind Informationen jedoch oft auf mehrere Tabellen verteilt. Um Daten aus diesen Tabellen gemeinsam auszuwerten, werden Verknüpfungen (Joins) verwendet. Dabei entsteht zunächst das Kreuzprodukt aller beteiligten Tabellen, das anschließend über Bedingungen eingeschränkt wird.

Beispiel: Ausgabe aller Tiere mit dem Namen ihres Geheges

```sql
-- select animal names with their corresponding enclosure names
SELECT Tier.Tname, Gehege.Gname
    FROM Tier, Gehege
    WHERE Tier.GNr = Gehege.GNr;
```

Erläuterung:

- Die `FROM`-Zeile enthält alle Tabellen, die für die Anfrage benötigt werden.
- Zunächst wird das Kreuzprodukt (*Cartesian Product*) der Tabellen gebildet.
- Über die `WHERE`-Bedingung wird dieses Ergebnis auf passende Kombinationen reduziert, z. B. entsprechend der Fremdschlüsselbeziehung.

Beispiel für das Kreuzprodukt:

```sql
-- create the full Cartesian product of both tables
SELECT *
    FROM Tier, Gehege;
```

Hinweis zum Kreuzprodukt

Das Kreuzprodukt enthält alle möglichen Kombinationen der Zeilen aus beiden Tabellen und führt häufig zu inhaltlich unsinnigen Ergebnissen. Daher ist es notwendig, die Beziehung zwischen den Tabellen mit einer `WHERE`-Bedingung einzuschränken.

Korrigierte Anfrage:

```
-- select only matching rows based on the foreign key relationship
SELECT *
    FROM Tier, Gehege
    WHERE Tier.GNr = Gehege.GNr;
```

Hinweis

Ein Kreuzprodukt mit einer Tabelle, die keine Einträge enthält, führt zu einer Ausgabe ohne Elemente. Das Kreuzprodukt verknüpft nämlich alle Einträge der ersten Tabelle mit allen Einträgen der zweiten Tabelle. Wenn eine der Tabellen leer ist, können folglich keine Kombinationen gebildet werden.

2.6 Łączenie tabel za pomocą `JOIN`

SQL umożliwia łączenie danych z wielu tabel poprzez operator `JOIN`. W przeciwieństwie do starszej składni, w której tabele były wymieniane po prostu w klauzuli `FROM`, podejście z użyciem `JOIN` jest bardziej czytelne i ułatwia zrozumienie relacji między tabelami.

Podstawowe rodzaje JOIN:

- `INNER JOIN` – zwraca tylko te wiersze, które mają dopasowanie w obu tabelach.
- `LEFT JOIN` – zwraca wszystkie wiersze z tabeli po lewej stronie, nawet jeśli brak dopasowania w prawej.
- `RIGHT JOIN` – zwraca wszystkie wiersze z tabeli po prawej stronie, nawet jeśli brak dopasowania w lewej.
- `FULL JOIN` – zwraca wszystkie wiersze z obu tabel (jeśli obsługiwane przez DBMS).

Wskazówka dotycząca JOIN

- Warunek po słowie kluczowym `ON` określa relację między tabelami — zwykle poprzez klucz główny i klucz obcy.
- `INNER JOIN` działa tak samo jak starsza metoda z wieloma tabelami i warunkiem `WHERE`.
- W przypadku `LEFT JOIN` lub `RIGHT JOIN` brakujące wartości zostaną zastąpione przez `NULL`.

2.6.1 Porównanie `INNER`, `LEFT` i `RIGHT JOIN`

Aby zrozumieć różnicę między poszczególnymi typami `JOIN`, rozważmy prosty przykład z dwiema tabelami. Tabela `Student` zawiera listę studentów, a tabela `Kurs` — listę kursów, do których zapisano wybranych studentów.

