

| Im ERD | Best Practices Überführung |
|--|--|
| Entitätstyp mit Attributen und Schlüsselattributen | <ul style="list-style-type: none"> Jeder Entitätstyp wird zu einer eigenen Relation mit entsprechenden Attributen übernommen Der Schlüsselattribut(e) wird als Primärschlüssel übernommen und üblicherweise an den Anfang des Relationenschemas gestellt |
| Schwache Entitätstypen | <ul style="list-style-type: none"> Attribute der schwachen Entität werden, um den Schlüssel der starken Entität erweitert Primärschlüssel: Schlüssel der starken Entität und partieller Schlüssel der schwachen Entität |
| Beziehungstyp 1:1 | <ul style="list-style-type: none"> Sind sehr selten und werden nach intensivem Review in der Regel in einer Relation dargestellt oder in zwei mit Fremdschlüsselbeziehung wenn DBMS das kann (Zirkelbezug) |
| Beziehungstypen n:m | <ul style="list-style-type: none"> Für n:m-Beziehungen muss eine Beziehungsrelation angelegt werden |
| Beziehungstyp 1:n | <ul style="list-style-type: none"> Bei zwei Relationen: In einer 1:N Beziehung kommt der Fremdschlüssel immer auf die Seite, wo das N steht Drei möglich werden aber gemieden |



Datenbanksysteme

SQL

Prof. Dr. Patrick Cato

Technische Hochschule Ingolstadt 

Überblick: DB-Entwurf und Modellierung

Anforderungen



Konzeptueller Entwurf

ERD



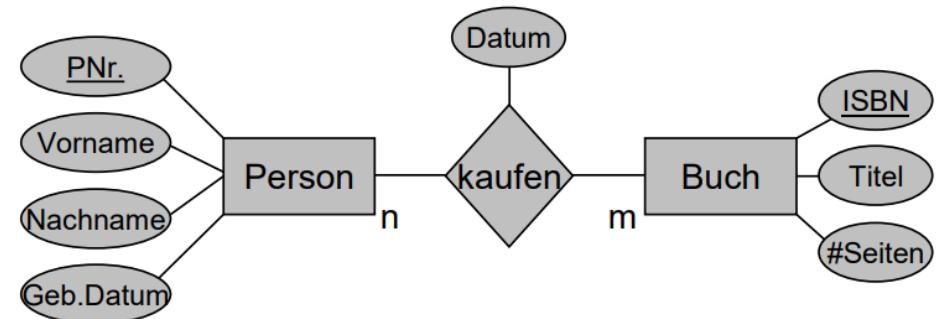
Logischer Entwurf

Relationenmodell



Implementierung

DBS



Person(PNR, Vorname, Nachname, GebDatum)
Buch(ISBN, Titel, Seiten)
Kauf(! PNR, ! ISBN | Datum)

SQL DDL

```

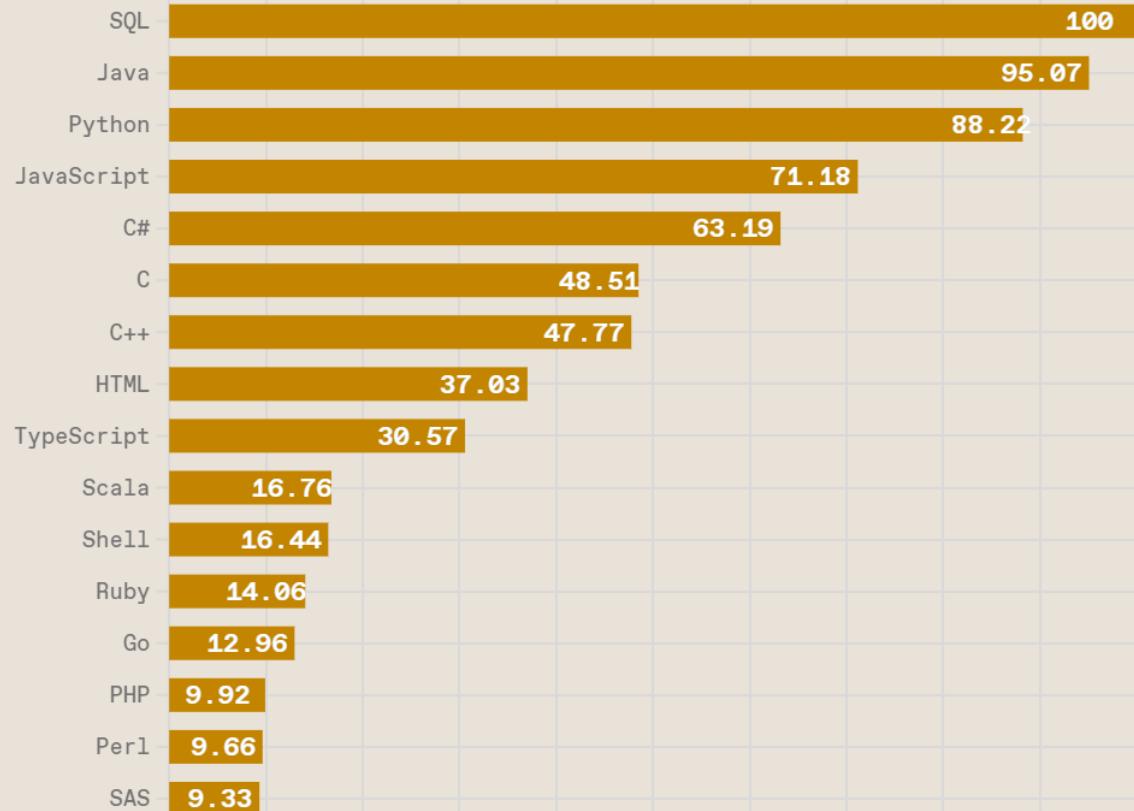
CREATE TABLE Person (
    PNR      int          PRIMARY KEY,
    Vorname  varchar(50),
    Nachname varchar(50) NOT NULL,
    GebDatum date         NOT NULL
);
  
```



Top Programming Languages 2022

Click a button to see a differently weighted ranking

Spectrum **Jobs** Trending



SQL ist eine deklarative Abfragesprache

types of programming:

procedural (imperative)

object-oriented

SQL

declarative (nonprocedural)

