Prof. Dr.-Ing. Johannes Schildgen
johannes.schildgen@oth-regensburg.de

Datenbanken

Kapitel 4: SQL

OTH REGENSBURG

33, 36

SQL...

- ... ist der Nachfolger von SEQUEL (Structured English Query Language),
- ... wurde in den 1970er-Jahren entworfen,
- ... ist ein eigenständiger Begriff,
- ... wird manchmal als Abkürzung für Structured Query Language gesehen,
- ... ist eine standardisierte Anfragesprache für strukturierte Datenbanken,
- ... ist größtenteils unabhängig vom verwendeten relationalen DBMS (RDBMS).

SQL ist eine leicht zu erlernende aber trotzdem sehr mächtige Anfragesprache für alle Aufgaben bei der Arbeit mit relationalen Datenbanken. Gelegentliche Benutzer können schnell simple Abfragen schreiben, erfahrene Benutzer können sich fortgeschrittenen Sprachkonstrukten bedienen. Der SQL-Standard ist ein ISO-Standard und wird ständig weiterentwickelt (SQL92, ..., SQL:2016). Dadurch, dass die Sprache standardisiert ist, verwenden alle populären relationalen Datenbankmanagementsysteme als Anfragesprache SQL. Dies macht Anwendungen portabel (Wechsel zu anderem DBMS) und es reduziert den Lernaufwand für die Anwender.

Populäre RDBMS

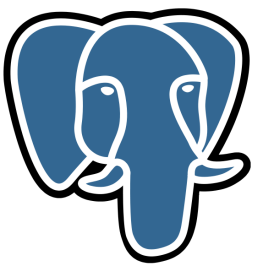
Quelle: db-engines.com/de/ranking

1. Oracle
2. MySQL
3. Microsoft SQL Server
4. PostgreSQL ← verwenden wir im Rahmen dieser Vorlesung
5. IBM DB2
6. SQLite
7. MariaDB

Die hier dargestellte Liste ist geordnet nach der Popularität der relationalen DBMS. Oracle, SQL Server und DB2 sind kommerzielle Systeme, MySQL, PostgreSQL, SQLite und MariaDB sind open source. MySQL wurde 2010 von Oracle aufgekauft und es hat sich ein zu MySQL kompatibler Fork MariaDB entwickelt. Alle hier dargestellten Systeme außer SQLite laufen als Serverprozess, sodass sich Anwendungen mit dem Datenbank-Server verbinden können, um Anfragen auf diesen zu stellen.

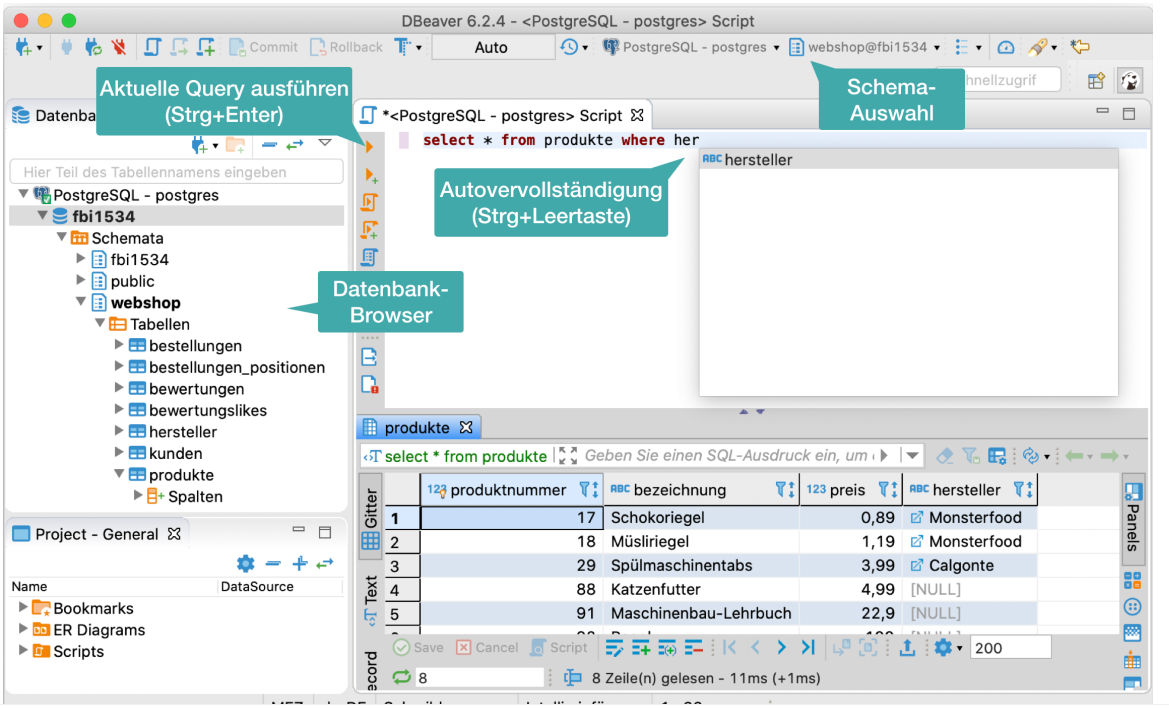
PostgreSQL

- "The world's most advanced open source database"
- Einfach zu installieren, viele Features, Transaktionen, gute Dokumentation, ...
- SQL-Clients: psql, pgAdmin3, pgAdmin4



psql ist ein Konsolen-Client, um z. B. in einer SSH-Shell direkt SQL-Befehle an eine PostgreSQL-Datenbank zu schicken und das Ergebnis zu betrachten. pgAdmin ist ein GUI-Client für verschiedene Betriebssysteme.

DBeaver



Wir verwenden im Rahmen dieser Vorlesung den kostenlosen universellen SQL-Client DBeaver.

37

Sprachkomponenten

DDL: Data Definition Language

- Definition des Datenbankschemas (Metadaten)
- `CREATE TABLE`, `DROP TABLE`, `ALTER TABLE`, `CREATE VIEW`, ...

DML: Data Manipulation Language

- Einfügen, Ändern, Löschen und Lesen von Daten
- `INSERT`, `UPDATE`, `DELETE`, `SELECT`, ...

DCL: Data Control Language

- Benutzer-, Rollen- und Rechteverwaltung
- `CREATE USER`, `CREATE ROLE`, `GRANT`, `REVOKE`, ...

SELECT gehört zur DML, da die Daten für den Zeitpunkt der Anfragestellung manipuliert werden. Der Benutzer sieht eine modifizierte Form der Daten. Natürlich wird diese Änderung anders als z. B. bei UPDATE nicht persistent in der Datenbank gespeichert.

SQL ausprobieren

SQL Island

- Lernspiel, welches keine Vorkenntnisse voraussetzt.
- sql-island.de

SQL Fiddle, DB Fiddle, Rextester

- Tabellen erstellen, befüllen und abfragen im Browser
- sqlfiddle.com, dbfiddle.uk, rextester.com

Alle hier dargestellten Anwendungen sind SQL-Spielwiesen, die ohne Anmeldung direkt im Browser verwendet werden können. Das Lernspiel SQL Island hat eine Handlung, man muss von einer Insel entkommen. Die Tabellen dazu sind bereits angelegt und mit Daten gefüllt. Bei den anderen Tools muss man selbst Tabellen anlegen und befüllen. Man hat verschiedene DBMS zur Auswahl.

7

Sprachelemente

```
SELECT email FROM kunden WHERE name = 'Ute';
```

SQL-Schlüsselworte (Keywords)

- Groß-/Kleinschreibung egal
- SELECT, FROM, WHERE, GROUP, BY, INSERT, CREATE, USER, AS, ...

