

# Session 8 – SQL Data Definition (DDL) & Manipulation (DML)

Session-Typ: Lecture Dauer: 90 Minuten Lernziele: LZ 2 (SQL-Praxis)

## Intro: Von Abfragen zu Strukturen

Bisher haben Sie gelernt, Daten abzufragen – SELECT, WHERE, GROUP BY, alles in Session 7. Aber wie kommen die Tabellen überhaupt in die Datenbank? Wie definieren Sie Spalten, Datentypen, Constraints? Und wie fügen Sie Daten ein, ändern sie, löschen sie? Das ist der nächste Schritt: Von der Abfrageebene zur Strukturebene.

Heute lernen Sie:

- **DDL (Data Definition Language):** CREATE, ALTER, DROP – Ihre Werkzeuge für Schema-Design
- **DML (Data Manipulation Language):** INSERT, UPDATE, DELETE – Daten schreiben, nicht nur lesen
- **Constraints:** PRIMARY KEY, FOREIGN KEY, UNIQUE, CHECK – Datenintegrität sichern

Warum ist das wichtig? Weil Ihre Datenbank nur so gut ist wie Ihr Schema. Falsche Datentypen führen zu Performance-Problemen. Fehlende Constraints führen zu Inkonsistenzen. Unsichere Updates können Ihre gesamte Datenbank zerstören. Diese Session gibt Ihnen die Kontrolle.

## Was ist DDL & DML?

SQL ist keine monolithische Sprache. Es gibt Kategorien: DDL für Schema-Definition, DML für Datenmanipulation, DCL für Zugriffsrechte, TCL für Transaktionen. Heute fokussieren wir DDL und DML – das Fundament für alles Weitere.

### SQL-Kategorien im Überblick

Kategorie	Abkürzung	Zweck	Befehle	Session
Data Definition Language	DDL	Schema erstellen/ändern	<code>CREATE</code> , <code>ALTER</code> , <code>DROP</code>	Heute
Data Manipulation Language	DML	Daten lesen/schreiben	<code>SELECT</code> , <code>INSERT</code> , <code>UPDATE</code> , <code>DELETE</code>	Heute
Data Control Language	DCL	Zugriffsrechte	<code>GRANT</code> , <code>REVOKE</code>	Später
Transaction Control Language	TCL	Transaktionen	<code>BEGIN</code> , <code>COMMIT</code> , <code>ROLLBACK</code>	Session 11+

**Heute:** DDL (Struktur) + DML (Daten)

DDL ist wie der Bauplan Ihres Hauses: Sie definieren Räume (Tabellen), Türen (`Foreign Keys`), Regeln (`Constraints`). DML ist das Leben im Haus: Sie stellen Möbel auf (`INSERT`), verschieben sie (`UPDATE`), werfen sie raus (`DELETE`).

## 1. DDL: Data Definition Language

### Tabellen erstellen `CREATE TABLE`

`CREATE TABLE` ist Ihr wichtigster DDL-Befehl. Sie definieren den Tabellennamen, die Spalten, die Datentypen, die Constraints. Schauen wir uns die Grundsyntax an.

#### Grundsyntax

Minimal-Beispiel:

```

1 CREATE TABLE products (
2   id INTEGER,
3   name TEXT,
4   price DECIMAL(10, 2)
5 );
```

```
CREATE TABLE products (
    id INTEGER,
    name TEXT,
    price DECIMAL(10, 2)
)
```

ok

Was passiert hier?

- Tabelle `products` wird erstellt
- 3 Spalten: `id` (Ganzzahl), `name` (Text), `price` (Dezimalzahl mit 2 Nachkommastellen)
- Keine Constraints – jeder Wert ist erlaubt, auch NULL

Aber das ist zu simpel. In der Praxis wollen Sie mehr Kontrolle: Ein Primärschlüssel, NOT NULL für Pflichtfelder, DEFAULT-Werte. Schauen wir uns eine realistischere Version an.