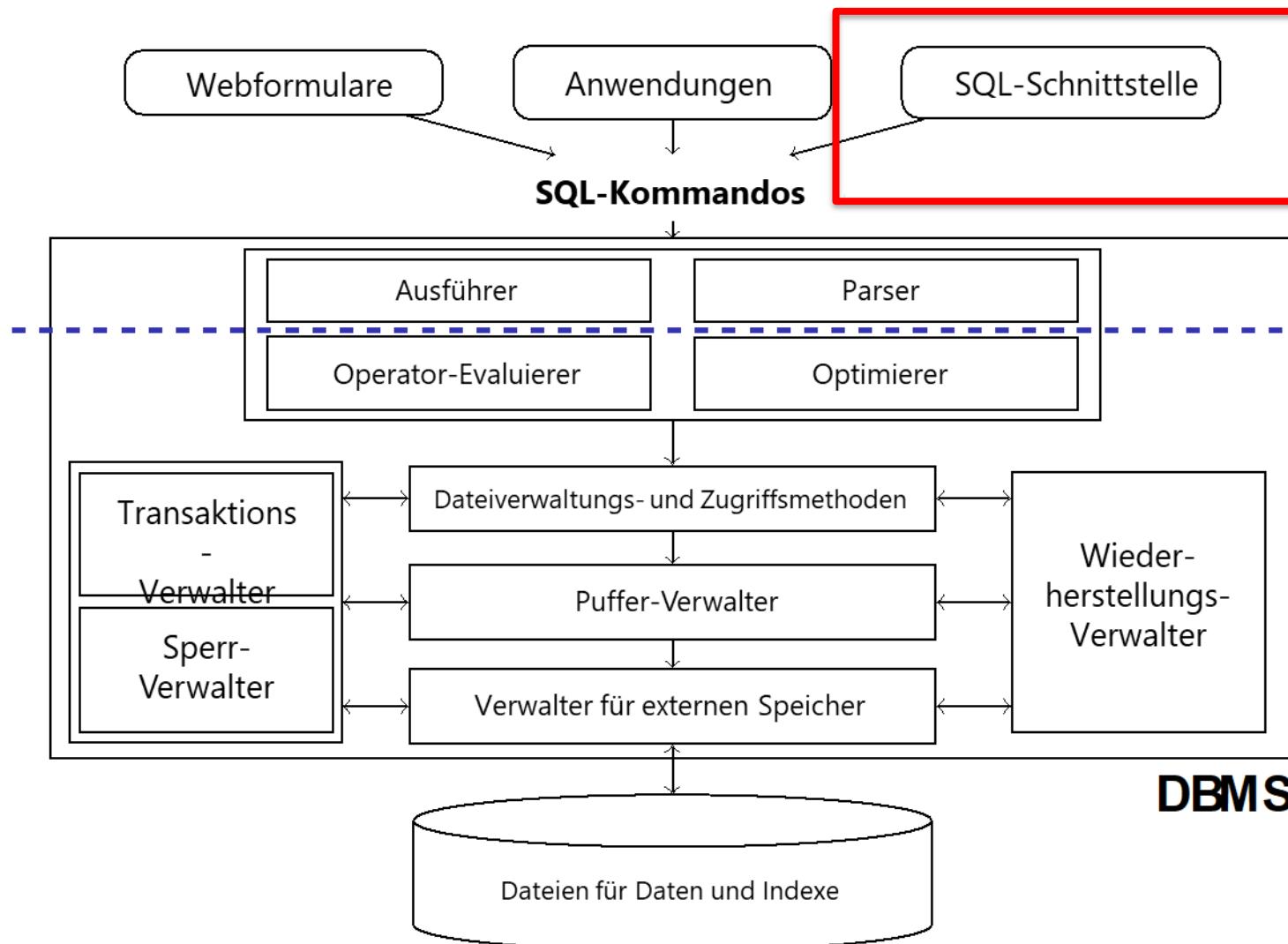
functional



Architektur eines relationalen DBMS - Modulperspektive

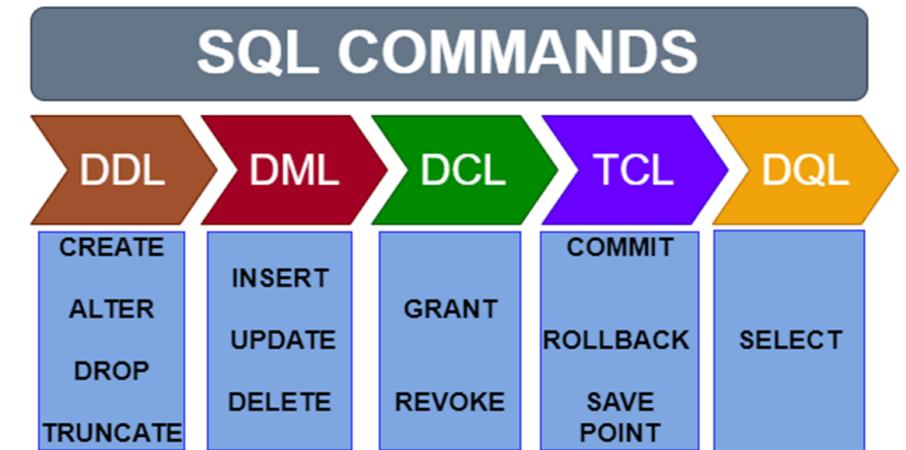


Verortung
(Aus 01_Einführung)

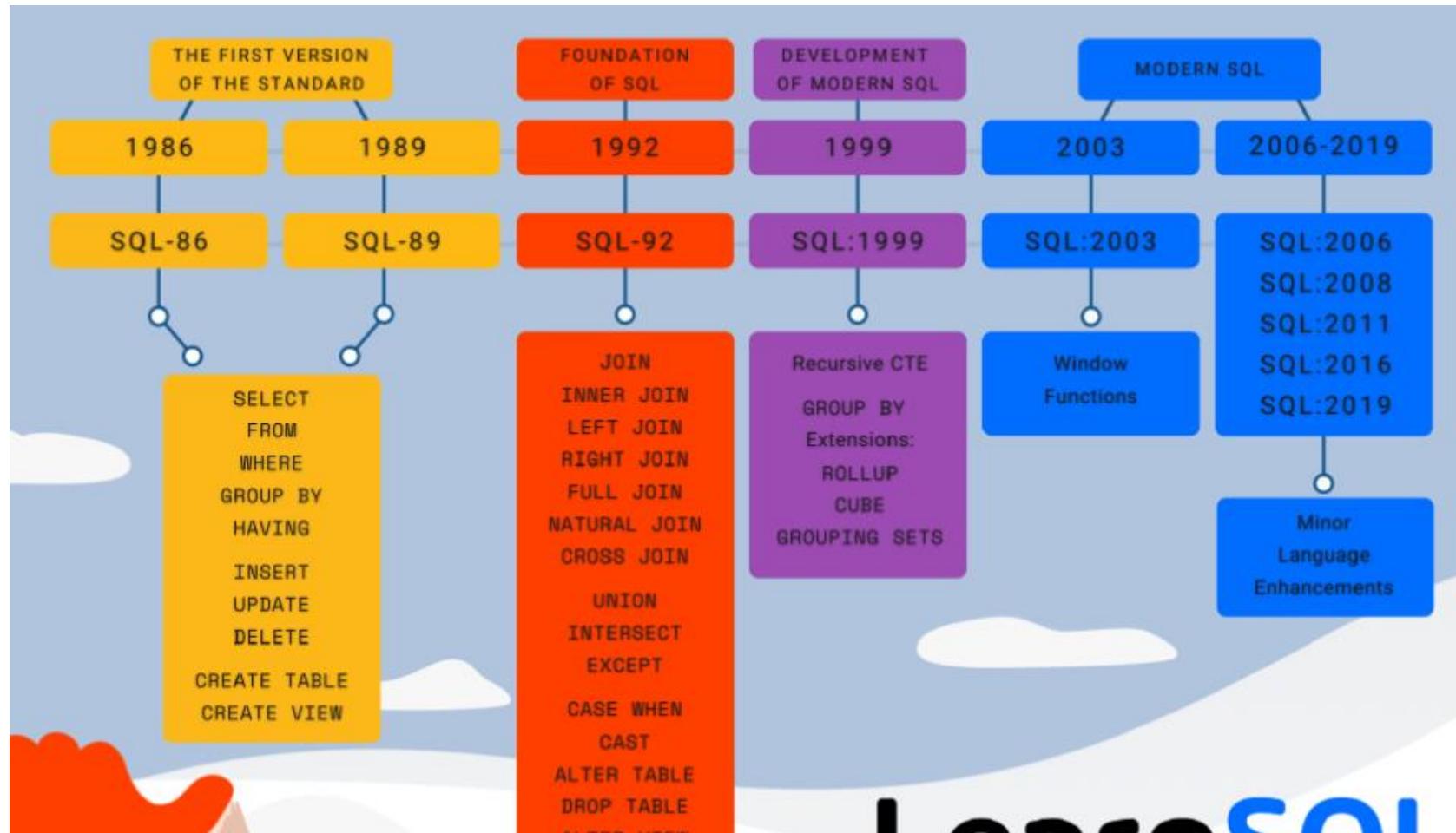


Kernaspekte

- Structured English Query Language (SEQUEL)
- erste SQL unterstützende kommerzielle Systeme waren Oracle (1979) und IBM SQL/DS (1981)
- SQL standardisiert durch American National Standards Institute (ANSI) und International Organization for Standardization (ISO)
- SQL kann in 5 Sprachen unterteilt werden



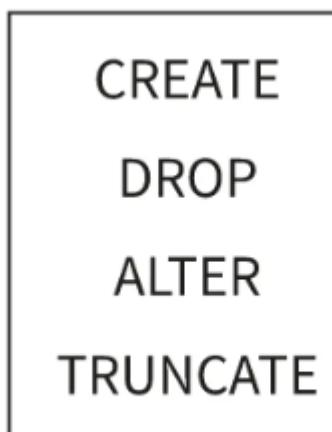
- **Data Definition Language (DDL)** zur Schemadefinition (z.B. Anlegen von Tabellen)
- **Data Manipulation Language (DML)** zur Datenmanipulation (z.B. Einfügen von Daten in Tabellen)
- **Data Query Language (DQL)** zum Anfragen (z.B. bestimmter Zeilen)
- **Data Control Language (DCL)** zur Rechteverwaltung (z.B. Zugriffsrechte auf Tabellen)
- **Transaction Control Language (TCL)** zur Transaktionsverwaltung (z.B. rückgängige machen von Transaktionen)



Data Definition Language (DDL)

Definition DDL

Befehlssatz zum Anlegen, Ändern und Löschen von Datenbanken, Schemas, Tabellen und ihren Strukturen.



CREATE
DROP
ALTER
TRUNCATE

CREATE DATABASE

Befehlssatz zum Anlegen einer Datenbank auf dem Datenbankserver.

Beispiel SQL-Syntax für CREATE DATABASE

```
CREATE DATABASE <datenbankname>;
```

Hinweis: Je nach Datenbanksystem ist datenbankname klein zu schreiben

Beispiel CREATE DATABASE

```
CREATE DATABASE escooterverwaltung ;
```

CREATE TABLE

Befehlssatz zum Anlegen einer Tabelle (Relation) auf dem Datenbankserver.

Beispiel SQL-Syntax für CREATE Table

```
CREATE TABLE <Name der Tabelle> (
    <Name von Attribut 1> datentyp PRIMARY KEY,
    <Name von Attribut 2> datentyp (NOT) NULL,
    ...
)
```

Hinweis: Je nach Datenbanksystem ist Name der Tabelle klein zu schreiben

Beispiel CREATE TABLE

eScooter(ScooterID:integer, SerialNumber:integer, Brand:varchar, BatteryStatus:int, isRentable:smallint, EmployeeID:int)

```
CREATE TABLE escooter (
    scooterid integer NOT NULL PRIMARY KEY,
    serialnumber integer,
    brand varchar(100),
    batterystatus integer,
    isrentable smallint,
    employeeid integer
);
```

CREATE SCHEMA

Manche Datenbanksysteme ermöglichen eine zusätzliche Strukturierungsebene einzufügen. Bei Postgres werden standardmäßig Tabellen in das Schema „public“ abgelegt.