Identifikatoren (Namen von Tabellen, Spalten, etc.)

- Reguläre: case-insensitive; Begrenzte: case-sensitive
- Beispiele: email, kunden, name, "user", "Kunden", "A B"

Literale (Werte eines Datentyps)

- Groß-/Kleinschreibung nicht egal!
- Beispiele: 'Ute', 'Otto''s Imbiss', 5, 2.99, NULL, '2020-11-18'

Merkregel: Bei allem, was man in Anführungszeichen schreibt, kommt es auf die korrekte Groß-/Kleinschreibung an. Bei regulären Identifikatoren sind reservierte SQL-Keywords und Sonderzeichen (außer _) verboten, bei begrenzten Identifikatoren ist dies erlaubt. In MySQL werden begrenzte Identifikatoren in `Apostrophe` geschrieben, bei PostgreSQL in "Anführungszeichen".

8

Kommentare

```
SELECT * FROM t -- Ein Doppelminus leitet einen Kommentar ein
```

```
SELECT * FROM t
--WHERE x = 5
```

```
SELECT kundennummer /*, email*/ FROM kunden
```

Alles, was in einer Zeile auf ein `--` folgt, ist ein Kommentar und gehört nicht mehr zur Anfrage. Dies ist auch praktisch, um vorübergehend gewisse Teile einer Anfrage wegzulassen und später wieder zu entkommentieren. Die Syntax `/* ... */` ermöglicht Kommentare innerhalb einer Zeile und Kommentare, die mehrere Zeilen umfassen.

DDL: CREATE, ALTER, DROP

44

Anlegen, Verändern und Entfernen von *Datenbankobjekten* (Tabellen, Sichten, Funktionen, ...)

```
CREATE TABLE personen (pid INT, name VARCHAR(500), geboren DATE);
```

CREATE TABLE

- Name der zu erstellenden Tabelle
- Spalten in Klammern, Komma-getrennt
- Zu jeder Spalte: Name, Datentyp, evtl. Spaltenoptionen

Das Semikolon am Ende einer Anfrage lassen wir ab sofort einfach weg. Es ist lediglich wichtig, wenn man in einem SQL-Editor mehrere Anfragen formuliert, um diese voneinander zu treffen.

Datentypen

Datentyp	Alias	Beschreibung	Beispiel
INTEGER	INT	Ganze Zahl	-217
DECIMAL(p,s)	NUMERIC	Zahl mit p Stellen, davon s nach dem Dezimalpunkt	DECIMAL(5,2) -149.99
DOUBLE	REAL	Fließkommazahl (nicht exakt)	-15.127E-2
CHARACTER(l)	CHAR	Zeichenkette mit fixer Länge l	CHAR(3) 'AB '
CHARACTER VARYING(l)	VARCHAR	Zeichenkette mit max. Länge l	VARCHAR(3) 'AB'
DATE		Datum (Jahr, Monat, Tag)	'2020-11-18'

Diese Liste ist nicht vollständig. Es gibt noch viele weitere Datentypen in SQL und jedes DBMS bringt weitere Typen mit. Die hier dargestellte Fließkommazahl ist $-15{,}127 \cdot 10^{-2}$.

CREATE TABLE - Spaltenoptionen


```
CREATE TABLE kunden (
 kundennummer INT PRIMARY KEY, name VARCHAR(100) NOT NULL,
 email VARCHAR(500) CHECK (email LIKE '%@%') UNIQUE,
 passwort CHAR(32), land VARCHAR(100) DEFAULT 'Deutschland',
 geworben_von INT REFERENCES kunden(kundennummer));
```

Spaltenoptionen:

- PRIMARY KEY: Primärschlüssel
- NOT NULL: Verbot von NULL-Werten
- CHECK: Bedingung für akzeptierte Werte
- UNIQUE: Eindeutigkeit
- DEFAULT: Standardwert
- REFERENCES: Fremdschlüsselreferenz

Hinter den Namen und Datentypen einer Spalte können Spaltenconstraints geschrieben werden. Dies sind Integritätsbedingungen, die stets erfüllt sein müssen. Beispielsweise verbietet ein UNIQUE-Constraint, dass in der Spalte in zwei Zeilen der gleiche Wert vorkommt. Das im Beispiel verwendete CHECK-Constraint garantiert, dass in jeder E-Mail-Adresse ein @-Zeichen vorkommen muss.

NULL / NOT NULL

NULL (Standard)

```
CREATE TABLE kunden (/* ... */, name VARCHAR(100) NULL)
```

NULL-Werte sind in der Spalte erlaubt.

NOT NULL

```
CREATE TABLE kunden (/* ... */, name VARCHAR(100) NOT NULL)
```

NULL-Werte sind in der Spalte verboten.

In den meisten DBMS ist NULL die Standard-Eigenschaft von Spalten, das heißt, man darf NULL-Werte in die Spalte eintragen. Will man dies verhindert, deklariert man die Spalte als NOT NULL. Im hier gezeigten unteren Beispiel ist das Attribut "Name" also ein Pflichtfeld.

UNIQUE-Constraint

```
CREATE TABLE kunden (/* ... */, email VARCHAR(500) UNIQUE)
```

oder:

```
CREATE TABLE kunden (/* .... */ email VARCHAR(500),
                     UNIQUE(email))
```

Spezifiziert man hinter einer Spalte die Eigenschaft UNIQUE, darf es in dieser Spalte keine doppelten Werte gegen. In unserer Kundentabelle dürfen also keine zwei Kunden die gleiche E-Mail-Adresse haben. Die untere Notation ist nötig, wenn das UNIQUE-Constraint aus mehreren Spalten besteht, also wenn die Kombination aus mehreren Spaltenwerten eindeutig sein soll. Anders als bei Primärschlüsseln (in einer Tabelle darf höchstens einen geben), können auf einer Tabelle beliebig viele UNIQUE-Constraints definiert werden.

Primärschlüssel-Constraint

```
CREATE TABLE kunden (kundennummer INT PRIMARY KEY, ...)
```

oder:

```
CREATE TABLE kunden (kundennummer INT, ...,
                     PRIMARY KEY(kundennummer))
```

Primärschlüssel sind UNIQUE und NOT NULL.

Wenn der Primärschlüssel nur aus einer Spalte besteht, kann man `PRIMARY KEY` direkt hinter die Spalte schreiben. Primärschlüssel sind UNIQUE, d. h. es gibt keine Duplikate in der Tabelle in diesen Spalten. Außerdem sind Primärschlüsselattribute alle NOT NULL, d. h. in ihnen dürfen keine NULL-Werte vorkommen.

Fremdschlüssel-Constraints

```
CREATE TABLE kunden (/* ... */,
 geworben_von INT REFERENCES kunden(kundennummer));
```

oder:

```
CREATE TABLE kunden (/* ... */,
 geworben_von INT,
 FOREIGN KEY(geworben_von) REFERENCES kunden(kundennummer));
```

Besteht der Fremdschlüssel nur aus einer Spalte, kann `REFERENCES tabelle(spalte)` direkt hinter die Spalte geschrieben werden. Ansonsten muss die unten stehende `FOREIGN KEY`-Schreibweise verwendet werden.