Tabele źródłowe

Student

ID	Name
1	Anna
2	Ben
3	Clara
4	David

Kurs

KursID	StudentID
1	1
2	1
3	2
4	5

1. `INNER JOIN` – tylko dopasowane rekordy

```
-- show only students who are registered in a course
SELECT Student.Name, Kurs.KursID
    FROM Student
    INNER JOIN Kurs ON Student.ID = Kurs.StudentID;
```

Wynik:

Name	KursID
Anna	1
Anna	2
Ben	3

2. `LEFT JOIN` – wszystkie rekordy z lewej tabeli

```
-- show all students, even if they are not registered in any course
SELECT Student.Name, Kurs.KursID
    FROM Student
    LEFT JOIN Kurs ON Student.ID = Kurs.StudentID;
```

Wynik:

Name	KursID
Anna	1
Anna	2
Ben	3
Clara	NULL
David	NULL

3. `RIGHT JOIN` – wszystkie rekordy z prawej tabeli

```
-- show all courses, even if no student is assigned
SELECT Student.Name, Kurs.KursID
    FROM Student
    RIGHT JOIN Kurs ON Student.ID = Kurs.StudentID;
```

Wynik:

Name	KursID
Anna	1
Anna	2
Ben	3
NULL	4

Opis

- `INNER JOIN` – pokazuje tylko wiersze z dopasowaniem po obu stronach.
- `LEFT JOIN` – zwraca wszystkie wiersze z tabeli po lewej stronie od `JOIN`, nawet jeśli nie mają dopasowania w tabeli po prawej.
- `RIGHT JOIN` – zwraca wszystkie wiersze z tabeli po prawej stronie od `JOIN`, nawet jeśli nie mają dopasowania w tabeli po lewej.

Różnica w praktyce:

- W zapytaniu `LEFT JOIN` – lewa tabela (**Student**) to ta po słowie `FROM`. Wszystkie jej rekordy są zachowane.
- W zapytaniu `RIGHT JOIN` – prawa tabela (**Kurs**) to ta po `JOIN`. Wszystkie jej rekordy są zachowane.

2.7 Gruppierung in einer Tabelle

Gehenge

Gnr	GName	Flaeche
1	Wald	20
2	Feld	10
3	Weide	9

Tier

Gnr	TName	Gattung
1	Laber	Bär
1	Sabber	Bär
2	Klopfer	Hase
3	Bunny	Hase
3	Runny	Hase
3	Hunny	Hase
2	Harald	Schaf
3	Walter	Schaf
3	Doerthe	Schaf

Art

Gattung	Min Fläche
Bär	8
Hase	2
Schaf	5

Ausgabe deer Anzahl der Tiere Je Gattung

```
SELECT Tier.Gattung, COUNT(*) Tieranzahl
    FROM Tier
    GROUP BY Tier.Gattung
```

3 `GROUP BY`, `HAVING` i `ORDER BY` w SQL

3.1 Motywacja

Zwykła klauzula `WHERE` pozwala filtrować pojedyncze wiersze, ale nie umożliwia tworzenia zbiorczych zestawień.

```
SELECT COUNT(*) AS Schafanzahl
    FROM Tier
    WHERE Tier.Gattung = 'Schaf';
```

Takie rozwiązanie jest niewygodne, ponieważ dla każdej wartości (`Gattung`) trzeba wykonać osobne zapytanie. Dzięki klauzuli `GROUP BY` można uzyskać podsumowanie dla wszystkich grup w jednym zapytaniu.

3.2 Podstawowa idea grupowania

`GROUP BY` dzieli dane w tabeli na grupy wierszy, które mają te same wartości określonych kolumn. Dla każdej grupy można następnie obliczyć wartości przy użyciu funkcji agregujących takich jak `COUNT` , `SUM` , `AVG` , `MIN` czy `MAX` .

```
SELECT Tier.Gattung, COUNT(*) AS Tieranzahl
    FROM Tier
    GROUP BY Tier.Gattung;
```

Etapy działania

1. Posortowanie danych według kolumny `Gattung`:
   ```
   SELECT * FROM Tier ORDER BY Tier.Gattung;
   ```
2. Utworzenie grup dla każdej wartości atrybutu (np. `Bär`, `Hase`, `Schaf`)
3. Zastosowanie funkcji agregującej `COUNT(*)` dla każdej grupy

3.3 Zasady stosowania `GROUP BY`

W klauzuli `SELECT` mogą znajdować się wyłącznie:

- kolumny wymienione w `GROUP BY`,
- lub funkcje agregujące.

```
SELECT Tier.GNr, Tier.Gattung, COUNT(*) AS Tieranzahl
FROM Tier
GROUP BY Tier.GNr, Tier.Gattung;
```

3.4 Klauzula `HAVING`

Klauzula `HAVING` służy do filtrowania **całych grup**, w przeciwieństwie do `WHERE`, która filtruje pojedyncze wiersze. Może zawierać warunki z funkcjami agregującymi.