Beispiel SQL-Syntax

```
CREATE SCHEMA <name>;  
  
CREATE TABLE schemaname.escooter (  
scooterid integer NOT NULL PRIMARY KEY,  
serialnumber integer,  
brand varchar(100),  
batterystatus integer,  
isrentable smallint,  
employeeid integer
```

Definition Constraints

Constraints: Integritätsbedingungen, die bei der Tabellendefinition (oder Änderung) festgelegt werden.
Z. B. Statistische Wertebereiche, definierte Datentypen und Feldlängen, Fremdschlüssele

Beispiel SQL-Syntax

```
CREATE TABLE escooter2 (
    scooterid integer NOT NULL PRIMARY KEY
        CHECK(scooterid>0),
    serialnumber varchar(100) NOT NULL,
    brand varchar(100),
    batterystatus integer,
    isrentable boolean,
    employeeid integer
);
```

Definition CHECK Constraint

Statische Bedingungen werden in SQL von einer CHECK-Anweisung gefolgt von einer Bedingung implementiert Änderungen an einer Tabelle werden zurückgewiesen, wenn die Bedingung zu false ausgewertet wird.

Beispiel SQL-Syntax

```
CHECK batterystatus BETWEEN 0 AND 100
CHECK brand IN ('Minimotors', 'Xiaomi', 'Inokim', 'Zero', 'Zoom')
```

Definition Constraint NOT NULL

- RDBMs unterstützen Null-Werte für alle Datentypen
- Null-Wert zeigt an, dass der Wert des Attributs nicht bekannt ist oder das Attribut nicht anwendbar ist
- Null-Werte können bei der Schemadefinition (d.h. dem Anlegen von Tabellen) erlaubt oder untersagt werden
- ***NOT NULL lässt keine NULL-Werte zu***
- ***NONE (oder keine Angabe) lässt NULL-Werte zu***
- Bei Primary Key ist Angabe optional, weil diese per Definition nie NULL-Werte haben

Definition Fremdschlüssel

- Fremdschlüssel verweisen auf den Primärschlüssel einer anderen Tabelle oder auf eine Spalte, die UNIQUE definiert ist. Hierdurch kann sichergestellt werden, dass Eintrag vorhanden ist
- Eine Tabelle kann höchstens einen Primärschlüssel, aber mehrere Fremdschlüssel besitzen
- Fremdschlüssel werden mittel FOREIGN KEY ... REFERENCES angegeben

```
CREATE TABLE escooter (
    scooterid integer PRIMARY KEY,
    serialnumber varchar(100) UNIQUE NOT NULL,
    brand varchar(100),
    batterystatus integer,
    isrentable boolean,
    employeeid integer,
    FOREIGN KEY (employeeid) REFERENCES
        employees(employeeid));
```

Beispiel

```
CREATE TABLE verwaltung.employee (
    eid INTEGER PRIMARY KEY,
    vorname varchar(100),
    nachname varchar(100));

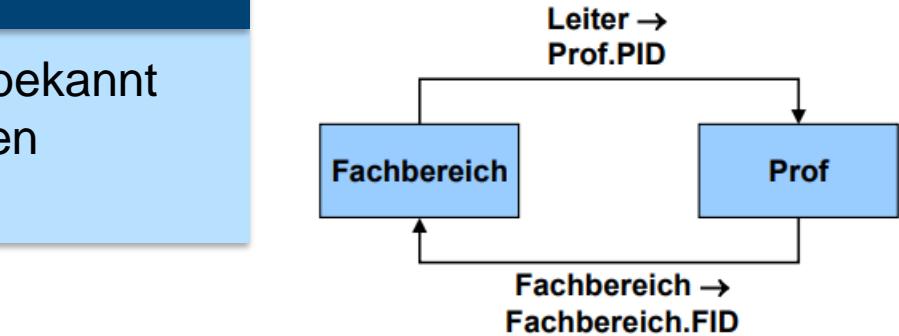
CREATE TABLE verwaltung.escooter (
    scooterid integer PRIMARY KEY,
    serialnumber varchar(100) UNIQUE NOT NULL,
    brand varchar(100),
    batterystatus integer,
    isrentable boolean,
    employeeid integer,
    FOREIGN KEY (employeeid) REFERENCES
    verwaltung.employee(eid));
```

Zyklische Referenzen

- Beim Anlegen der ersten Tabelle ist andere Tabelle noch nicht bekannt
- Fremdschlüsselbeziehung muss nachträglich hinzugefügt werden

```
CREATE TABLE fachbereich (
    fid integer PRIMARY KEY,
    leiter integer);
```

```
CREATE TABLE professor (
    pid integer PRIMARY KEY,
    fachbereich integer,
    CONSTRAINT fk_fb FOREIGN KEY (fachbereich)
    REFERENCES fachbereich(fid));
```



```
ALTER TABLE fachbereich ADD
CONSTRAINT fk_lt FOREIGN KEY
(leiter) REFERENCES
professor(pid);
```

UNIQUE-Constraint

- Alternativschlüssel (im Beispiel eScooter das Attribut SerialNumber können als UNIQUE deklariert werden)
- UNIQUE impliziert nicht NOT NULL da er kein Primary Key ist!

```
CREATE TABLE escooter (
    scooterid integer PRIMARY KEY,
    serialnumber varchar(100) UNIQUE NOT NULL,
    brand varchar(100),
    batterystatus integer,
    isrentable boolean,
    employeeid integer
);
```

Referentielle Aktionen

Bei der Definition des Fremdschlüssels können wir angeben, was bei Änderung mit den referenzierten Daten geschehen soll (Referential Actions). Ohne weitere Angabe verweigert das RDBMS das Löschen eines Tupels, sofern noch ein anderes Tupel per Fremdschlüssel darauf verweist.

Angabe des Ereignisses:

ON DELETE (Lösung)

ON UPDATE (Änderung)

Angabe des Verhaltens:

NO ACTION (der Default: Fehlermeldung)

CASCADE (Zeile mit Fremdschlüssel wird mitgelöscht)

SET NULL (Fremdschlüsseleintrag wird auf Null gesetzt)

SET DEFAULT (Fremdschlüsseleintrag wird auf einen Default Wert gesetzt)

Beispiel für ON UPDATE CASCADE



```
CREATE TABLE escooter (
...
FOREIGN KEY (employeeid) REFERENCES
employee(eid)
ON UPDATE CASCADE )
;
```

Ändert sich die ID eines Mitarbeiters (eid), wird die geänderte ID in der Relation escooter übernommen und ein konsistenter Datensatz sichergestellt.

Beispiel für ON DELETE CASCADE



```
CREATE TABLE escooter (
...
FOREIGN KEY (employeeid) REFERENCES
employee(eid)
ON DELETE CASCADE )
;
```

Wird ein Mitarbeiter gelöscht, werden auch alle seine eScooter gelöscht

Default Werte

- Spalten können einen Default Wert erhalten
- Wenn ein neues Tupel eingefügt wird und kein Wert für das Attribut angegeben wird, wird der Default Wert genutzt
- Falls kein Default Wert spezifiziert wird, ist NULL der Default Wert

```
CREATE TABLE products(  
product_no integer PRIMARY KEY,  
price numeric DEFAULT 9.99 );
```

IDs / eindeutige Zahlen

- Oft muss man eindeutige Zahlen generieren, z.B. ScooterID für eScooter (künstlicher Primärschlüssel)
- Theoretisch könnte man das aktuelle Minimum in der Tabelle abfragen, und eins dazu addieren. Dies ist aber bei Mehrbenutzerbetrieb schwierig. Deswegen enthalten viele Datenbanksysteme einen Sequenzgenerator

```
ScooterID INTEGER GENERATED ALWAYS AS IDENTITY PRIMARY KEY
```

GENERATED ALWAYS bedeutet, dass man keinen Wert für diese Spalte beim Einfügen eines Datensatzes festlegen kann

GENERATED AS IDENTITY ist die SQL-Standard konforme Variante einer SERIAL Spalte

Ändern von Tabellen / Relationen

Tabellen (d.h. ihr Schema nicht ihr Inhalt) lassen sich mittels **ALTER TABLE** ändern.

Hinzufügen eines Attributs/Spalte:

```
ALTER TABLE <Name der Tabelle>
```

```
ADD COLUMN <Name des Attributs> datentyp
```

Ändern des Datentyps eines Attributs:

```
ALTER TABLE <Name der Tabelle>
```

```
ALTER COLUMN <Name des Attributs> TYPE datentyp
```

Ändern des Spaltennamens

Mittels ALTER Table lässt sich ebenfalls der Spaltenname ändern. Solche Schemaänderungen werden in der Praxis vermieden, da ggf. Anpassungen in der Applikation vorgenommen werden müssen.

```
ALTER TABLE <Name der Tabelle>
RENAME <COLUMN_OLD> TO <COLUMN_NEW>
```

DELETE / TRUNCATE

Tabellen lassen sich mittels DROP TABLE löschen. Tabelleninhalte lassen sich mittels TRUNCATE TABLE löschen. Die Schemadefinition bleibt dann jedoch erhalten.

```
DROP TABLE <Name der Tabelle>
```

```
DROP TABLE <schemaname>.<Name der Tabelle>
```

```
TRUNCATE TABLE <Name der Tabelle>
```

```
TRUNCATE TABLE <schemaname>.<Name der Tabelle>
```

Data Manipulation Language



Definition DML

Die Data Manipulation Language (DML) stellt Befehle zum Einfügen, Ändern und Löschen von Tupeln bereit. RDBMS überprüft bei Einfügen, Ändern und Löschen bestehende Integritätsbedingungen, z.B. Primär- oder Fremdschlüssel. Bei einer Verletzung wird die Ausführung des Befehls verweigert.