Zusammengesetzte Primär-/ Fremdschlüssel

46, 48

```sql
CREATE TABLE bewertungen (
 kundennummer INT REFERENCES kunden(kundennummer),
 produktnummer INT REFERENCES produkte(produktnummer),
 sterne INT DEFAULT 5 CHECK(sterne BETWEEN 1 AND 5),
 bewertungstext VARCHAR(100000),
 PRIMARY KEY(kundennummer, produktnummer));
```

```sql
CREATE TABLE bewertungslikes (
 liker INT REFERENCES kunden(kundennummer),
 kundennummer INT, produktnummer INT,
 PRIMARY KEY(liker, kundennummer, produktnummer),
 FOREIGN KEY(kundennummer, produktnummer)
  REFERENCES bewertungen(kundennummer, produktnummer));
```

Wenn der Primärschlüssel zusammengesetzt ist aus mehreren Spalten, ist es nicht mehr möglich, einfach PRIMARY KEY in die Spaltenoptionen zu schreiben. Stattdessen muss das PRIMARY-KEY-Constraint unten drunter geschrieben werden. Der Grund dafür ist, dass es nicht mehrere Primärschlüssel geben kann. Eine Tabelle hat immer nur einen Primärschlüssel, dieser kann jedoch zusammengesetzt sein aus mehreren Spalten. Das gleiche gilt für FOREIGN-KEY-Constraints für zusammengesetzte Fremdschlüssel.

Referentielle Aktionen

49

```sql
CREATE TABLE produkte (
 produktnummer INT PRIMARY KEY,
 bezeichnung VARCHAR(100) NOT NULL, preis DECIMAL(9,2),
 hersteller VARCHAR(50) REFERENCES hersteller(firma)
                        ON DELETE SET NULL ON UPDATE CASCADE);
```

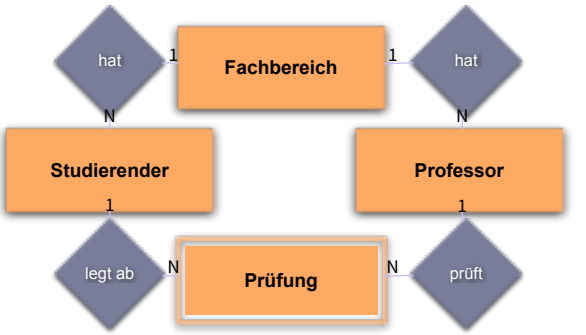
Was soll passieren, wenn ein Hersteller gelöscht wird, von dem es noch Produkte gibt? (Analog dazu: Was passiert, wenn man ihn umbenennt? - `ON UPDATE`)

- **`ON DELETE CASCADE`**: Alle Produkte auch löschen
- **`ON DELETE SET NULL / SET DEFAULT`**: NULL / Default-Wert setzen
- **`ON DELETE RESTRICT`**: Löschen verbieten
- **`ON DELETE NO ACTION`** (Standardverhalten): erst mal abwarten

Unter einer Aktion versteht man den Dann-Teil einer Wenn-Dann-Regel. Wenn man einen Hersteller löscht, dann... Im oben gezeigten Beispiel wird in einem solchen Fall wegen `ON DELETE SET NULL` der Wert der Hersteller-Spalte in der Produkte-Tabelle auf NULL gesetzt. Das `ON UPDATE CASCADE` sorgt dafür, dass bei einer Umbenennung im Primärschlüsselattribut des Herstellers (Firma) diese Umbenennung auch bei den Produkten dieses Herstellers erfolgt.

ON DELETE NO ACTION

Fachbereich (FBName)
Studierende (MatrNr, Name, FBName)
Professoren (ProfNr, Name, FBName)
Prüfungen (MatrNr, ProfNr, Fach, Versuch, Note)



Studierende(FBName) REFERENCES Fachbereich(FBName) ON DELETE CASCADE
Professoren(FBName) REFERENCES Fachbereich(FBName) ON DELETE CASCADE
Prüfungen(MatrNr) REFERENCES Studierende(MatrNr) ON DELETE CASCADE
Prüfungen(ProfNr) REFERENCES Professoren(MatrNr)

Fachbereich wird gelöscht ⇒ alle Studierenden in diesem werden gelöscht ⇒ Die Prüfungen der Studierenden werden gelöscht. Professoren werden gelöscht. OK!

Würde statt NO ACTION hier RESTRICT verwendet werden, kann es passieren, dass zuerst versucht wird, Professoren des Fachbereichs zu löschen. Dies wird jedoch abgelehnt, weil noch Prüfungen von diesem existieren. Würden zuerst die Studenten gelöscht, funktioniert das Löschen des Fachbereichs. Da das Verhalten in dem Fall nicht deterministisch ist, spricht man hier von einem *unsicheren Schema*.

DEFAULT-Werte für Spalten

```
CREATE TABLE kunden (/* ... */,
                     land VARCHAR(100) DEFAULT 'Deutschland');
```

Wird beim Einfügen in die Tabelle der Wert für eine Spalte nicht gesetzt, wird der DEFAULT-Wert dort eingetragen (standardmäßig NULL).

Später beim `INSERT`-Kommando wird gezeigt, dass man nicht zwangsweise alle Spaltenwerte beim Einfügen einer neuen Zeile setzen muss. Die DEFAULT-Option legt für solche Fälle den Standardwert fest. Gibt es keinen definierten DEFAULT-Wert und wird der Wert nicht gesetzt, erhält die Zeile in der entsprechenden Spalte den Wert NULL. Ist auf der Spalte ein `NOT NULL`-Constraint definiert, kommt ein Fehler. Vor solchen Fehlern kann dann ein DEFAULT-Wert schützen. Wird mittels `ALTER TABLE ... ADD COLUMN` nachträglich eine neue Spalte mit DEFAULT-Wert einer existierenden Tabelle hinzugefügt, erhalten alle bisherigen Zeilen in der neuen Spalte diesen DEFAULT-Wert.

Sequenzen / Autoincrement

Primärschlüsselwerte (z. B. IDs) automatisch erzeugen lassen.

In MySQL:

```
CREATE TABLE kunden(kundennr INT PRIMARY KEY AUTO_INCREMENT, ...);
```

In PostgreSQL (aber auch MySQL):

```
CREATE TABLE kunden(kundennr SERIAL PRIMARY KEY  , ...);
```

In verschiedenen DBMS gibt es die Spaltenoption `AUTO_INCREMENT` (MySQL), `AUTOINCREMENT` (SQLite), `GENERATED BY DEFAULT AS IDENTITY` (SQL Standard, Oracle, DB2), `IDENTITY` (Microsoft SQL Server), o. ä. Spalten mit dieser Option müssen einen nummerischen Datentypen haben. Eingefügte Zeilen erhalten standardmäßig eine automatisch generierte laufende Nummer. Problematisch kann es werden, wenn nan beim Einfügen auch selbst einen Wert setzt, statt ihn automatisch generieren zu lassen. Der SQL Standard und DB2 unterstützen `INTEGER GENERATED ALWAYS AS IDENTITY`, damit immer (nicht nur als Default) eine automatische ID erzeugt wird.

CHECK-Constraint


```
CREATE TABLE kunden (/* ... */,
 email VARCHAR(500) CHECK (email LIKE '%@%'));
```

```
CREATE TABLE bewertungen (/* ... */,
 sterne INT CHECK (sterne BETWEEN 1 AND 5));
```

```
CREATE TABLE personen (/* ... */,
 geburtsdatum DATE,
 hochzeitsdatum DATE CHECK (hochzeitsdatum > geburtsdatum));
```

Mit einem CHECK-Constraint lassen sich benutzerdefinierte Spalten-Constraints definieren. In Klammern hinter dem Stichwort CHECK steht ein beliebiger boolescher Ausdruck, der stets wahr sein muss. Beim Einfügen und Ändern von Zeilen überprüft das DBMS die Bedingung und lehnt die entsprechende Operation ab, wenn das CHECK-Constraint verletzt würde. In den gezeigten Beispielen muss in der E-Mail-Spalte immer ein @-Zeichen vorkommen, der Wertebereich der Sterne-Spalte wird eingeschränkt auf ganze Zahlen zwischen 1 und 5, und das Hochzeitsdatum muss immer echt später sein als das Geburtsdatum einer Person.