Przykład 1: Numery wybiegów, w których znajduje się co najmniej 3 zwierzęta.

```
SELECT Tier.GNr
    FROM Tier
    GROUP BY Tier.GNr
    HAVING COUNT(*) >= 3;
```

Przykład 2: Gatunki inne niż `Schaf`, w których liczba zwierząt nie przekracza 3.

```
SELECT Tier.Gattung, COUNT(*) AS Tieranzahl
    FROM Tier
    GROUP BY Tier.Gattung
    HAVING Tier.Gattung <> 'Schaf' AND COUNT(*) <= 3;
```

3.5 Klauzula `ORDER BY`

Po grupowaniu można uporządkować wynik według określonego kryterium, np. liczby zwierząt:

```
SELECT Tier.Gattung, COUNT(*) AS Tieranzahl
    FROM Tier
    GROUP BY Tier.Gattung
    ORDER BY COUNT(*) DESC;
```

3.6 Grupowanie przy wielu tabelach

`GROUP BY` może być stosowane również po połączeniu tabel przy użyciu `JOIN` lub warunków w `WHERE`.

Przykład: Powierzchnia zajmowana przez zwierzęta w każdym wybiegu.

```
SELECT Gehege.GName, SUM(Art.MinFlaeche) AS Verbraucht
    FROM Gehege, Tier, Art
    WHERE Gehege.GNr = Tier.GNr
    AND Tier.Gattung = Art.Gattung
    GROUP BY Gehege.GName;
```

Przykład: Powierzchnia (rosnąco), którą zajmuje każdy gatunek, pod warunkiem, że w zoo znajduje się co najmniej 3 przedstawicieli tego gatunku:

```
SELECT Tier.Gattung, SUM(Art.MinFlaeche) AS Bedarf
    FROM Tier, Art
    WHERE Tier.Gattung = Art.Gattung
    GROUP BY Tier.Gattung
    HAVING COUNT(*) >= 3
    ORDER BY SUM(Art.MinFlaeche);
```

3.7 Kolejność wykonywania zapytania SQL

Klauzula	Kolejność	Opis
`FROM`	1	Wybór tabel (tworzenie iloczynu kartezjańskiego)
`WHERE`	2	Filtrowanie wierszy przed grupowaniem
`GROUP BY`	3	Tworzenie grup
`HAVING`	4	Filtrowanie grup
`ORDER BY`	5	Sortowanie wyników końcowych
`SELECT`	6	Wybór kolumn i funkcji agregujących do wyświetlenia

3.8 Strategia pisania zapytań z grupowaniem

1. Wybierz tabele i wpisz je w `FROM`.
2. Zdefiniuj warunki połączeń i filtrowania w `WHERE`.
3. Dodaj `ORDER BY`, aby sprawdzić dane przed grupowaniem.
4. Dodaj `GROUP BY` z odpowiednimi funkcjami agregującymi.
5. Użyj `HAVING`, jeśli chcesz odfiltrować całe grupy.
6. Na końcu określ kolejność wyników za pomocą `ORDER BY`.

—

4 Aufgabenblätter

1 Aufgabenblatt 1

1.1 Aufgabe 1

Der Ren-Operator wird benötigt, wenn ein Kreuzprodukt einer Tabelle mit sich selbst (Self-Join) gebildet werden muss.