### Realistisches Beispiel mit Constraints

```
1 CREATE TABLE products (
2     product_id INTEGER PRIMARY KEY,
3     name TEXT NOT NULL,
4     description TEXT,
5     price DECIMAL(10, 2) NOT NULL CHECK (price >= 0),
6     stock INTEGER DEFAULT 0,
7     created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
8 );
```

```
CREATE TABLE products (
    product_id INTEGER PRIMARY KEY,
    name TEXT NOT NULL,
    description TEXT,
    price DECIMAL(10, 2) NOT NULL CHECK (price >= 0),
    stock INTEGER DEFAULT 0,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
)
```

relation "products" already exists

Was ist neu?

- **PRIMARY KEY**: `product_id` ist eindeutig + NOT NULL
- **NOT NULL**: `name` und `price` sind Pflichtfelder
- **CHECK**: `price` muss  $\geq 0$  sein (keine negativen Preise!)
- **DEFAULT**: `stock` ist standardmäßig 0, `created_at` wird automatisch gesetzt

Testen:

```

1 -- Funktioniert:
2 INSERT INTO products (product_id, name, price)
3 VALUES (1, 'Laptop', 999.99);
4
5 -- Funktioniert NICHT (price negativ):
6 INSERT INTO products (product_id, name, price)
7 VALUES (2, 'Mouse', -10.00);

```

-- Funktioniert:  
**INSERT INTO products (product\_id, name, price)  
VALUES (1, 'Laptop', 999.99)**

column "product\_id" of relation "products" does not exist

Sie können Primärschlüssel auch inline definieren oder als separaten Constraint. Beides funktioniert, aber die separate Form ist flexibler – vor allem bei Composite Keys.

### Primärschlüssel: Inline vs. Constraint

Inline (einfach):

```

CREATE TABLE orders (
    order_id INTEGER PRIMARY KEY,
    customer_name TEXT
);

```

Als Constraint (flexibel):

```

CREATE TABLE orders (
    order_id INTEGER,
    customer_name TEXT,
    CONSTRAINT pk_orders PRIMARY KEY (order_id)
);

```

Composite Key (mehrere Spalten):

```

1 CREATE TABLE order_items (
2     order_id INTEGER,
3     product_id INTEGER,
4     ...
5 );

```

```

4     quantity INTEGER,
5     PRIMARY KEY (order_id, product_id)
6 );

```

```

CREATE TABLE order_items (
    order_id INTEGER,
    product_id INTEGER,
    quantity INTEGER,
    PRIMARY KEY (order_id, product_id)
)

```

ok

### Wann Composite Keys?

- Wenn Eindeutigkeit nur durch Kombination gegeben ist
- Beispiel: Ein Produkt kann in mehreren Bestellungen vorkommen, aber pro Bestellung nur einmal

## Datentypen-Überblick

Datentypen sind wichtig für Speichereffizienz, Performance und Validierung. Ein INTEGER braucht weniger Platz als TEXT. Eine DECIMAL-Zahl ist präziser als FLOAT. Datum-Typen ermöglichen Zeitberechnungen. Schauen wir uns die wichtigsten an.

### Numerische Typen

Typ	Bereich	Speicher	Wann nutzen?
INTEGER / INT	-2 <sup>31</sup> bis 2 <sup>31-1</sup>	4 Bytes	IDs, Zähler, ganze Zahlen
BIGINT	-2 <sup>63</sup> bis 2 <sup>63-1</sup>	8 Bytes	Große IDs, Zeitstempel (Unix)
DECIMAL(p,s) 	Präzise Dezimalzahl	Variabel	Geld, Preise (keine Rundungsfehler!)
FLOAT / DOUBLE	Approximativ	4/8 Bytes	Wissenschaftliche Berechnungen