CREATE TABLE LIKE / AS

CREATE TABLE

```
CREATE TABLE pkw (bez VARCHAR(50) PRIMARY KEY, leistung INT);
```

CREATE TABLE LIKE

```
CREATE TABLE lkw (LIKE pkw);
```

CREATE TABLE AS

```
CREATE TABLE schrottkarren AS
 SELECT * FROM pkw WHERE leistung < 60;
```

Alle drei hier gezeigten Befehle erstellen eine Tabelle. Bei der ersten gibt man die Spalten der Tabelle an, beim `CREATE TABLE LIKE` wird eine Tabelle mit den gleichen Spalten erstellt wie eine schon existierende Tabelle und `CREATE TABLE AS` speichert das Ergebnis einer SELECT-Anfrage in eine neu zu erstellende Tabelle. In den ersten beiden Kommandos ist die neue Tabelle leer, beim dritten Kommando können Daten in der neuen Tabelle sein. Mit `CREATE TABLE T2 AS SELECT * FROM T1` kann man eine Tabelle komplett mit Struktur und Daten kopieren.

ALTER / DROP

ALTER: Verändern von Datenbankobjekten

```
ALTER TABLE bewertungen ADD COLUMN zeitstempel TIMESTAMP;
```

DROP: Entfernen von Datenbankobjekten

```
DROP TABLE bewertungen;
```

Die DDL-Befehle ALTER und DROP wirken sich genau wie CREATE auf den DB-Katalog aus. Das hier gezeigte ALTER TABLE ... ADD COLUMN-Kommando fügt eine neue Spalte namens "zeitstempel" vom Typ TIMESTAMP der Bewertungstabelle hinzu. Das gezeigte DROP-Kommando würde die Bewertungstabelle entfernen. Das heißt, es sind nicht nur alle Daten dieser Tabelle weg, sondern die ganze Tabelle existiert nicht mehr. Das DROP-Kommando hier würde aber abgelehnt werden, da es noch eine Tabelle Bewertungslikes gibt, bei der auf die Bewertungstabelle Fremdschlüsselreferenzen definiert sind. Man müsste also zunächst diese Tabelle oder zumindest die Fremdschlüssel droppen.

DML: INSERT, UPDATE, DELETE, ...

Einfügen, ändern, löschen, abrufen von Daten.

```
INSERT INTO produkte(produktnr, bezeichnung, preis, hersteller)
VALUES (88, 'Katzenfutter', 4.99, NULL)
```

```
UPDATE produkte SET preis = 5.99 WHERE produktnr = 88
```

```
DELETE FROM produkte WHERE produktnr = 88
```

Hier wird das Produkt Katzenfutter mit der Produktnummer 88 eingefügt, dann der Preis auf 5,99 EUR erhölt und im dritten Kommando wird die Zeile wieder gelöscht.

53

INSERT

```
INSERT INTO produkte(produktnr, bezeichnung, preis, hersteller)
VALUES (88, 'Katzenfutter', 4.99, NULL)
```

Nur ein Teil der Spalten setzen:

```
INSERT INTO produkte(preis, bezeichnung)
VALUES (4.99, 'Katzenfutter')
```

Spaltennamen weglassen:

```
INSERT INTO produkte VALUES (88, 'Katzenfutter', 4.99, NULL)
```

Nachteile: Schwerer verständlich zu lesen, man muss auf die korrekte Reihenfolge achten, man darf keine Spalte vergessen, in der Zukunft könnten sich die Spalten

Wenn nur ein Teil der Tabellenspalten gesetzt werden, werden die anderen Spalten auf ihre DEFAULT-Werte, den nächsten AUTO_INCREMENT / SERIAL-Wert oder auf NULL gesetzt. Lässt man die Spaltennamen vor dem Stichwort VALUES weg, müssen alle Spalten der Tabelle in der richtigen Reihenfolge gesetzt werden. Wenn in der Zukunft eine neue Spalte in die Produkttabelle hinzugefügt wird, würde das untere Kommando nicht mehr funktionieren.

54

UPDATE

```
UPDATE produkte SET preis = 5.99 WHERE produktnr = 88
```

```
UPDATE produkte
SET preis = 5.99, bezeichnung = 'Spezial-Katzenfutter'
WHERE produktnr = 88
```

```
UPDATE produkte SET preis = preis+1
```

```
UPDATE produkte SET preis = preis * 1.1
```

Das UPDATE-Kommando besteht aus einem SET-Teil und einem WHERE-Teil. Im SET werden Komma-getrennte Zuweisungen angegeben, um die neuen Spaltenwerte zu setzen. Der WHERE-Teil besteht aus einer Bedingung, die für eine Zeile erfüllt sein muss, damit diese entsprechend geändert wird. Lässt man die WHERE-Klausel weg, steht dies für `WHERE TRUE`, man ändert also alle Zeilen.

27

55

DELETE / TRUNCATE TABLE

```
DELETE FROM produkte WHERE produktnr = 88
```

```
DELETE FROM produkte WHERE hersteller = 'Monsterfood'
                       AND preis < 1.00
```

```
DELETE FROM produkte
```

```
TRUNCATE TABLE produkte
```

Auch hier bestimmt die WHERE-Klausel wieder, welche Zeilen gelöscht werden sollen. Lässt man sie weg, löscht das System jede Zeile. Der TRUNCATE TABLE-Befehl ist jedoch meist schneller für solche Zwecke, er leert ebenfalls die komplette Tabelle.

28

SELECT

```
SELECT H.land, COUNT(*) AS anzahl, AVG(P.preis) AS avg_preis
FROM hersteller H JOIN produkte P ON H.firma = p.hersteller
WHERE P.preis > 3
GROUP BY h.land
HAVING COUNT(*) < 5
ORDER BY COUNT(*)
```

"Zeige mir für jedes Land, in welchem weniger als fünf Produkte, die mehr als 3 EUR kosten, hergestellt wurden, die Anzahl der Produkte sowie die Durchschnittspreise an, aufsteigend sortiert nach der Anzahl.

- SELECT: Projektion
- FROM: Tabellen und Joins
- WHERE: Selektion
- GROUP BY: Zeilen gruppieren
- HAVING: Selektion nach der Gruppierung
- ORDER BY: Ergebnis sortieren

SELECT <spalten> FROM <tabelle> 57

```
SELECT bezeichnung, preis, round(preis * 1.15, 2) AS preis_usd
FROM produkte
```

bezeichnung	preis	preis_usd
Schokoriegel	0.89	1.02
Müsliriegel	1.19	1.37
Spülmaschinentabs	3.99	4.59
Katzenfutter	4.99	5.74
Maschinenbau-Lehrbuch	22.9	26.34
Regal	100	115
Geschenkverpackung	0	0

Die SELECT-Klausel wird bei der Anfrageausführung erst sehr spät ausgewertet. In ihr findet die Projektion statt, um festzulegen, welche Spalten man letztendlich im Ergebnis sehen möchte. Man kann mehrere Spalten mit Kommata trennen, mit AS einen Alias definieren - also eine Umbenennung vornehmen - und man kann Funktionen aufrufen. In der FROM-Klausel stehen eine oder mehrere Tabellen, aus der die Daten stammen sollen.