Aufgaben:

ProzessNr	Name	VorgängerNr
1	Schneiden	-
2	Waschen	1
3	Biegen	2
4	Bohren	2
5	Malen	4

Gib die Aufgaben und deren Vorgänger aus.

$$\text{Proj}\Big(\text{Sel}(Aufgaben \times Ren(Aufgaben, A2),\ Aufgabe.VorgängerNr = A2.ProzessNr),\ \text{[Aufgabe.Name, A2.Name]}\Big)$$

1.2 Aufgabe 2

Relationen

Student

MatNr	Name
1	Meier
2	Meyer
3	Maier

Gericht

GNr	Name	Art
1	Pizza	Haupt
2	TomatenSuppe	Vor
3	Schnitzel	Haupt
4	Reis	Beilage
5	Pudding	Nach

Bewertung

MatrNr	GNr	Sterne
1	2	3
1	4	2
2	1	4
3	3	3

1. Geben Sie alle Arten von Gerichten aus.

$$\text{Proj}(\textit{Gericht}, [\text{Art}])$$

Ergebnisstabelle

Art
Haupt
Vor
Beilage
Nach

2. Geben Sie die Namen aller Hauptgerichte (mit der Art „Haupt“) aus.

$$\text{Proj}\Big(\text{Sel}(\textit{Gericht}, [\textit{Art} = \text{'Haupt'}]), [\text{Name}]\Big)$$

Ergebnisstabelle

Name
Pizza
Schnitzel

3. Geben Sie eine Liste aller einzelnen Bewertungen aus (Ausgabe: Name des Gerichts, Sterne).

$$\text{Proj}\Big(\text{Sel}(\textit{Gericht} \times \textit{Bewertung}, \textit{Gericht.GNr} = \textit{Bewertung.GNr}), [\text{Name, Sterne}]\Big)$$

Ergebnisstabelle

Name	Sterne
Pizza	4
TomatenSuppe	3
Schnitzel	3
Reis	2

4. Geben Sie die Namen aller Gerichte aus, die der Student Meier bewertet hat.

$$\text{Proj}\Big(\text{Sel}(\textit{Student} \times \textit{Gericht} \times \textit{Bewertung}, \textit{Student.MatNr} = \textit{Bewertung.MatNr} \text{ AND } \textit{Bewertung.GNr} = \textit{Gericht.GNr} \text{ AND } \textit{Student.Name} = \text{'Meier'}), [\text{Gericht.Name}]\Big)$$

Ergebnisstabelle

Name
TomatenSuppe
Reis

5. Geben Sie alle Bewertungen aus (Name Student, Name Gericht, Sterne), die mindestens vier Sterne haben.

$$\text{Proj}\Big(\text{Sel}(\mathit{Student} \times \mathit{Gericht} \times \mathit{Bewertung},\ \mathit{Student.MatNr} = \mathit{Bewertung.MatrNr}\ \text{AND}\ \mathit{Bewertung.GNr} = \mathit{Gericht.GNr}\ \text{AND}\ \mathit{Bewertung.Sterne} >= 4),\ [\text{Student.Name, Gericht.Name, Sterne}]\Big)$$

Ergebnisstabelle

Name Student	Name Gericht	Sterne
Meyer	Pizza	4

6. Geben Sie aus, welche Studierenden das Schnitzel bewertet haben.

$$\text{Proj}\Big(\text{Sel}(\mathit{Student} \times \mathit{Gericht} \times \mathit{Bewertung},\ \text{Student.MatNr} = \text{Bewertung.MatNr AND Bewertung.GNr} = \text{Gericht.GNr AND Gericht.Name} = \text{'Schnitzel'}),\ [\text{Student.Name}]\Big)$$

Ergebnisstabelle

Student Name
Maier

7. Geben Sie aus, welcher Studierende mindestens zwei Bewertungen abgegeben hat.

$$\text{Proj}\Big(\text{Sel}(\mathit{Student} \times \mathit{Bewertung} \times \text{Ren}(\mathit{Bewertung},\ B2),\ \mathit{Student.MatrNr} = \mathit{Bewertung.MatrNr}\ \text{AND}\ \mathit{Student.MatrNr} = B2.\mathit{MatrNr}\ \text{AND}\ \mathit{Bewertung.GNr} <> B2.\mathit{GNr}),\ [\mathit{Student.Name}]\Big)$$