### Beispiel: Warum DECIMAL für Geld?

```

1 -- FLOAT hat Rundungsfehler:
2 SELECT 0.1 + float + 0.2 AS float_sum; -- Ergebnis: 0

```

```

2   SELECT 0.1::float + 0.2::float AS float_sum,    -- Ergebnis: 0
      .30000000000000004
3
4   -- DECIMAL ist präzise:
5   SELECT CAST(0.1 AS DECIMAL(10,2)) + CAST(0.2 AS DECIMAL(10,2)) AS
      decimal_sum;

```

-- FLOAT hat Rundungsfehler:

```
SELECT 0.1::float + 0.2::float AS float_sum
```

#	float_sum
1	0.30000000000000004

1 rows

-- Ergebnis: 0.30000000000000004

-- DECIMAL ist präzise:

```
SELECT CAST(0.1 AS DECIMAL(10,2)) + CAST(0.2 AS DECIMAL(10,2)) AS decimal_sum
```

#	decimal_sum
1	0.30

1 rows

Text-Typen haben verschiedene Längen. VARCHAR begrenzt die Länge, TEXT ist unbegrenzt. In PGlite und PostgreSQL gibt es keinen Performance-Unterschied mehr, aber in älteren Systemen (MySQL) schon.

## Text-Typen

Typ	Max. Länge	Wann nutzen?
CHAR(n)	Fix n Zeichen	Festlängen-Codes (z.B. Ländercodes ,DE‘, ,US‘)
VARCHAR(n)	Variabel bis n	Namen, E-Mails mit Längenbegrenzung
TEXT	Unbegrenzt	Beschreibungen, Kommentare, JSON

## Beispiel:

```

1 CREATE TABLE users (
2   country_code CHAR(2),          -- Immer 2 Zeichen: 'DE', 'US'
3   email VARCHAR(255),           -- Max. 255 Zeichen
4   bio TEXT                   -- Unbegrenzt
5 );

```

```
CREATE TABLE users (
    country_code CHAR(2),      -- Immer 2 Zeichen: 'DE', 'US'
    email VARCHAR(255),        -- Max. 255 Zeichen
    bio TEXT                  -- Unbegrenzt
)
```

ok

Datum- und Zeit-Typen sind essenziell für zeitbasierte Analysen. DATE speichert nur das Datum, TIMESTAMP speichert Datum + Uhrzeit, INTERVAL repräsentiert Zeitdauern.

## Datum & Zeit

Typ	Format	Beispiel	Wann nutzen?
DATE	YYYY-MM-DD	2025-11-04	Geburtstage, Events
TIME	HH:MM:SS	14:30:00	Öffnungszeiten
TIMESTAMP P	YYYY-MM-DD HH:MM:SS	2025-11-04 14:30:00	Logs, created_at
INTERVAL	Duration	'3 days', '2 hours'	Zeitrechnungen

## Beispiel: Zeitberechnungen

```
1 * CREATE TABLE events (
2     event_id INTEGER PRIMARY KEY,
3     event_name TEXT,
4     event_date DATE,
5     start_time TIMESTAMP
6 );
7
8 INSERT INTO events VALUES
9     (1, 'Konferenz', '2025-12-15', '2025-12-15 09:00:00');
10
11 -- 3 Tage vor dem Event:
12 SELECT
13     event_name,
14     event_date,
15     event_date - INTERVAL '3 days' AS reminder_date
16 FROM events;
```

```
CREATE TABLE events (
    event_id INTEGER PRIMARY KEY,
    event_name TEXT,
    event_date DATE,
    start_time TIMESTAMP
)
```

ok

```
INSERT INTO events VALUES
(1, 'Konferenz', '2025-12-15', '2025-12-15 09:00:00')
```

ok

-- 3 Tage vor dem Event:

```
SELECT
    event_name,
    event_date,
    event_date - INTERVAL '3 days' AS reminder_date
FROM events
```

#	event_name	event_date	reminder_date
1	Konferenz	2025-12-15	2025-12-12

1 rows

Boolean und Spezialtypen runden das Bild ab. BOOLEAN für Ja/Nein-Flags, JSON für strukturierte Daten, ARRAY für Listen.