SELECT * FROM <tabelle>

```
SELECT * FROM produkte
```

produktnr	bezeichnung	preis	hersteller
17	Schokoriegel	0.89	Monsterfood
18	Müsliriegel	1.19	Monsterfood
29	Spülmaschinentabs	3.99	Calgonte
88	Katzenfutter	4.99	-
91	Maschinenbau-Lehrbuch	22.9	-
92	Regal	100	-
998	Geschenkverpackung	0	-
999	Katalog	0	-

Mit SELECT * wird das ganze Tupel ausgegeben. In der Anfrage auf dieser Folio werden alle Spalten der Produkte-Tabelle angezeigt.

SELECT DISTINCT

```
SELECT hersteller
FROM produkte
```

hersteller
Monsterfood
Monsterfood
Calgonte
-
-
-
-

```
SELECT DISTINCT hersteller
FROM produkte
```

hersteller
Monsterfood
Calgonte
-

In SQL wird standardmäßig keine Duplikateliminierung vorgenommen. Das Ergebnis einer Anfrage kann also - anders als in der relationalen Algebra - gleiche Zeilen doppelt beinhalten. Um eine Duplikateliminierung durchzuführen schreibt man `SELECT DISTINCT` statt `SELECT`.

WHERE

```
SELECT * FROM produkte WHERE preis > 3
```

produktnr	bezeichnung	preis	hersteller
29	Spülmaschinentabs	3.99	Calgonte
88	Katzenfutter	4.99	-
91	Maschinenbau-Lehrbuch	22.9	-
92	Regal	100	-

```
SELECT * FROM produkte WHERE preis >= 3 AND preis <= 9
```

```
SELECT * FROM produkte WHERE preis BETWEEN 3 AND 9
```

In der WHERE-Klausel steht eine beliebige boolesche Bedingung. Alle Zeilen, die diese Bedingung erfüllen, sind im Ergebnis. Es wird also eine Selektion ausgeführt. Hat eine Anfrage keine WHERE-Klausel, qualifiziert sich jede Zeile. Bedingungen lassen sich mit AND und OR verknüpfen und mit NOT negieren. Auch die Klammerung von Ausdrücken ist möglich.

LIKE-Prädikat

```
SELECT * FROM produkte WHERE bezeichnung = 'Müsliriegel'
```

produktnr	bezeichnung	preis	hersteller
18	Müsliriegel	1.19	Monsterfood

```
SELECT * FROM produkte WHERE bezeichnung LIKE 'M%'
```

produktnr	bezeichnung	preis	hersteller
18	Müsliriegel	1.19	Monsterfood
91	Maschinenbau-Lehrbuch	22.9	-

```
SELECT * FROM produkte WHERE bezeichnung LIKE '%-%'
```

```
SELECT * FROM produkte WHERE bezeichnung LIKE 'M sliriegel'
```

Strings müssen stets in 'einfache Anführungszeichen' geschrieben werden. 'Innerhalb eines Strings braucht''s zwei Anführungszeichen', um es zu escapen. Im LIKE-Prädikat wird eine Musterüberprüfung vorgenommen. % steht dabei für beliebige Zeichen und _ für genau ein beliebiges Zeichen.

NULL-Werte

```
SELECT * FROM produkte WHERE hersteller IS NULL
```

produktnr	bezeichnung	preis	hersteller
88	Katzenfutter	4.99	-
91	Maschinenbau-Lehrbuch	22.9	-
92	Regal	100	-
998	Geschenkverpackung	0	-
999	Katalog	0	-

```
SELECT * FROM produkte WHERE hersteller IS NOT NULL
```

Überprüfungen, ob der Wert einer Spalte NULL ist, muss mit `IS NULL` erfolgen. Das Prädikat `IS NOT NULL` trifft auf Zeilen zu, die in der entsprechenden Spalte keinen NULL-Wert haben.

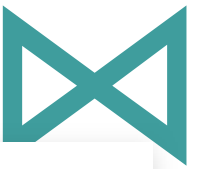
Kreuzprodukt

```
SELECT * FROM produkte, hersteller
```

produktnr	bezeichnung	preis	hersteller	firma	land
17	Schokoriegel	0.89	Monsterfood	Calgonte	Italien
17	Schokoriegel	0.89	Monsterfood	Monsterfood	USA
17	Schokoriegel	0.89	Monsterfood	Holzkopf	Österrei
18	Müsliriegel	1.19	Monsterfood	Calgonte	Italien
18	Müsliriegel	1.19	Monsterfood	Monsterfood	USA
18	Müsliriegel	1.19	Monsterfood	Holzkopf	Österrei
29	Spülmaschinentabs	3.99	Calgonte	Calgonte	Italien
29	Spülmaschinentabs	3.99	Calgonte	Monsterfood	USA

Listet man Komma-getrennt mehrere Tabellen in der FROM-Klausel auf, wird das kartesische Produkt (Cross-Join) zwischen diesen gebildet. Jede Zeile der einen Tabelle wird mit jeder Zeile der anderen verbunden.

Join



```
SELECT * FROM produkte, hersteller
WHERE produkte.hersteller = hersteller.firma
```

produktnr	bezeichnung	preis	hersteller	firma	land
17	Schokoriegel	0.89	Monsterfood	Monsterfood	USA
18	Müsliriegel	1.19	Monsterfood	Monsterfood	USA
29	Spülmaschinentabs	3.99	Calgonte	Calgonte	Italien

In der WHERE-Klausel lässt sich im Anschluss an das kartesische Produkt eine Join-Bedingung definieren, sodass sich ein Verbund wie hier auf der Folie gezeigt ausführen lässt.

<tabelle> JOIN <tabelle> ON <präd.>

61-62

```
SELECT * FROM produkte JOIN hersteller
 ON produkte.hersteller = hersteller.firma
```

produktnr	bezeichnung	preis	hersteller	firma	land
17	Schokoriegel	0.89	Monsterfood	Monsterfood	USA
18	Müsliriegel	1.19	Monsterfood	Monsterfood	USA
29	Spülmaschinentabs	3.99	Calgonte	Calgonte	Italien

Tabellen-Alias

```
SELECT * FROM produkte P JOIN hersteller H
 ON P.hersteller = H.firma
```

Die `JOIN ... ON`-Syntax in der FROM-Klausel ist eine Alternative zum Join in der WHERE-Klausel. An ihr ist besser erkennbar, dass hier ein Join ausgeführt wird, es verhindert, dass man aus Versehen die Join-Bedingung vergisst und es trennt klar den Verbund von weiteren Prädikaten. Außerdem lässt sich mit der `JOIN`-Klausel auch einen äußeren Verbund auszuführen (siehe nächste Folie). Bei nicht eindeutigen Spaltennamen muss unbedingt ein Tabellenprefix vor die Spalte geschrieben werden. Um Anfragen kompakter zu gestalten, bietet es sich an Tabellenaliase in der FROM-Klausel einzuführen (hier: P und H).

LEFT/RIGHT/FULL [OUTER] JOIN

```
SELECT * FROM produkte P LEFT JOIN hersteller H
 ON P.hersteller = H.firma
```

produktnr	bezeichnung	preis	hersteller	firma	land
17	Schokoriegel	0.89	Monsterfood	Monsterfood	USA
18	Müsliriegel	1.19	Monsterfood	Monsterfood	USA
29	Spülmaschinentabs	3.99	Calgonte	Calgonte	Italien
88	Katzenfutter	4.99	-	-	-
91	Maschinenbau-Lehrbuch	22.9	-	-	-
92	Regal	100	-	-	-

Ein äußerer Verbund lässt sich ausführen, indem statt dem `JOIN`-Schlüsselwort `LEFT JOIN`, `RIGHT JOIN` oder `FULL JOIN` geschrieben wird. Bei hier gezeigten linken äußeren Verbund sind alle Zeilen der linken Tabelle im Join-Ergebnis. Diejenigen, die keinen Join-Partner in der rechten Tabelle finden, erhalten in den entsprechenden Spalten einen NULL-Wert.