Ergebnisstabelle

Student Name
Meier

2 Aufgabenblatt 2

Relationen

Student

MatNr	Name
1	Meier
2	Meyer
3	Maier

Klausur

KNr	Name	Datum	Zeit
1	Java 1	2024-01-14	10:00:00
2	Einführung Inf.	2024-01-16	08:00:00
3	Mathematik 1	2024-01-20	13:00:00
4	Medieninformatik	2024-01-20	08:00:00
5	Audio/Video	2024-01-28	15:30:00

Bewertung

MatrNr	KNr	Versuch
1	2	1
1	4	2
2	1	2
3	3	3

2.1 Aufgabe 2.1

Listing 4.1: Erstellen der Tabellen STUDENT, KLAUSUR und ANMELDUNG

```
CREATE TABLE STUDENT(
    MatrNr INTEGER,
    Name VARCHAR(30),
    PRIMARY KEY (MatrNr)
);

CREATE TABLE KLAUSUR(
    KNr INTEGER,
    Name VARCHAR(25),
    Datum DATE,
    Zeit TIME,
    PRIMARY KEY (KNr)
);

CREATE TABLE ANMELDUNG(
    MatrNr INTEGER,
    KNr INTEGER,
    Versuch INTEGER,
    PRIMARY KEY (MatrNr, KNr)
    CONSTRAINT FK_MatrNr
        FOREIGN KEY (MatrNr)
        REFERENCES STUDENT(MatrNr),
    CONSTRAINT FK_KNr
```

```
        FOREIGN KEY (KNr)
        REFERENCES KLAUSUR(KNr)
);
```

Listing 4.2: Einfügen von Datensätzen in die Tabellen STUDENT, KLAUSUR und ANMELDUNG

```
INSERT INTO STUDENT VALUES(1,'Meier');
INSERT INTO STUDENT VALUES(2,'Meyer');
INSERT INTO STUDENT VALUES(3,'Maier');

INSERT INTO KLAUSUR VALUES(1,'Java 1','2024-01-14', '10:00:00');
INSERT INTO KLAUSUR VALUES(2,'Einfuhrung Inf.','2024-01-16', '08:00:00');
INSERT INTO KLAUSUR VALUES(3,'Mathematik 1','2024-01-20', '13:00:00');
INSERT INTO KLAUSUR VALUES(4,'Medieninformatik','2024-01-20', '08:00:00');
INSERT INTO KLAUSUR VALUES(5,'Audio/Video','2024-01-28', '15:30:00');

INSERT INTO ANMELDUNG VALUES(1,2,1);
INSERT INTO ANMELDUNG VALUES(1,4,2);
INSERT INTO ANMELDUNG VALUES(2,1,2);
INSERT INTO ANMELDUNG VALUES(3,3,3);
```

2.2 Aufgabe 2.2

Listing 4.3: Einfügen von Datensätzen in die Tabellen STUDENT, KLAUSUR und ANMELDUNG

```
-- 1. Geben Sie die Namen aller Studierenden aus.
SELECT klausur.Name FROM student;

-- 2. Geben Sie die Namen aller Klausuren aus, die um 08:00 Uhr geschrieben
   werden.
SELECT klausur.name FROM klausur WHERE zeit = '08:00:00';

-- 3. Geben Sie eine Liste aller Erstanmeldungen (nur 1. Versuch) fuer eine
   Klausur aus (Ausgabe: Name der Klausur, Name des Studierenden).
SELECT klausur.name, student.name FROM klausur, student, anmeldung AND klausur.
   knr = student.knr AND student.matrnr = anmeldung.matrnr WHERE anmeldung.
   versuch = 1;
```

```
-- 4. Geben Sie die Namen aller Klausuren aus, feur die sich die Studentin "
    Meier" angemeldet hat.
SELECT klausur.name FROM klausur, student, anmeldung WHERE student.name Like '
    Meier' AND klausur.knr = anmeldung.knr AND student.matrnr = anmeldung.
    matrnr;

    -- oder

SELECT klausur.name FROM student, klausur, anmeldung WHERE anmeldung.matrnr =
    student.matrnr AND anmeldung.knr = klausur.knr AND student.name='Meier';

-- 5. Geben Sie die Namen aller Studierenden aus, die mindestens zwei Klausuren
     im letzten Versuch (3. Versuch) schreiben.
SELECT DISTINCT student.name
    FROM student, anmeldung a1, anmeldung a2
    WHERE student.matrnr = a1.matrnr
        AND student.matrnr = a2.matrnr
        AND a1.knr <> a2.knr
        AND a1.versuch >= 3
        AND a2.versuch >= 3;

    -- oder

SELECT DISTINCT name FROM student
    JOIN anmeldung AS a1 ON a1.matrnr=student.matrnr
    JOIN anmeldung AS a2 ON a2.matrnr=student.matrnr
    WHERE a1.versuch >= 3
        AND a2.versuch >= 3
        AND a1.knr <> a2.knr;
```