## Boolean & Spezialtypen

Typ	Werte	Beispiel	Wann nutzen?
BOOLEAN	TRUE, FALSE, NULL	is_active	Flags, Status
JSON	JSON-Objekt	{"key": "value"}	Flexible Daten
ARRAY	Liste	[1, 2, 3]	Tags, Listen
UUID	Universally Unique ID	550e8400-e29b...	Verteilte IDs

## Beispiel: JSON-Spalte

```
1 CREATE TABLE products_ext (
2     product_id INTEGER PRIMARY KEY,
3     name TEXT
```



```

5     name TEXT,
4   metadata JSON -- Flexible Zusatzdaten
5 );
6
7 INSERT INTO products_ext VALUES
8   (1, 'Laptop', '{"brand": "Dell", "warranty_years": 3}');
9
10 -- JSON abfragen (PGLite.:
11 SELECT
12   name,
13   metadata->>'brand' AS brand,
14   metadata->>'warranty_years' AS warranty
15 FROM products_ext;

```

```

CREATE TABLE products_ext (
  product_id INTEGER PRIMARY KEY,
  name TEXT,
  metadata JSON -- Flexible Zusatzdaten
)

```

*ok*

```

INSERT INTO products_ext VALUES
(1, 'Laptop', '{"brand": "Dell", "warranty_years": 3}')

```

*ok*

```

-- JSON abfragen (PGLite.:
SELECT
  name,
  metadata->>'brand' AS brand,
  metadata->>'warranty_years' AS warranty
FROM products_ext

```

#	name	brand	warranty
1	Laptop	Dell	3

1 rows

## Tabellen ändern ([ALTER TABLE](#))

Schritt : Ausgangstabelle erstellen

```

1 -- Frische Tabelle für ALTER-Experimente:
2 CREATE TABLE products_alter (
3   product_id INTEGER PRIMARY KEY,
4   name TEXT NOT NULL

```

```

    .      name TEXT NOT NULL,
5     price DECIMAL(10, 2),
6     stock INTEGER
7 );
8
9 INSERT INTO products_alter VALUES
10   (1, 'Laptop', 999.50, 5),
11   (2, 'Mouse', 25.00, 100);
12
13 SELECT * FROM products_alter;

```

-- Frische Tabelle für ALTER-Experimente:

```

CREATE TABLE products_alter (
  product_id INTEGER PRIMARY KEY,
  name TEXT NOT NULL,
  price DECIMAL(10, 2),
  stock INTEGER
)

```

*ok*

**INSERT INTO products\_alter VALUES**

```

(1, 'Laptop', 999.50, 5),
(2, 'Mouse', 25.00, 100)

```

*ok*

**SELECT \* FROM products\_alter**

#	product_id	name	price	stock
1	1	Laptop	999.50	5
2	2	Mouse	25.00	100

2 rows

## Spalten hinzufügen (ADD COLUMN)

Schemas ändern sich. Sie fügen Spalten hinzu, ändern Datentypen, löschen veraltete Felder. ALTER TABLE ist Ihr Werkzeug für Schema-Evolution. Aber Vorsicht: Manche Operationen sind riskant bei großen Tabellen.