JOIN ... USING(...)

```
SELECT P.*, B.sterne
 FROM produkte P JOIN bewertungen B USING(produktnr)
```

produktnr	bezeichnung	preis	hersteller	sterne
17	Schokoriegel	0.89	Monsterfood	4
29	Spülmaschinentabs	3.99	Calgonte	1
29	Spülmaschinentabs	3.99	Calgonte	2

`USING(produktnr)` ist eine verkürzte Schreibweise für den Equi-Join auf der Spalte Produktnummer, also in dem Fall für `ON P.produktnr = B.produktnr`

Aggregatfunktionen

- COUNT: Anzahl
- SUM: Summe
- AVG: Durchschnittswert
- MIN: Kleinster Wert
- MAX: Größter Wert
- ...

Aggregatfunktion in der SELECT-Liste ohne GROUP BY-Klausel
⇒ die ganze Tabelle bildet eine große Gruppe
⇒ es kommt nur eine Zeile raus.

COUNT

```
SELECT COUNT(*)
FROM produkte
```

COUNT(*)
8

```
SELECT COUNT(hersteller)
FROM produkte
```

COUNT(hersteller)
3

```
SELECT COUNT(DISTINCT hersteller) FROM produkte
```

COUNT(DISTINCT hersteller)
2

`COUNT(*)` (manche schreiben auch gerne `COUNT(1)`) zählt die Anzahl der Zeilen in der Gruppe. Da hier die Gruppe die ganze Tabelle ist, liefert dies die Anzahl der Zeilen in der Tabelle. COUNT mit Spaltenangaben zählt, wie viele Zeilen in den gegebenen Spalten keinen NULL-Wert haben.
`COUNT(DISTINCT ...)` zählt die von NULL verschiedenen distinkten (unterschiedlichen) Werte in den gegebenen Spalten. In unserem Beispiel existieren die beiden Werte Calgonte und Monsterfood in der Hersteller-Spalte.

SUM / AVG / MIN / MAX

```
SELECT SUM(preis), AVG(preis), MIN(preis), MAX(preis)
FROM produkte
```

SUM(preis)	AVG(preis)	MIN(preis)	MAX(preis)
133.96	16.745	0	100

Den Aggregatfunktionen SUM, AVG, MIN und MAX wird als Argument eine Spalte (oder ein Ausdruck, z. B. `SUM(preis * (1+steuersatz/100))` übergeben. Wir sehen hier im Ergebnis die Summe aller Preise, den Durchschnitt sowie den kleinsten und größten Preis. NULL-Werte werden in den Aggregatfunktionen übersprungen. Das heißt `AVG(preis)` berechnet sich aus der Summe der Preise, die nicht NULL sind, geteilt durch die Anzahl der Preise, die nicht NULL sind. Also: `SUM(preis)/COUNT(preis)`

65

GROUP BY

```
SELECT hersteller, COUNT(*), AVG(preis)
FROM produkte
GROUP BY hersteller
```

hersteller	COUNT(*)	AVG(preis)
-	5	25.578
Calgonte	1	3.99
Monsterfood	2	1.04

In der GROUP BY-Klausel werden Spalten spezifiziert, nach deren Werten Gruppen gebildet werden. `GROUP BY hersteller` bedeutet, dass für jeden distinkten Wert in der hersteller-Spalte eine Gruppe gebildet wird. In unserem Beispiel sind dies drei. Für jede Gruppe kann nun der Wert in dieser GROUP BY-Spalte ausgegeben werden. Alle anderen Spalten dürfen nur innerhalb von Aggregatfunktionen in der SELECT-Liste vorkommen. Wir berechnen in dem Beispiel auf dieser Folie für jeden einzelnen Hersteller die Anzahl und Durchschnittspreise seiner Produkte.

GROUP BY

```
SELECT hersteller, preis, COUNT(*)
FROM produkte
GROUP BY hersteller, preis
```

hersteller	preis	COUNT(*)
-	0	2
-	4.99	1
-	22.9	1
-	100	1
Calgonte	3.99	1
Monsterfood	0.99	1

Hier gruppieren wir nach Hersteller und Preis. Das heißt für jede Kombination aus einem Hersteller und einem Preis wird eine Gruppe gebildet. Das Ergebnis lesen wir jetzt so: Vom Hersteller Calgonte gibt es ein Produkt, welches 3.99 kostet. Auch hier ist wieder darauf zu achten, dass im SELECT-Teil nur Spalten stehen dürfen, die im GROUP BY vorkommen. Alle anderen dürfen nur innerhalb von Aggregatfunktionen auftreten.

HAVING

```
SELECT hersteller, AVG(preis)
FROM produkte
GROUP BY hersteller
HAVING COUNT(*) >= 2
```

hersteller	AVG(preis)
-	25.578
Monsterfood	1.04

Die Ausführungsreihenfolge von SELECT-Abfragen ist wie folgt: Zunächst werden die Daten aus den Tabellen geholt, die in der FROM-Klausel stehen, dann wird die WHERE-Klausel ausgewertet, um die Zeilen zu filtern. Im Anschluss daran erfolgt die Gruppierung und die Ausgabe der Spalten und berechneten Aggregatfunktionen in der SELECT-Liste. Möchte man nach der Gruppierung nochmals eine Selektion ausführen, verwendet man die HAVING-Klausel. In ihr stehen in der Regel nur Aggregatfunktionen, da man andere Prädikate ja bereits im WHERE-Teil ausführen kann. Im gezeigten Beispiel berechnen wir für jeden Hersteller, von dem es mindestens zwei Produkte gibt, die Durchschnittspreise dieser Produkte.

Sub-Anfragen

```
SELECT hersteller, avg_preis FROM (
 SELECT hersteller, COUNT(*) AS anzahl, AVG(preis) AS avg_preis
 FROM produkte GROUP BY hersteller
) P WHERE anzahl >= 2
```

```
SELECT * FROM produkte
WHERE preis = (SELECT MAX(preis) FROM produkte)
```

```
SELECT * FROM produkte P1
WHERE preis = (SELECT MAX(preis) FROM produkte P2
               WHERE P2.hersteller = P1.hersteller)
```

Unteranfragen dürfen fast überall innerhalb einer Anfrage stehen, auch in INSERT, UPDATE, DELETE, usw. Auch korrelierende Sub-Anfragen sind erlaubt. In einer solchen kommen Spalten der Oberanfrage vor (P1.hersteller in der unteren Anfrage). Die obere Query entspricht der Anfrage mit HAVING aus der vorherigen Folie. Die zweite Anfrage liefert Produkte, die den Maximalpreis haben, es gibt also keine teureren als diese. Die untere Anfrage ist ähnlich: Sie liefert Produkte, die den Maximalpreis von Produkten vom jeweiligen Hersteller haben. Die Anfrage liefert z. B. den Müsliriegel, da es von seinem Hersteller Monsterfood keine Produkte gibt, die mehr kosten.

CTE: Common Table Expressions

WITH ... AS (SELECT ...) SELECT ...

```
WITH monsterfood_produkte AS
 (SELECT * FROM produkte WHERE hersteller = 'Monsterfood')
SELECT count(*) FROM monsterfood_produkte
```

count(*)
2

```
WITH monsterfood_produkte AS
    (SELECT * FROM produkte WHERE hersteller = 'Monsterfood')
SELECT * FROM monsterfood_produkte
WHERE preis = (SELECT max(preis) FROM monsterfood_produkte)
```

produktnr	bezeichnung	preis	hersteller
18	Müsliriegel	1.19	Monsterfood

Mit CTEs lassen sich Subanfragen Alias-Namen geben. Das ist vor allem dann praktisch, wenn man eine Unteranfrage mehrfach benötigt. Im unteren Beispiel suchen wir Monsterfood-Produkte, die den Höchstpreis aller Monsterfood-Produkte haben.