Im Kern die Befehle **INSERT, UPDATE, DELETE**



INSERT INTO

In SQL gibt es zwei Methoden zum Einfügen von Daten in eine Tabelle:

1. Ohne Angabe der Spaltennamen. Dann ist die Reihenfolge der Schemadefinition entscheidend
2. Spezifikation der Spaltennamen. Hier kann dann beliebige Reihenfolge gewählt werden

```
INSERT INTO escooter VALUES (1, '702', 'Segway', 1, true);
```

```
INSERT INTO employee (eid, vorname, nachname) VALUES (1, 'Max', 'Mayr');
```



UPDATE

Bereits existierende Tupel lassen sich mittels des UPDATE Kommandos verändern.

```
UPDATE <Tabelle>
SET <Attribut> = <Wert>, ...
WHERE <Bedingungen>
```

```
UPDATE Customer
SET ispremium = 1
WHERE custid = 7;
```

Ändere Premium Status von Kunde 7 von 0 (kein Premiumkunde) auf 1 (Premiumkunde).

Löschen von Datensätzen

DELETE

Bereits existierende Tupel lassen sich mittels des DELETE Kommandos löschen.

```
DELETE FROM <Tabelle>
WHERE <Bedingungen>
```

```
DELETE FROM customer
WHERE city = 'Stuttgart'
```

Lösche alle Kunden aus Stuttgart

Data Query Language (DQL)



Definition DQL

Die DQL (Data Query Language) stellt Sprachelemente zum Lesen von gespeicherten Daten zur Verfügung. Einziger Befehl dieses Sprachbestandteils von SQL ist die SELECT-Anweisung.

Beispiel SQL-Syntax

```
SELECT <Attribute>
FROM <Tabellen>
WHERE <Bedingungen>
```

Selektion und Projektion

SELECT-Klausel: ist zwingend notwendig; die hierbei spezifizierten Attribute bewirken eine Projektion.
Falls keine Projektion gewünscht ist, kann das Symbol * anstelle der Attributnamen angegeben werden

Beispiel

```
SELECT <Attribute>  
FROM <Tabellen>  
WHERE <Bedingungen>
```

Projection

| | | | |
|--|--|--|--|
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

Table 1

```
SELECT *  
FROM <Tabellen>  
WHERE <Bedingungen>
```

Selection

| | | | |
|--|--|--|--|
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

Table 1

Beispielabfrage: „Finde die Namen aller weiblichen Kunden“

```
SELECT name  
FROM customer  
WHERE gender = 'f'
```

DISTINCT

Mit dem Befehl DISTINCT werden Redundanzen, die in der Tabelle auftreten können, eliminiert.

Beispiel

```
SELECT birthday  
FROM customer
```

| birthday |
|------------|
| 04.05.1983 |
| 10.12.2006 |
| 20.08.1980 |
| 20.08.1980 |
| 01.03.1998 |

```
SELECT DISTINCT birthday  
FROM customer
```

| birthday |
|------------|
| 04.05.1983 |
| 10.12.2006 |
| 20.08.1980 |
| 01.03.1998 |

ORDER BY

Abfrageergebnisse können mittels ORDER BY sortiert werden. Natürliche Ordnung der Attribute wird gemäß ihres Datentyps verwendet, d.h. numerische Attribute (z.B. int und float) werden nach numerischen Werten, textuelle Attribute (z.B. varchar) lexikografisch sortiert. Angabe von ASC und DESC bestimmt, ob nach einem Attribut aufsteigend oder absteigend sortiert wird.

Beispiel

```
SELECT FirstName, MiddleName, LastName
FROM Person
ORDER BY FirstName ASC,
LastName DESC;
```

| | FirstName | MiddleName | LastName |
|----|-----------|------------|------------|
| 1 | A. | Scott | Wright |
| 2 | A. | Francesca | Leonetti |
| 3 | A. Scott | NULL | Wright |
| 4 | Aaron | A | Zhang |
| 5 | Aaron | M | Young |
| 6 | Aaron | C | Yang |
| 7 | Aaron | L | Wright |
| 8 | Aaron | L | Washington |
| 9 | Aaron | V | Wang |
| 10 | Aaron | NULL | Simmons |
| 11 | Aaron | J | Sharma |
| 12 | Aaron | NULL | Shan |
| 13 | Aaron | C | Scott |
| 14 | Aaron | NULL | Russell |
| 15 | Aaron | N | Ross |
| 16 | Aaron | NULL | Roberts |
| 17 | Aaron | NULL | Powell |
| 18 | Aaron | R | Phillips |
| 19 | Aaron | NULL | Perry |
| 20 | Aaron | L | Perez |
| 21 | Aaron | W | Patterson |
| 22 | Aaron | NULL | Nicholls |
| 23 | Aaron | R | Nelson |
| 24 | Aaron | W | Mitchell |

Übersicht

Aggregatfunktionen führen Berechnungen für verschiedene Werte durch und geben einen einzelnen Wert zurück. Alle Aggregatfunktionen, außer COUNT(*), ignorieren NULL-Werte. Aggregatfunktionen werden häufig mit der GROUP BY-Klausel der SELECT-Anweisung verwendet.

COUNT(): Zählen

COUNT(DISTINCT A): Zählen Anzahl verschiedener Werte

SUM(): Aufsummieren

AVG(): Durchschnitt bilden

MAX(A,B): Maximum der Attribute A und B

SQL unterstützt zahlreiche Funktionen für nichtnumerische Datentypen, etwa für Zeichenketten

- **LOWER(A): Zeichenkette in Kleinbuchstaben**
- **UPPER(A): Zeichenkette in Großbuchstaben**
- **LENGTH(A): Länge der Zeichenkette**
- **SUBSTRING: (A, start, end): Ausschnitt der Zeichenkette**
- **TRIM(A): Zeichenkette ohne umgebende Leerzeichen**

Beispiel

Wie viele Kunden gibt es?

```
SELECT COUNT (custid)  
FROM customer
```

Beispiel

Wie viele **unterschiedliche** Geburtstage gibt es?

```
SELECT COUNT (distinct Birthday)  
FROM customer
```

Welcher Kunde hat die größte Kundensumme?

```
SELECT Nachname, MAX(CustID)  
FROM Customer;
```



Diese Anfrage wird nicht funktionieren. Aggregatfunktionen reduzieren alle Werte einer Spalte zu einem einzigen Wert:

- Für das Attribut CustID wird das Maximum ermittelt
- Für das Attribut Nachname ist nicht klar, wie die ganzen verschiedenen Namen auf einen reduziert werden sollen

Min/Max mit geschachtelter Abfrage

Welcher Kunde hat die größte Kundennummer?

```
SELECT Nachname, CustID  
FROM Customer  
WHERE CustID =  
    (SELECT MAX(CustID)  
     FROM Customer);
```

Geschachtelte Anfragen

- Anfragen können in anderen Anfragen geschachtelt sein, d.h. es kann mehr als eine SELECT-Klausel geben
- Geschachteltes SELECT kann in der WHERE-Klausel, in der FROM-Klausel und sogar in einer SELECT-Klausel selbst auftauchen
- Im Prinzip wird in der inneren Anfrage ein Zwischenergebnis berechnet, das in der äußeren benutzt wird

Definition

Man unterscheidet zwei verschiedene Arten von Anfragen:

- **unkorrelierte Anfragen:** Unteranfrage bezieht sich nur auf die eigenen Attribute
- **korrelierte Anfragen:** Unteranfrage bezieht sich auch auf Attribute der äußeren Anfrage

Beispiel

Gebe die ID und Namen aller Mitarbeiter, die Rolle vom Typen Pure-Air warten.

```
SELECT EmployeeID, Lastname  
FROM Employee E  
WHERE E.EmployeeID IN  
    (SELECT S.EmployeeID  
     FROM eScooter S  
     WHERE S.Brand = 'Pure Air');
```

Unteranfrage wird einmal ausgewertet. Für jedes Tupel der äußeren Anfrage wird dann geprüft, ob die EmployeeID im Ergebnis der Unteranfrage vorkommt.