Syntax:

```

ALTER TABLE table_name
ADD COLUMN column_name datatype [constraints];

```

Beispiel:

```

1 -- Neue Spalte hinzufügen:
2 ALTER TABLE products_alter
3 ADD COLUMN category TEXT DEFAULT 'Uncategorized';
4
5 -- Prüfen:
6 SELECT * FROM products_alter;

```

-- Neue Spalte hinzufügen:  
**ALTER TABLE products\_alter  
ADD COLUMN category TEXT DEFAULT 'Uncategorized'**

*ok*

-- Prüfen:  
**SELECT \* FROM products\_alter**

#	product_id	name	price	stock	category
1	1	Laptop	999.50	5	Uncategorized
2	2	Mouse	25.00	100	Uncategorized

2 rows

 **Best Practice:** Neue Spalten mit DEFAULT oder NULL hinzufügen, um Lock-Probleme zu vermeiden.

### Spalten ändern (ALTER COLUMN)

Spalten ändern ist komplexer. Sie können Datentypen ändern, Defaults setzen, Constraints hinzufügen. Aber nicht alle Datenbanken unterstützen alle Operationen gleich. Lassen Sie uns das Schritt für Schritt durchspielen.

**Schritt 2: Datentyp ändern (price: DECIMAL(10,2) → DECIMAL(12,2))**

```

1 -- Präzision erhöhen für größere Preise:
2 ALTER TABLE products_alter
3 ALTER COLUMN price TYPE DECIMAL(12, 2);
4
5 -- Prüfen (Struktur hat sich geändert, Daten bleiben):
6 SELECT * FROM products_alter;
7
8 -- In MySQL würde man schreiben:
9 -- ALTER TABLE products_alter
10 -- MODIFY COLUMN price DECIMAL(12, 2);

```

```
-- Präzision erhöhen für größere Preise:  
ALTER TABLE products_alter  
ALTER COLUMN price TYPE DECIMAL(12, 2)
```

ok

```
-- Prüfen (Struktur hat sich geändert, Daten bleiben):  
SELECT * FROM products_alter
```

#	product_id	name	price	stock	category
1	1	Laptop	999.50	5	Uncategorized
2	2	Mouse	25.00	100	Uncategorized

2 rows

```
-- In MySQL würde man schreiben:  
-- ALTER TABLE products_alter  
-- MODIFY COLUMN price DECIMAL(12, 2);
```

ok

### Schritt 3: Default setzen für stock

```
1 -- Neue Produkte sollen standardmäßig 10 auf Lager haben:  
2 ALTER TABLE products_alter  
3 ALTER COLUMN stock SET DEFAULT 10;  
4  
5 -- Test: Neues Produkt ohne stock-Angabe:  
6 INSERT INTO products_alter (product_id, name, price)  
7 VALUES (3, 'Keyboard', 79.99);  
8  
9 SELECT * FROM products_alter;  
10 -- → Keyboard hat automatisch stock = 10
```

```
-- Neue Produkte sollen standardmäßig 10 auf Lager haben:  
ALTER TABLE products_alter  
ALTER COLUMN stock SET DEFAULT 10
```

ok

```
-- Test: Neues Produkt ohne stock-Angabe:  
INSERT INTO products_alter (product_id, name, price)  
VALUES (3, 'Keyboard', 79.99)
```

ok

```
SELECT * FROM products_alter
```

#	product_id	name	price	stock	category
1	1	Laptop	999.50	5	Uncategorized
2	2	Mouse	25.00	100	Uncategorized
3	3	Keyboard	79.99	10	Uncategorized

3 rows

```
-- → Keyboard hat automatisch stock = 10
```

ok

#### Schritt 4: NOT NULL hinzufügen

```
1 -- Alle Produkte müssen einen Preis haben:  
2 -- (Funktioniert nur, wenn keine NULL-Werte existieren!)  
3 ALTER TABLE products_alter  
4 ALTER COLUMN price SET NOT NULL;  
5  
6 -- Test: INSERT ohne price wird jetzt fehlschlagen:  
7 -- INSERT INTO products_alter (product_id, name)  
8 -- VALUES (4, 'Monitor'); -- ❌ Fehler!  
9  
10 SELECT * FROM products_alter;
```

```
-- Alle Produkte müssen einen Preis haben:  
-- (Funktioniert nur, wenn keine NULL-Werte existieren!)  
ALTER TABLE products_alter  
ALTER COLUMN price SET NOT NULL
```

ok

```
-- Test: INSERT ohne price wird jetzt fehlgeschlagen:  
-- INSERT INTO products_alter (product_id, name)  
-- VALUES (4, 'Monitor'); -- ❌ Fehler!
```

```
SELECT * FROM products_alter
```