3 Aufgabenblatt 3

3.0.1 Geben Sie aus, wie viele Module in der Datenbank gespeichert sind.

```
SELECT COUNT(*)
    FROM modul
```

3.0.2 Geben Sie alle Module (Name) aus, die weniger als 4 ECTS-Punkte haben.

```
SELECT name
    FROM modul
    WHERE ects < 4
```

3.0.3 Geben Sie alle Module aus, in denen eine Klausur als Prüfung möglich ist.

```
SELECT modul.name
    FROM modul
    WHERE modul.pruefung like '%Klausur%'
```

3.0.4 Geben Sie die Namen der Module aus, bei denen als Modulverantwortlicher "F. Rump" angegeben ist.

```
SELECT DISTINCT modul.name
    FROM modul, person
    WHERE modul.pid = person.pid AND person.name = 'F. Rump'

        -- oder

SELECT DISTINCT modul.name
    FROM modul
    JOIN person ON modul.pid = person.pid
    WHERE person.name = 'F. Rump'
```

3.0.5 eben Sie alle Studiengänge mit den zugehörigen Zertifikaten aus (Ausgabe: Studiengang, Zertifikat).

```
SELECT DISTINCT studiengang.name, zertifikat.name
    FROM zertifikatzuordnung
    JOIN zertifikat ON zertifikat.zid = zertifikatzuordnung.zid
    JOIN studiengang ON studiengang.sid = zertifikatzuordnung.sid

    -- oder

SELECT DISTINCT studiengang.name Modulname, zertifikat.name Zertifikatname
    FROM zertifikat, zertifikatzuordnung, studiengang
    WHERE studiengang.sid = zertifikatzuordnung.sid
    AND zertifikatzuordnung.zid = zertifikat.zid
```

3.0.6 Geben Sie alle Lehrenden und die Veranstaltungen aus, die sie unterrichten.

```
SELECT DISTINCT person.name, veranstaltung.name
    FROM lehrende
    JOIN person ON person.pid = lehrende.pid
    JOIN veranstaltung ON veranstaltung.vid = lehrende.vid

    -- oder

SELECT DISTINCT person.name, veranstaltung.name
    FROM lehrende, person, veranstaltung
    WHERE lehrende.pid = person.pid AND lehrende.vid = veranstaltung.vid
```

3.0.7 Geben Sie alle Module aus, sofern die enthaltenen Veranstaltung in Summe mehr als vier SWS haben

```
SELECT DISTINCT modul.name
    FROM modul, veranstaltung
    WHERE modul.mid = veranstaltung.mid
    AND veranstaltung.sws > 4
```

3.0.8 Geben Sie nur die Veranstaltungen aus, denen mehrere Lehrende zugeordnet sind.

```
SELECT veranstaltung.name
    FROM veranstaltung, lehrende
    WHERE veranstaltung.vid = lehrende.vid
    GROUP BY veranstaltung.name
    HAVING COUNT(DISTINCT lehrende.pid) > 1
```

3.0.9 Geben Sie alle Studiengänge absteigend nach der Anzahl der "reinen" Wahlpflichtmodule, die somit keinem Zertifikat zugeordnet sind, aus.

```
SELECT studiengang.name, COUNT(*) count
    FROM studiengang, modulzuordnung
    WHERE studiengang.sid = modulzuordnung.sid
    AND modulzuordnung.semester IS NULL
    GROUP BY studiengang.name
    ORDER BY COUNT(*) desc
```