ORDER BY

"Tabellen haben keine Ordnung, Ergebnisse schon."

- ASC: aufsteigend sortieren (Standard)
- DESC: absteigend sortieren

```
SELECT * FROM produkte ORDER BY preis
```

produktnr	bezeichnung	preis	hersteller
998	Geschenkverpackung	0	-
999	Katalog	0	-
17	Schokoriegel	0.89	Monsterfood
18	Müsliriegel	1.19	Monsterfood
29	Spülmaschinentabs	3.99	Calgonte

Hier wird das Ergebnis der Anfrage aufsteigend nach den Werten der Preis-Spalte sortiert.

ORDER BY

"Tabellen haben keine Ordnung, Ergebnisse schon."

```
SELECT * FROM produkte
ORDER BY hersteller NULLS LAST, preis DESC
```

produktnr	bezeichnung	preis	hersteller
29	Spülmaschinentabs	3.99	Calgonte
18	Müsliriegel	1.19	Monsterfood
17	Schokoriegel	0.89	Monsterfood
92	Regal	100	-
91	Maschinenbau-Lehrbuch	22.9	-

Gibt man mehrere Ausdrücke im ORDER BY an, wird zuerst nach dem ersten sortiert, dann nach dem zweiten, usw. NULLS FIRST bzw. NULLS LAST sorgt dafür, dass NULL-Werte in der zu sortierenden Spalte an den Anfang bzw. ans Ende sortiert werden. Hier kommt Calgonte im Alphabet vor Monsterfood, aber da es zwei Produkte von Monsterfood gibt, kommt wegen `preis DESC` das teuerste zuerst.

LIMIT

SQL Standard (und DB2):

```
SELECT * FROM produkte ORDER BY preis DESC
FETCH FIRST 5 ROWS ONLY
```

PostgreSQL, MySQL, MariaDB, SQLite, ...

```
SELECT * FROM produkte ORDER BY preis DESC
LIMIT 5
```

Mit der LIMIT-Klausel lässt sich die Ergebnismenge auf eine bestimmte Anzahl von Tupeln begrenzen. `SELECT * FROM produkte LIMIT 5` würde irgendwelche fünf Produkte liefern. Die untere Anfrage auf dieser Folie liefert die fünf teuersten Produkte. `SELECT * FROM produkte ORDER BY preis DESC LIMIT 1` liefert das teuerste Produkt. Haben zwei Produkte den gleichen und den teuersten Preis, wird irgendeines von diesen ausgegeben.

51

LIMIT mit OFFSET

```
SELECT * FROM produkte ORDER BY preis DESC
LIMIT 2, 5
```

```
SELECT * FROM produkte ORDER BY preis DESC
LIMIT 5 OFFSET 2
```

produktnr	bezeichnung	preis	hersteller
88	Katzenfutter	4.99	-
29	Spülmaschinentabs	3.99	Calgonte
18	Müsliriegel	1.19	Monsterfood
17	Schokoriegel	0.89	Monsterfood
998	Geschenkverpackung	0	-

Die beiden hier dargestellten Anfragen sind äquivalent. Der Offset sorgt dafür, dass zunächst eine bestimmte Anzahl von Zeilen übersprungen wird. Hier werden die beiden teuersten Produkte (Regal und Maschinenbau-Lehrbuch) übersprungen und erst die fünf danach teuersten Produkte ausgegeben.

52

SELECT

```
SELECT H.land, COUNT(*) AS anzahl, AVG(P.preis) AS avg_preis
FROM hersteller H JOIN produkte P ON H.firma = p.hersteller
WHERE P.preis > 3
GROUP BY h.land
HAVING COUNT(*) < 5
ORDER BY COUNT(*)
```

"Zeige mir für jedes Land, in welchem weniger als fünf Produkte, die mehr als 3 EUR kosten, hergestellt wurden, die Anzahl der Produkte sowie die Durchschnittspreise an, aufsteigend sortiert nach der Anzahl.

land	anzahl	avg_preis
Italien	1	3.99

Ich will im Ergebnis auch sehen, dass es z. B. aus Österreich 0 solche Produkte gibt!

SELECT mit OUTER JOIN

73

```
SELECT H.land, COUNT(*) AS anzahl, AVG(P.preis) AS avg_preis
FROM hersteller H LEFT JOIN produkte P ON p.hersteller = H.firma
WHERE P.preis > 3
GROUP BY h.land HAVING COUNT(*) < 5 ORDER BY COUNT(*)
```

Problem: WHERE-Prädikat `preis > 3` wird nach dem äußeren Verbund ausgeführt, schmeißt also die gewünschten Zeilen mit NULL-Werten wieder raus.

```
SELECT /****/
WHERE P.preis > 3 OR P.produktnr IS NULL
```

```
SELECT H.land, COUNT(*) AS anzahl, AVG(P.preis) AS avg_preis
FROM hersteller H
 LEFT JOIN (SELECT * FROM produkte WHERE preis > 3) P
 ON p.hersteller = H.firma
GROUP BY h.land HAVING COUNT(*) < 5 ORDER BY COUNT(*)
```

Ein LEFT JOIN alleine löst das Problem nicht, dass Länder ohne teure Produkte im Ergebnis nicht auftauchen. Hier sind zwei Lösungen. Die erste liefert die Zeilen, die beim Left Join keinen Joinpartner gefunden haben, die zweite erzwingt ein frühzeitiges Filtern vor der Ausführung des Joins.

UNION

Aus welchen Ländern sind unsere Kunden und Hersteller?

```
SELECT land FROM kunden
UNION
SELECT land FROM hersteller
```

land
Deutschland
Italien
USA
Österreich

Bei Mengenoperationen ohne das Stichwort ALL wird nach UNION, INTERSECT und EXCEPT eine Duplikateliminierung vorgenommen.

Die Ergebnismengen zweier SELECT-Anfragen lassen sich mit Mengenoperatoren (UNION, UNION ALL, INTERSECT, INTERSECT ALL, EXCEPT, EXCEPT ALL) kombinieren.

UNION ALL

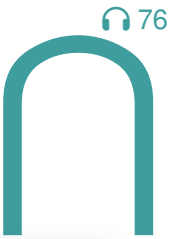
Aus welchen Ländern sind unsere Kunden und Hersteller?

```
SELECT land FROM kunden
UNION ALL
SELECT land FROM hersteller
```

land
Deutschland
Deutschland
Italien
Italien
USA

Auf der vorherigen Folie haben wir gesehen, dass jedes Land im Ergebnis nur einmal vorkommt. Das liegt daran, dass im Anschluss an Mengenoperationen ohne das Stichtwort ALL eine Duplikateliminierung vorgenommen wird. Hier sehen wir wegen `UNION ALL` auch doppelte Zeilen.

INTERSECT



In welchen Ländern gibt es Kunden und Hersteller?

```
SELECT land FROM kunden
INTERSECT
SELECT land FROM hersteller
```

land
Italien

```
SELECT land FROM kunden
INTERSECT ALL
SELECT land FROM hersteller
```

INTERSECT bildet die Schnittmenge zwischen zwei Mengen. Wichtig bei allen Mengenoperationen ist, dass die Mengen vereinigungsverträglich sind. Die Anzahl der Spalten muss also gleich und die Datentypen der Spalten müssen zueinander kompatibel sein. INTERSECT ALL eliminiert keine Duplikate. Würde es also zwei Kunden aus Italien und drei Hersteller aus Italien geben, stände Italien zweimal im Ergebnis.