#	product_id	name	price	stock	category
1	1	Laptop	999.50	5	Uncategorized
2	2	Mouse	25.00	100	Uncategorized
3	3	Keyboard	79.99	10	Uncategorized

3 rows

### ⚠ Achtung bei Datentyp-Änderungen:

- **TEXT** → **INTEGER**: Funktioniert nur, wenn alle Werte Zahlen sind
- **INTEGER** → **BIGINT**: Meist sicher
- **DECIMAL** Präzision erhöhen: Sicher
- **DECIMAL** Präzision verringern: Kann Daten abschneiden!
- Bei großen Tabellen: Kann lange dauern + Locks!

### Spalten löschen (DROP COLUMN)

Spalten löschen ist riskant. Sobald weg, sind die Daten weg. Überlegen Sie zweimal, bevor Sie DROP COLUMN nutzen. Manchmal ist es besser, eine Spalte zu „verstecken“ (in Views) statt zu löschen.

Syntax:

```
ALTER TABLE products  
DROP COLUMN description;
```



### ⚠ Vorsicht:

- Daten werden **permanent** gelöscht
- Kann nicht rückgängig gemacht werden (außer via Backup)
- Bei FOREIGN KEY Constraints: Kann fehlgeschlagen

## Alternative: Soft Delete

Statt Spalte zu löschen:

```
1 -- Spalte umbenennen (verstecken):  
2 ALTER TABLE products_alter  
3 RENAME COLUMN description TO _deprecated_description;  
4  
5 -- Oder in Views weglassen:  
6 CREATE VIEW products_view AS  
7 SELECT product_id, name, price FROM products;
```

```
-- Spalte umbenennen (verstecken):  
ALTER TABLE products_alter  
RENAME COLUMN description TO _deprecated_description
```

column "description" does not exist

## Tabelle umbenennen (`RENAME TO`)

Tabellen können umbenannt werden. Das ist nützlich, wenn Sie Schema-Migrationen machen oder alte Versionen als Backup behalten wollen. Schauen wir uns das mit unserer products\_alter Tabelle an.

Syntax:

```
ALTER TABLE old_name RENAME TO new_name;
```

### Schritt 1: Aktuellen Zustand prüfen

```
1 -- Was haben wir bisher in products_alter?  
2 SELECT * FROM products_alter;
```

```
-- Was haben wir bisher in products_alter?  
SELECT * FROM products_alter
```

#	product_id	name	price	stock	category
1	1	Laptop	999.50	5	Uncategorized
2	2	Mouse	25.00	100	Uncategorized
3	3	Keyboard	79.99	10	Uncategorized

3 rows

### Schritt 2: Backup erstellen (Tabelle kopieren)

```
1 -- Kopie als Backup:
```

```
2 CREATE TABLE products_alter_backup AS  
3 SELECT * FROM products_alter;
```

```
-- Kopie als Backup:  
CREATE TABLE products_alter_backup AS  
SELECT * FROM products_alter
```

ok

### Schritt 3: Original umbenennen (z.B. für Migration)

```
1 -- Original wird zu "old version":  
2 ALTER TABLE products_alter RENAME TO products_alter_v1;  
3  
4 -- Jetzt existiert nur noch products_alter_v1:  
5 SELECT * FROM products_alter_v1;
```

```
-- Original wird zu "old version":  
ALTER TABLE products_alter RENAME TO products_alter_v1
```

ok

```
-- Jetzt existiert nur noch products_alter_v1:  
SELECT * FROM products_alter_v1
```