Beispiel

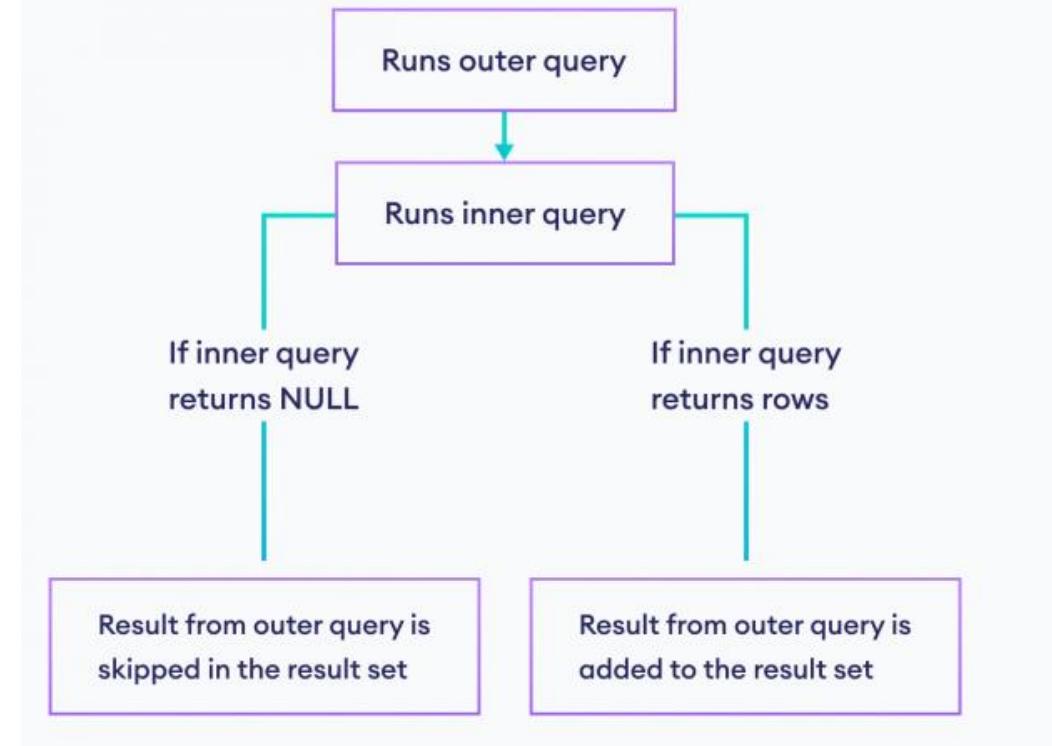
Finde alle Kunden, die genau zwei mal einen Scooter gefahren sind.

```
SELECT C.CustID, C.Lastname  
FROM Customer C  
WHERE 2 =  
    (SELECT COUNT(*)  
     FROM RIDE R  
     WHERE R.CustID = C.CustID);
```

- Unteranfrage ist abhängig von ihrer übergeordneten Anweisung, da sie mindestens eine Spalte von ihr benötigt
- Unteranfrage wird für jede in Frage kommende Zeile der Oberabfrage genau einmal ausgeführt

```
SELECT spaltennamen  
FROM table  
WHERE EXISTS  
(SELECT spaltennamen FROM table_name  
WHERE condition)
```

Working of SQL EXISTS operator



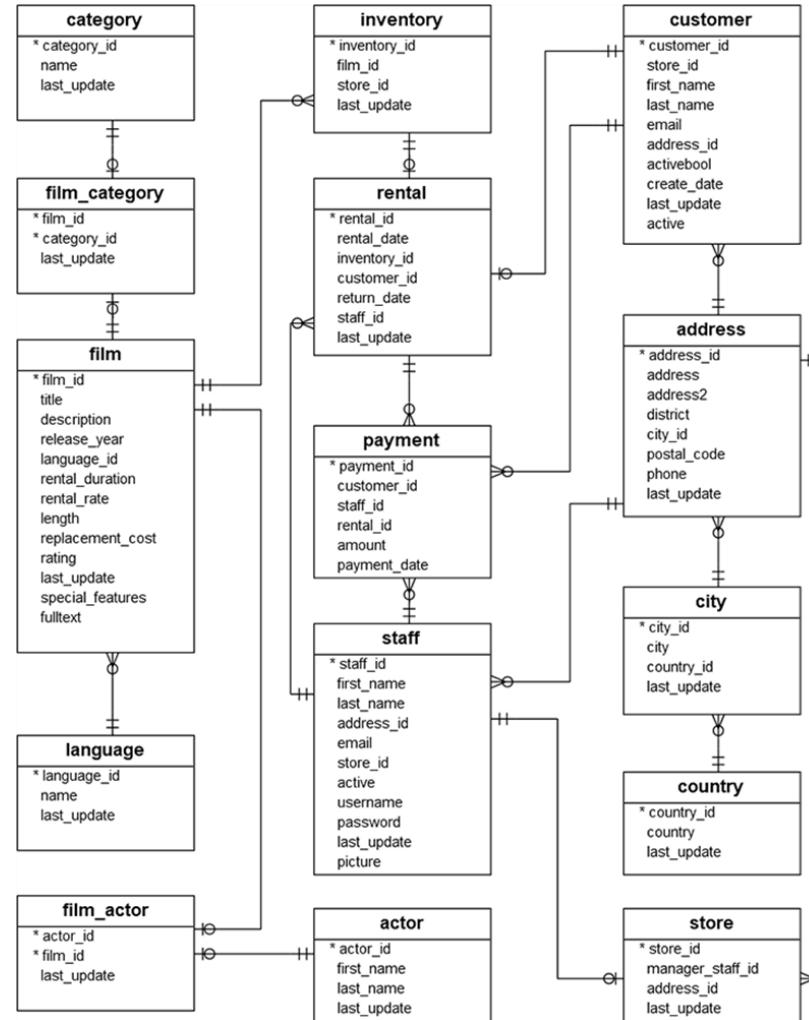
<https://www.programiz.com/sql/exists>

Beispiel



```
SELECT * FROM Kunden WHERE EXISTS (SELECT * FROM Bestellungen WHERE Bestellungen.KundenID = Kunden.KundenID)
```

Wiederholung an einem neuen Beispiel



GROUP BY

Die Gruppierung in SQL ermöglicht es, Zeilen gruppenweise zusammenzufassen. Die Gruppierung funktioniert in SQL mit dem Schlüsselwort GROUP BY. Wenn in der Projektion Spalten und Aggregatfunktion auftreten, müssen alle Spalten, die nicht Teil der Aggregatfunktion sind, in die group-by Komponente aufgenommen werden.

```
SELECT CITY, count(custid)
FROM customer
GROUP BY City
```

- Tupel werden in verschiedene Gruppen aufgeteilt
- **Pro Gruppe wird die Aggregatfunktion angewendet**

Wie viele Kunden gibt es pro Stadt?

```
SELECT City, COUNT(CustID)  
FROM Customer  
GROUP BY City;
```

- Tupel werden in verschiedene Gruppen aufgeteilt
- Pro Gruppe wird die Aggregatfunktion angewendet



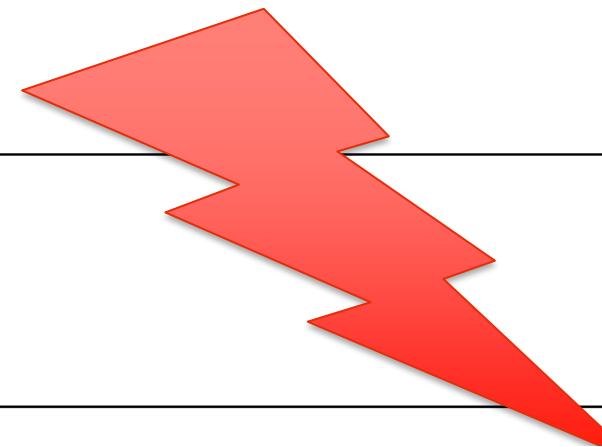
| | stadt | count |
|---|------------|-------|
| 1 | Stuttgart | 1 |
| 2 | Fürth | 1 |
| 3 | Hamburg | 3 |
| 4 | Ingolstadt | 1 |

Gruppieren - Merke

ERROR: column "MY_TABLE.MY_COLUMN" must appear in the GROUP BY clause or be used in an aggregate function

```
SELECT CustID, VehID, COUNT(*)  
FROM Ride  
GROUP BY CustID;
```

Falsch!



```
SELECT CustID, VehID, COUNT(*)  
FROM Ride  
GROUP BY CustID, VehID;
```

Wenn in der Projektion Spalten und Aggregatfunktion auftreten, müssen alle Spalten in die group-by Komponente aufgenommen werden (außer Aggregationsfunktion)