EXCEPT (MINUS)

In welchen Ländern gibt es Kunden, aber keine Hersteller?

```
SELECT land FROM kunden
EXCEPT
SELECT land FROM hersteller
```

land
Deutschland

```
SELECT land FROM kunden
EXCEPT ALL
SELECT land FROM hersteller
```

land
Deutschland
Deutschland

Gibt es drei Kunden und ein Hersteller aus Deutschland, bleibt bei EXCEPT ALL 2x Deutschland übrig.

Das EXISTS-Prädikat

- Eingabe: Eine SELECT-Anfrage
- Ausgabe:
  - TRUE, wenn die Anfrage mind. 1 Zeile liefert
  - FALSE, wenn die Anfrage die leere Menge liefert

```
SELECT * FROM hersteller H
WHERE EXISTS
 (SELECT * FROM produkte P WHERE p.hersteller = H.firma)
```

firma	land
Calgonte	Italien
Monsterfood	USA

Die Anfrage hier liefert Hersteller, von denen es Produkte gibt. Dies könnte man auch einfach mit einem (Semi-)Join lösen.

59

NOT EXISTS

Von welchen Herstellern gibt es keine Produkte?

```
SELECT * FROM hersteller H
WHERE NOT EXISTS
 (SELECT * FROM produkte P WHERE p.hersteller = H.firma)
```

firma	land
Holzkopf	Österreich

Für Hersteller, von denen es Produkte gibt, liefert die innere SELECT-Anfrage ein nicht-leeres Ergebnis (nämlich die Produkte des jeweiligen Herstellers). Für die anderen Hersteller (hier: Holzkopf) liefert die Unteranfrage ein leeres Ergebnis, EXISTS ist also FALSE, NOT EXISTS ist damit TRUE und der Hersteller wird ausgegeben.

60

Das IN-Prädikat

- Eingabe: Ein Ausdruck und entweder eine Werte-Liste oder eine SELECT-Anfrage
- Ausgabe:
  - TRUE, wenn der Ausdruck enthalten ist
  - FALSE, wenn der Ausdruck nicht enthalten ist

```
SELECT * FROM hersteller H
WHERE firma IN ('Holzkopf', 'Monsterfood')
```

firma	land
Holzkopf	Österreich
Monsterfood	USA

Hier erspart man durch die IN-Liste Schreibarbeit, da man ansonsten `WHERE hersteller = 'Holzkopf' OR hersteller = 'Monsterfood'` schreiben müsste.

61

Das IN-Prädikat

```
SELECT * FROM hersteller H
WHERE firma IN (SELECT hersteller FROM produkte)
```

firma	land
Calgonte	Italien
Monsterfood	USA

```
SELECT * FROM hersteller H
WHERE firma NOT IN
 (SELECT hersteller FROM produkte WHERE hersteller IS NOT NULL)
```

firma	land
Holzkopf	Österreich

Oben: Hersteller, von denen es Produkte gibt; unten: Hersteller, von denen es keine Produkte gibt. Würde man unten das `WHERE hersteller IS NOT NULL` weglassen, würden in der Sub-Anfrage NULL-Werte auftauchen und das Ergebnis der Anfrage wäre leer. Berechnungen und Vergleiche mit NULL-Werten liefern stets NULL, was bei Prädikatsauswertungen immer mit FALSE gedeutet wird.

62

80

CAST: Typumwandlung

CAST(ausdruck AS datentyp)

```
SELECT preis, CAST(preis AS INT), CAST(preis AS VARCHAR(50))
FROM produkte
```

preis	CAST(preis AS INT)	CAST(preis AS VARCHAR(50))
0.89	0	0.89
1.19	1	1.19
3.99	3	3.99
4.99	4	4.99
22.9	22	22.9
100	100	100

Unsere Preis-Spalte hat den Datentyp DECIMAL(9,2). Wenn wir die Werte dieser Spalte in INTEGER umwandeln, werden die Nachkommastellen verworfen. Casten wir auf VARCHAR, sehen wir im Ergebnis keinen Unterschied, allerdings liegen unsere Preise nun als Strings und nicht mehr als Zahl vor.

81

CASE WHEN

CASE WHEN ... THEN ... ELSE ... END

```
SELECT bezeichnung, preis,
CASE WHEN preis < 1 THEN 'billig'
     WHEN preis < 5 THEN 'mittel'
     ELSE 'teuer' END
FROM produkte
```

bezeichnung	preis	CASE WHEN preis < 1 THEN 'billig' WHEN preis < 5 THEN 'mittel' ELSE 'teuer' END
Schokoriegel	0.89	billig
Müsliriegel	1.19	mittel
Spülmaschinentabs	3.99	mittel
Katzenfutter	4.99	mittel
Maschinenbau	22.9	teuer

Wenn die erste Bedingung nicht erfüllt ist, wird die zweite überprüft, usw.

CASE WHEN

CASE ... WHEN ... THEN ... ELSE ... END

```
SELECT bezeichnung, preis,
CASE preis WHEN 0 THEN 'kostenlos' ELSE preis END
FROM produkte ORDER BY preis
```

bezeichnung	preis	CASE preis WHEN 0 THEN 'kostenlos' ELSE preis END
Geschenkverpackung	0	kostenlos
Katalog	0	kostenlos
Schokoriegel	0.89	0.89
Müsliriegel	1.19	1.19

Steht zwischen CASE und WHEN ein Ausdruck, kann man diesen Ausdruck ohne viel Schreibarbeit auf verschiedene Werte überprüfen. Der hier stehende Ausdruck ist äquivalent zu `CASE WHEN preis = 0 THEN 'kostenlos' ELSE preis END`

Dummy-Tabelle DUAL

81

In einigen DBMS (Oracle, DB2, ...) gibt es die Tabelle DUAL.

```
SELECT * FROM dual
```

dummy
-

Praktisch zum Rechnen und Ausprobieren von Funktionen:

```
SELECT 5*3, CASE WHEN 7>4 THEN 'Ja' ELSE 'Nein' END FROM DUAL
```

5*3	CASE WHEN 7>4 THEN 'Ja' ELSE 'Nein' END
15	Ja

SELECT ohne FROM in PostgreSQL

```
SELECT 5*3, CASE WHEN 7>4 THEN 'Ja' ELSE 'Nein' END
```

5*3	CASE WHEN 7>4 THEN 'Ja' ELSE 'Nein' END
15	Ja

Wie viele Tage sind noch bis Weihnachten?

```
SELECT DATE '2023-12-25' - CURRENT_DATE
```

PostgreSQL erlaubt SELECT-Anfragen ohne FROM-Klausel. Dies entspricht Anfragen auf der DUAL-Tabelle von der vorherigen Folie. Man kann mit solchen Anfragen leicht Funktionen aufrufen und ausprobieren oder Berechnungen durchführen.

Kapitelzusammenfassung

- SQL: Anfragesprache für strukturierte Datenbanken
- DDL: CREATE / ALTER / DROP TABLE
- Datentypen: INT, VARCHAR, CHAR, DECIMAL, DATE, ...
- Constraints: PRIMARY KEY, NOT NULL, CHECK, UNIQUE, ...
- Referentielle Aktionen: ON DELETE/UPDATE ...
- DML: INSERT, UPDATE, DELETE, TRUNCATE, SELECT
- SELECT ... FROM ... WHERE ... GROUP BY ... HAVING ... ORDER BY ... LIMIT
- [LEFT/RIGHT/FULL] JOIN ON ...
- Aggregatfunktionen: COUNT, SUM, AVG, MIN, MAX
- Subanfragen, CTEs, EXISTS, IN
- Mengenoperationen: UNION / INTERSECT / EXCEPT [ALL]