#	product_id	name	price	stock	category
1	1	Laptop	999.50	5	Uncategorized
2	2	Mouse	25.00	100	Uncategorized
3	3	Keyboard	79.99	10	Uncategorized

3 rows

### Schritt 4: Neue Version als „products\_alter“ anlegen

```
1 -- Neue, verbesserte Version (z.B. mit zusätzlicher Spalte):  
2 CREATE TABLE products_alter AS  
3 SELECT  
4   product_id,  
5   name,  
6   price,  
7   stock,  
8   category,  
9   'active' AS status -- Neue Spalte!  
10  FROM products_alter_v1;
```

```

10  FROM products.Alter_v1;
11
12 -- Neue Version prüfen:
13 SELECT * FROM products.Alter;

```

-- Neue, verbesserte Version (z.B. mit zusätzlicher Spalte):

**CREATE TABLE products.Alter AS**

**SELECT**

```

product_id,
name,
price,
stock,
category,
'active' AS status -- Neue Spalte!
FROM products.Alter_v1

```

ok

-- Neue Version prüfen:

**SELECT \* FROM products.Alter**

#	product_id	name	price	stock	category	status
1	1	Laptop	999.50	5	Uncategorized	active
2	2	Mouse	25.00	100	Uncategorized	active
3	3	Keyboard	79.99	10	Uncategorized	active

3 rows

### 💡 Use Cases für RENAME:

- **Schema-Migration:** Wenn du die Struktur einer Tabelle ändern willst (z.B. neue Spalten, andere Datentypen), kannst du die alte Tabelle umbenennen (z.B. `products.Alter` → `products.Alter_v1`). Dann legst du eine neue Tabelle mit dem alten Namen und neuer Struktur an und überträgst die Daten. So bleibt die alte Version als Backup erhalten, bis alles funktioniert.
- **Zero-Downtime-Deployment:** Bei Systemen, die immer verfügbar sein müssen, kannst du eine neue Tabelle im Hintergrund aufbauen und befüllen. Sobald sie fertig ist, benennst du sie mit RENAME atomar um – so ist der Wechsel für Nutzer:innen sofort und ohne Ausfall sichtbar.
- **Backup vor großen Änderungen:** Vor riskanten Änderungen (z.B. Spalte löschen) kannst du die Original-Tabelle einfach kopieren und umbenennen (z.B. `products` → `products.Backup`). So hast du eine Sicherung, falls etwas schiefgeht, und kannst Daten leicht wiederherstellen.

## Tabellen löschen (`DROP TABLE`)

## Grundsyntax

`DROP TABLE` ist der gefährlichste DDL-Befehl. Einmal ausgeführt, ist die Tabelle weg – inklusive aller Daten. Nutzen Sie `IF EXISTS`, um Fehler zu vermeiden, und `CASCADE/RESTRICT`, um Abhängigkeiten zu kontrollieren.

### Einfaches DROP:

```
DROP TABLE products;
```



### Mit Sicherheitsnetz:

```
DROP TABLE IF EXISTS products;
```



#### ⚠ Gefahr:

- Tabelle wird **sofort** gelöscht
- Alle Daten gehen verloren
- Kann nicht rückgängig gemacht werden (außer Backup/Transaktion)

CASCADE und RESTRICT steuern, was mit abhängigen Objekten passiert. CASCADE löscht alles mit (Views, Foreign Keys), RESTRICT verhindert das Löschen, wenn Abhängigkeiten existieren.