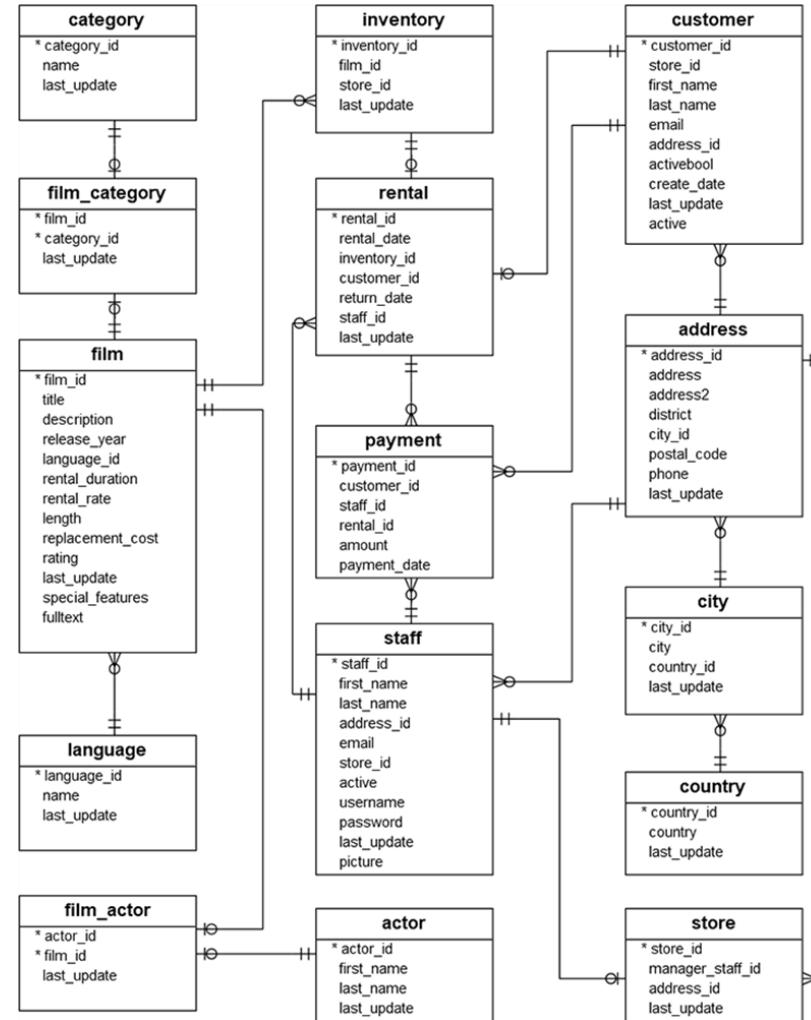
HAVING

Wenn nach dem Gruppieren weiter gefiltert werden soll, muss die HAVING-Klausel eingesetzt werden, weil WHERE-Klausel wird vor dem Gruppieren ausgewertet.

```
SELECT custid, count(startride) AS anzahl  
FROM ride  
GROUP BY custid  
Having count (*) > 3
```

Umbenennung Ergebnisspalte
(Aliase)

Count (*) gibt die Anzahl der Items einer Gruppe aus





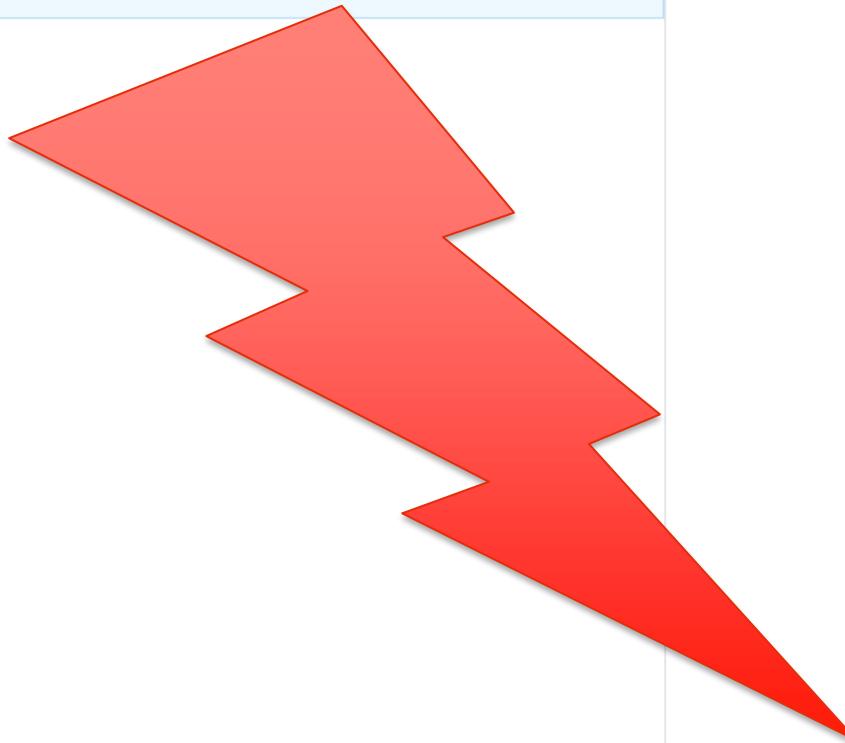
dvdrental/postgres@PostgreSQL 14 ▾

Query Editor Query History

Scratch Pac



```
1 select film_id, store_id, count(*) AS Anzahl from inventory
2 GROUP BY film_id, store_id
3 ORDER BY Anzahl
4 HAVING count(*) > 3
5
6
```



Data Output Explain Messages Notifications

ERROR: FEHLER: Syntaxfehler bei »HAVING«
LINE 4: HAVING count(*) > 3
 ^

SQL state: 42601
Character: 104



Query Editor Query History

```
1 select film_id, store_id, count(*) AS Anzahl from inventory
2 GROUP BY film_id, store_id
3 HAVING count(*) > 2
4 ORDER BY Anzahl
5
```



Data Output Explain Messages Notifications

| | film_id smallint | store_id smallint | anzahl bigint | |
|---|---------------------|----------------------|------------------|--|
| 1 | 37 | 2 | 3 | |
| 2 | 277 | 2 | 2 | |

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
HAVING condition
ORDER BY column_name(s);
```

datacamp

SQL for Data Science

SQL Basics Cheat Sheet

Learn SQL online at www.DataCamp.com

What is SQL?

SQL stands for “structured query language”. It is a language used to query, analyze, and manipulate data from databases. Today, SQL is one of the most widely used tools in data.

> The different dialects of SQL

Although SQL languages all share a basic structure, some of the specific commands and styles can differ slightly. Popular dialects include MySQL, SQLite, SQL Server, Oracle SQL, and more. PostgreSQL is a good place to start —since it's close to standard SQL syntax and is easily adapted to other dialects.

> Sample Data

Throughout this cheat sheet, we'll use the columns listed in this sample table of `airbnb_listings`

| airbnb_listings | | | | |
|-----------------|----------|---------|-----------------|-------------|
| id | city | country | number_of_rooms | year_listed |
| 1 | Paris | France | 5 | 2018 |
| 2 | Tokyo | Japan | 2 | 2017 |
| 3 | New York | USA | 2 | 2022 |

> Querying tables

- Get all the columns from a table


```
SELECT *
FROM airbnb_listings;
```
- Return the city column from the table


```
SELECT city
FROM airbnb_listings;
```
- Get the city and year_listed columns from the table


```
SELECT city, year_listed
FROM airbnb_listings;
```
- Get the listing id, city, ordered by the number_of_rooms in ascending order


```
SELECT id, city
FROM airbnb_listings
ORDER BY number_of_rooms ASC;
```

```
5. Get the listing id, city, ordered by the number_of_rooms in descending order
SELECT id, city
FROM airbnb_listings
ORDER BY number_of_rooms DESC;

6. Get the first 5 rows from the airbnb_listings table
SELECT *
FROM airbnb_listings
LIMIT 5;

7. Get a unique list of cities where there are listings
SELECT DISTINCT city
FROM airbnb_listings;
```

> Filtering Data

Filtering on numeric columns

- Get all the listings where `number_of_rooms` is more or equal to 3


```
SELECT *
FROM airbnb_listings
WHERE number_of_rooms >= 3;
```
- Get all the listings where `number_of_rooms` is more than 3


```
SELECT *
FROM airbnb_listings
WHERE number_of_rooms > 3;
```
- Get all the listings where `number_of_rooms` is exactly equal to 3


```
SELECT *
FROM airbnb_listings
WHERE number_of_rooms = 3;
```
- Get all the listings where `number_of_rooms` is lower or equal to 3


```
SELECT *
FROM airbnb_listings
WHERE number_of_rooms <= 3;
```
- Get all the listings where `number_of_rooms` is lower than 3


```
SELECT *
FROM airbnb_listings
WHERE number_of_rooms < 3;
```
- Get all the listings with 3 to 6 rooms


```
SELECT *
FROM airbnb_listings
WHERE number_of_rooms BETWEEN 3 AND 6;
```

Filtering on text columns

- Get all the listings that are based in ‘Paris’


```
SELECT *
FROM airbnb_listings
WHERE city = 'Paris';
```
- Get the listings based in the ‘USA’ and in ‘France’


```
SELECT *
FROM airbnb_listings
WHERE country IN ('USA', 'France');
```
- Get all the listings where the city starts with ‘J’ and where the city does not end in ‘t’


```
SELECT *
FROM airbnb_listings
WHERE city LIKE 'J%' AND city NOT LIKE '%t';
```

Filtering on multiple columns

- Get all the listings in ‘Paris’ where `number_of_rooms` is bigger than 3


```
SELECT *
FROM airbnb_listings
WHERE city = 'Paris' AND number_of_rooms > 3;
```
- Get all the listings in ‘Paris’ OR the ones that were listed after 2012


```
SELECT *
FROM airbnb_listings
WHERE city = 'Paris' OR year_listed > 2012;
```

Filtering on missing data

- Return the listings where `number_of_rooms` is missing


```
SELECT *
FROM airbnb_listings
WHERE number_of_rooms IS NULL;
```
- Return the listings where `number_of_rooms` is not missing


```
SELECT *
FROM airbnb_listings
WHERE number_of_rooms IS NOT NULL;
```

> Aggregating Data

Simple aggregations

- Get the total number of rooms available across all listings


```
SELECT SUM(number_of_rooms)
FROM airbnb_listings;
```
- Get the average number of rooms per listing across all listings


```
SELECT AVG(number_of_rooms)
FROM airbnb_listings;
```
- Get the listing with the highest number of rooms across all listings


```
SELECT MAX(number_of_rooms)
FROM airbnb_listings;
```
- Get the listing with the lowest number of rooms across all listings


```
SELECT MIN(number_of_rooms)
FROM airbnb_listings;
```

Grouping, filtering, and sorting

- Get the total number of rooms for each country


```
SELECT country, SUM(number_of_rooms)
FROM airbnb_listings
GROUP BY country;
```
- Get the average number of rooms for each country


```
SELECT country, AVG(number_of_rooms)
FROM airbnb_listings
GROUP BY country;
```
- Get the listing with the maximum number of rooms per country


```
SELECT country, MAX(number_of_rooms)
FROM airbnb_listings
GROUP BY country;
```
- Get the listing with the lowest amount of rooms per country


```
SELECT country, MIN(number_of_rooms)
FROM airbnb_listings
GROUP BY country;
```
- For each country, get the average number of rooms per listing, sorted by ascending order


```
SELECT country, AVG(number_of_rooms) AS avg_rooms
FROM airbnb_listings
GROUP BY country
ORDER BY avg_rooms ASC;
```
- For Japan and the USA, get the average number of rooms per listing in each country


```
SELECT country, MAX(number_of_rooms)
FROM airbnb_listings
WHERE country IN ('USA', 'Japan')
GROUP BY country;
```

- Get the number of cities per country, where there are listings


```
SELECT country, COUNT(city) AS number_of_cities
FROM airbnb_listings
GROUP BY country;
```
- Get all the years where there were more than 100 listings per year


```
SELECT year_listed
FROM airbnb_listings
GROUP BY year_listed
HAVING COUNT(id) > 100;
```



Vergleichsoperatoren

Prädikate in der where-Klausel können logisch kombiniert werden: AND, OR, NOT
Als Vergleichsoperatoren können verwendet werden: =, >, <, >=, <= between, like

Beispiel

Gib mir alle Kunden aus, die zwischen 01.01.2000 und 01.01.2011 geboren wurden.

```
SELECT * FROM customer
WHERE Birthday BETWEEN 01.01.2000 AND 01.01.2011
```

IN-Operator

Der IN-Operator prüft, ob ein zu vergleichender Ausdruck in einer Menge von Werten vorkommt. Der IN-Operator hat den Vorteil der besseren Lesbarkeit. Beachten Sie, dass IN-Operatoren in der Praxis aus Performancegründen vermieden werden.

SQL-Syntax

```
SELECT column1, column2, ...
FROM table
WHERE column1 IN (value1, value2,
value3,...)
```

```
SELECT column1, column2, ...
FROM table
WHERE column1 IN (SELECT query)
```

Beispiel IN-Operator

Beispiel

Gebe alle Kunden aus Regensburg, Ingolstadt, Pfaffenhofen.

```
SELECT * FROM customer  
WHERE city IN ('Regensburg', 'Ingolstadt', 'Pfaffenhofen')
```

| Id | Firstname | Lastname | Phone | Birthday | Gender | City | isPremium |
|----|-----------|-----------|------------|------------|--------|--------------|-----------|
| 1 | Homer | Bart | 0175445566 | 03.02.2010 | m | Ingolstadt | 0 |
| 2 | Bart | Gallagher | 0175445566 | 05.02.2008 | m | Ingolstadt | 0 |
| 3 | Lisa | Hamilton | 0173441111 | 01.02.1986 | w | Ingolstadt | 0 |
| 4 | Apu | Thompson | 0175445566 | 01.12.1995 | m | Pfaffenhofen | 1 |
| 6 | Edna | Mays | 0175863678 | 01.05.1986 | w | Ingolstadt | 1 |

LIKE-Operator

Mittels des LIKE-Operators können Zeichenketten gesucht werden. Hierbei gibt es folgende Parametereinstellungen:

- L%S: Alle Zeichenketten, die mit L beginnen und mit S enden
- L__S mit L beginnen, zwei Zeichen in der Mitte haben und mit S enden
- A% : Alle Zeichenketten, die mit A beginnen
- %UNG: Alle Zeichenketten, die mit UNG enden
- %ST% : Alle Zeichen, die an irgendeiner Stelle das Muster ST enthalten

Beachten Sie, dass die Suchstrings *CASE-sensitive* sind!

SQL-Syntax

```
SELECT column1, column2, ...
FROM table
WHERE column1 LIKE '%Suchstring'
```

```
SELECT column1, column2, ...
FROM table
WHERE UPPER(column1) LIKE '%SUCHSTRING'
```

Beispiel LIKE-Operator

Beispiel

Gebe alle Kunden aus, die mit dem Buchstaben B anfangen.

```
SELECT * FROM customer  
WHERE lastname LIKE 'B%'
```

IS-Operator

In SQL gibt es den speziellen Wert Null (oder NULL), der für jeden Datentypen gilt. Null steht für unbekannte, nicht verfügbare oder nicht anwendbare Werte. Auf Null wird mit dem Operator IS geprüft.

Beispiel

```
SELECT *
FROM customer
WHERE Birthday IS NULL
```

LINKS

| A | B |
|---|---|
| 1 | 2 |
| 2 | 3 |

RECHTS

| B | C |
|---|---|
| 3 | 4 |
| 4 | 5 |

\bowtie

| A | B | C |
|---|---|---|
| 2 | 3 | 4 |

\bowtie^C

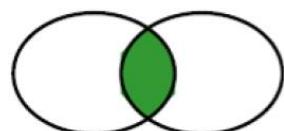
| A | B | C |
|---------|---|---------|
| 1 | 2 | \perp |
| 2 | 3 | 4 |
| \perp | 4 | 5 |

\bowtie^L

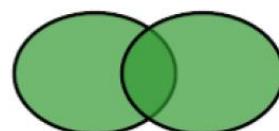
| A | B | C |
|---|---|---------|
| 1 | 2 | \perp |
| 2 | 3 | 4 |

\bowtie^R

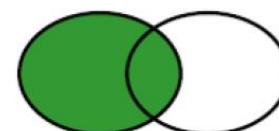
| A | B | C |
|---------|---|---|
| 2 | 3 | 4 |
| \perp | 4 | 5 |



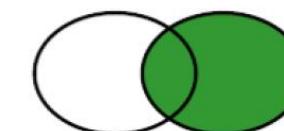
natural join
inner join



full outer join



left outer join



right outer join

Natürlicher / INNER JOIN

```
SELECT *
FROM employee, escooter
WHERE employee.eid = escooter.employeedid
```

```
SELECT *
FROM employee JOIN escooter
ON (employee.eid = escooter.employeedid)
```

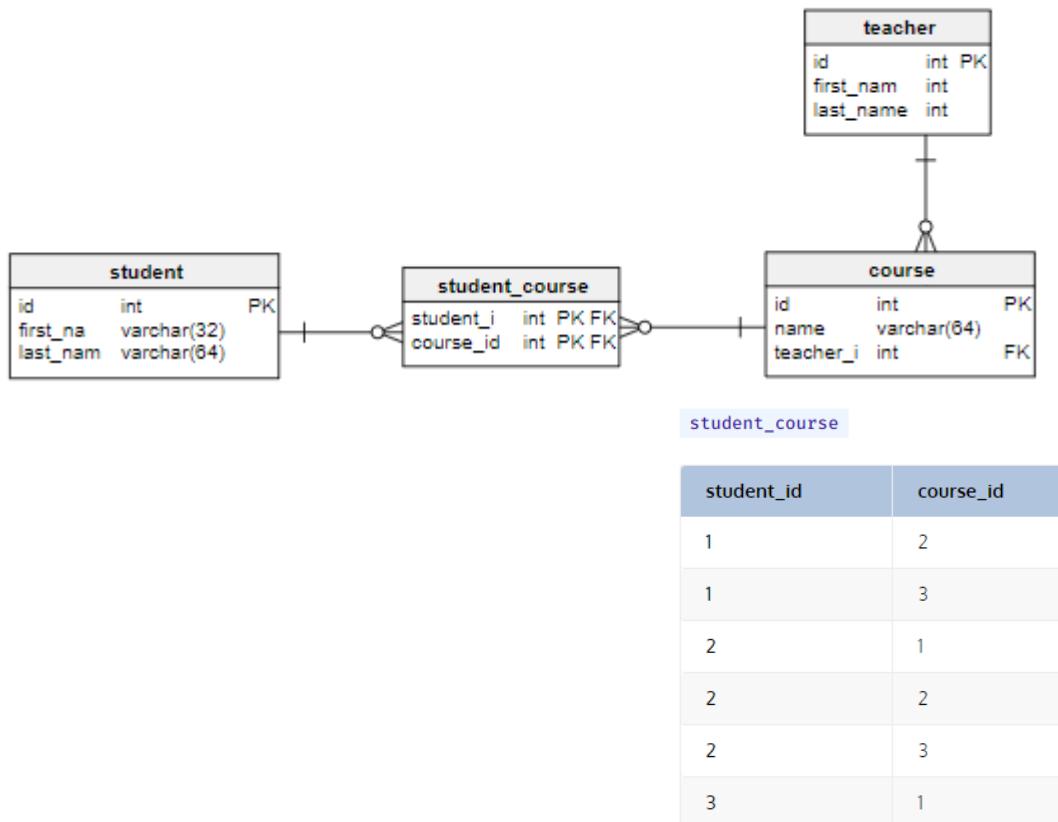
```
SELECT * FROM employee NATURAL JOIN escooter;
```

```
SELECT table1.column1, table1.column2, table2.column1, ....
FROM table1
INNER JOIN table2
ON table1.matching_column = table2.matching_column;
```

Eine Umbenennung der Relationen ist ebenfalls möglich und dies verbessert die Lesbarkeit des SQL-Statements.

```
SELECT *
FROM Customer C, eScooter S, Ride R
WHERE C.CustId = R.CustId
AND S.ScooterID = R.ScooterID
```

JOINS über mehrere Relationen



student

| id | first_name | last_name |
|----|------------|-----------|
| 1 | Shreya | Bain |
| 2 | Rianna | Foster |
| 3 | Yosef | Naylor |

course

| id | name | teacher_id |
|----|--------------------|------------|
| 1 | Database design | 1 |
| 2 | English literature | 2 |
| 3 | Python programming | 1 |

Gebe mir Vorname, Nachname und Kurs der Studierenden aus



```
SELECT
    student.first_name,
    student.last_name,
    course.name
FROM student
JOIN student_course
    ON student.id = student_course.student_id
JOIN course
    ON course.id = student_course.course_id
```



Result Set

| first_name | last_name | name |
|------------|-----------|--------------------|
| Shreya | Bain | English literature |
| Shreya | Bain | Python programming |
| Rianna | Foster | Database design |
| Rianna | Foster | English literature |
| Rianna | Foster | Python programming |
| Yosef | Naylor | Database design |

- Wenn in einer FROM Klausel mehrere Relationen stehen, werden diese mit einem Kreuzprodukt verbunden
- Es werden also alle Kombinationen ausgegeben

```
SELECT *
FROM Employee, eScooter
```

oder

```
SELECT *
FROM Employee CROSS JOIN eScooter
```

Klassische Operationen der Mengenlehre gibt es auch in SQL

Vereinigungsmenge: *UNION / UNION ALL*

Schnittmenge: *INTERSECT*

Differenzmenge: *EXCEPT*

```
SELECT Firstname, Lastname, Birthday  
FROM Customer  
UNION  
SELECT Firstname, Lastname, Birthday  
FROM Employee
```

| Customer | | | UNION | Employee | | |
|-----------|-----------|------------|-------|-----------|----------|------------|
| Firstname | Lastname | Birthday | | Firstname | Lastname | Birthday |
| Homer | Bart | 03.02.2010 | | Peter | Griffin | 03.12.1985 |
| Bart | Gallagher | 05.02.2008 | | Bonnie | Hamilton | 30.09.2001 |
| ↓ | | | | | | |
| Firstname | | | | | | |
| Homer | Bart | 03.02.2010 | | | | |
| Bart | Gallagher | 05.02.2008 | | | | |
| Peter | Griffin | 03.12.1985 | | | | |
| Bonnie | Hamilton | 30.09.2001 | | | | |

Duplikateliminierung

- Im Gegensatz zu SELECT eliminiert UNION automatisch Duplikate
- Falls Duplikate im Ergebnis erwünscht sind, muss der UNION ALL-Operator benutzt werden

Beispiel Schnittmenge

Welche Personen sind sowohl Mitarbeiter als auch Kunden?

```
SELECT Firstname, Lastname  
FROM Customer  
INTERSECT  
SELECT Firstname, Lastname  
FROM Employee
```

| Customer | | Employee | |
|-----------|-----------|-----------|----------|
| Firstname | Lastname | Firstname | Lastname |
| Homer | Bart | Peter | Griffin |
| Bart | Gallagher | Homer | Bart |

INTERSECT

↓

| Firstname | Lastname |
|-----------|----------|
| Homer | Bart |

Beispiel Differenzmenge

Welche Personen sind auf der ersten Liste (Kunden), aber nicht auf der zweiten Liste (Mitarbeiter)?

```
SELECT Firstname, Lastname  
FROM Customer  
EXCEPT  
SELECT Firstname, Lastname  
FROM Employee
```

| Customer | | Employee | |
|-----------|-----------|-----------|----------|
| Firstname | Lastname | Firstname | Lastname |
| Homer | Bart | Peter | Griffin |
| Bart | Gallagher | Homer | Bart |

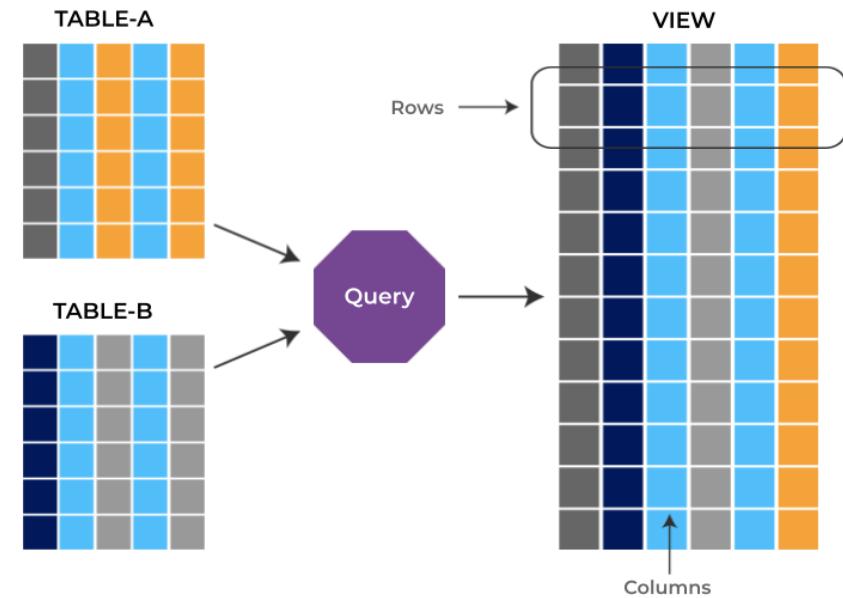
EXCEPT

↓

| Firstname | Lastname |
|-----------|-----------|
| Bart | Gallagher |

Begriffsbestimmung

Eine View ist eine virtuelle Tabelle, die über eine gespeicherte Abfrage definiert wird. Der Nutzer kann diese virtuelle Tabelle wie eine normale Relation abfragen; das Ergebnis (Sicht) wird bei jeder Abfrage neu berechnet.



```
CREATE VIEW View-Name AS  
<Anfrage-Ausdruck>
```

Beispiel View

Kundendaten von Kunden, die zwar angemeldet sind, aber noch nie mit einem Scooter gefahren sind.

```
CREATE VIEW CustomerWithoutRides AS
SELECT *
FROM Customer C
WHERE C.CustID NOT IN (
SELECT CustID
FROM Rides )
```

Informationen können jetzt direkt über die View angefragt werden:

```
SELECT Firstname, Lastname, Phone
FROM CustomerWithoutRides
```



- INSERT, UPDATE und DELETE Befehle sind auf Views genauso erlaubt wie auf einer regulären Tabelle
- die Übertragung der Veränderungen von der virtuellen Relation auf die wirkliche Relation kann schwierig werden
- Bei INSERT Eingaben ist beispielsweise nicht klar, wie die Attribute eines Tupels in der realen Relation angepasst werden sollen, die in der virtuellen Relation nicht vorkamen

- SQL ist eine deklarative Abfragesprache: sie erlaubt keine Steuerung des Ablaufs wie es ein `if` in imperativen Programmen macht
- Trotzdem gibt es den CASE Ausdruck: das Ergebnis von CASE ist abhängig von einer oder mehrerer Bedingungen

```
CASE WHEN <Bedingung> THEN <Ergebnis>
      [WHEN <Bedingung> THEN <Ergebnis>
       ...]
      [ELSE <Ergebnis>]
END [AS spaltennameergebnis]
```

Beispiel

```
SELECT OrderID, Quantity,  
CASE  
    WHEN Quantity > 30 THEN 'The quantity is greater than 30'  
    WHEN Quantity = 30 THEN 'The quantity is 30'  
    ELSE 'The quantity is under 30'  
END AS QuantityText  
FROM OrderDetails;
```



SQL Statement:

```
SELECT OrderID, Quantity,  
CASE WHEN Quantity > 30 THEN 'The quantity is greater than 30'  
WHEN Quantity = 30 THEN 'The quantity is 30'  
ELSE 'The quantity is under 30'  
END AS QuantityText  
FROM OrderDetails;
```

Edit the SQL Statement, and click "Run SQL" to see the result.

[Run SQL »](#)

Result:

Number of Records: 2155

| OrderID | Quantity | QuantityText |
|---------|----------|---------------------------------|
| 10248 | 12 | The quantity is under 30 |
| 10248 | 10 | The quantity is under 30 |
| 10248 | 5 | The quantity is under 30 |
| 10249 | 9 | The quantity is under 30 |
| 10249 | 40 | The quantity is greater than 30 |
| 10250 | 10 | The quantity is under 30 |
| 10250 | 35 | The quantity is greater than 30 |
| 10250 | 15 | The quantity is under 30 |

<https://www.w3schools.com/sql/>