## CASCADE vs. RESTRICT

### RESTRICT (Standard):

```
-- Fehlschlägt, wenn andere Tabellen via FOREIGN KEY abhängen:  
DROP TABLE products RESTRICT;
```



### CASCADE (Vorsicht!):

```
-- Löscht Tabelle UND alle abhängigen Objekte (Views, FKs):  
DROP TABLE products CASCADE;
```



### Beispiel:

```
1 * CREATE TABLE categories (  
2     category_id INTEGER PRIMARY KEY,  
3     name TEXT  
4 );  
5  
6 * CREATE TABLE products_fk (  
7     product_id INTEGER PRIMARY KEY,  
8     name TEXT,  
9     category_id INTEGER,  
10    FOREIGN KEY (category_id) REFERENCES categories(category_id)  
11 );
```

```
12
13 -- Fehlschlägt (products_fk hängt davon ab):
14 DROP TABLE categories RESTRICT;
15
16 -- Funktioniert (löscht auch FOREIGN KEY Constraint):
17 DROP TABLE categories CASCADE;
```

```
CREATE TABLE categories (
    category_id INTEGER PRIMARY KEY,
    name TEXT
)
```

ok

```
CREATE TABLE products_fk (
    product_id INTEGER PRIMARY KEY,
    name TEXT,
    category_id INTEGER,
    FOREIGN KEY (category_id) REFERENCES categories(category_id)
)
```

ok

```
-- Fehlschlägt (products_fk hängt davon ab):
DROP TABLE categories RESTRICT
```

cannot drop table categories because other objects depend on it

 Best Practice: Immer RESTRICT nutzen, außer Sie wissen genau, was Sie tun.

## Constraints – Datenintegrität sichern

### Warum Constraints?

Constraints sind Regeln, die Ihre Daten schützen. PRIMARY KEY verhindert Duplikate, FOREIGN KEY sichert Beziehungen, CHECK validiert Werte. Ohne Constraints ist Ihre Datenbank ein Wilder Westen – jeder Wert ist erlaubt.

### Ohne Constraints:

```
1 CREATE TABLE orders_bad (
2     order_id INTEGER,
3     customer_id INTEGER,
4     total DECIMAL(10,2)
5 );
6
7 -- Alles erlaubt.
```



```

    '      Alles erlaubt.
8   -- X Kein Kunde, negativer Betrag
9   INSERT INTO orders_bad VALUES (1, NULL, -100);
10  -- X Duplikat-ID, nicht-existierender Kunde
11  INSERT INTO orders_bad VALUES (1, 999, 50);

```

```

CREATE TABLE orders_bad (
    order_id INTEGER,
    customer_id INTEGER,
    total DECIMAL(10,2)
)

```

ok

```

-- Alles erlaubt:
-- X Kein Kunde, negativer Betrag
INSERT INTO orders_bad VALUES (1, NULL, -100)

```

ok

```

-- X Duplikat-ID, nicht-existierender Kunde
INSERT INTO orders_bad VALUES (1, 999, 50)

```

ok

Mit Constraints:

```

1 * CREATE TABLE customers (
2     customer_id INTEGER PRIMARY KEY,
3     name TEXT NOT NULL
4 );
5
6 * CREATE TABLE orders_good (
7     order_id INTEGER PRIMARY KEY,
8     customer_id INTEGER NOT NULL,
9     total DECIMAL(10,2) CHECK (total >= 0),
10    FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
11 );
12
13  -- Schutz aktiviert:
14  -- X customer_id NOT NULL
15  INSERT INTO orders_good VALUES (1, NULL, 50);
16  -- X customer_id existiert nicht
17  INSERT INTO orders_good VALUES (1, 999, 50);
18  -- X total CHECK fehlschlägt
19  INSERT INTO orders_good VALUES (1, 1, -100);

```

```
CREATE TABLE customers (
    customer_id INTEGER PRIMARY KEY,
    name TEXT NOT NULL
)
```

ok

```
CREATE TABLE orders_good (
    order_id INTEGER PRIMARY KEY,
    customer_id INTEGER NOT NULL,
    total DECIMAL(10,2) CHECK (total >= 0),
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
)
```

ok

```
-- Schutz aktiviert:
-- ✗ customer_id NOT NULL
INSERT INTO orders_good VALUES (1, NULL, 50)
```

null value in column "customer\_id" of relation "orders\_good" violates not-null constraint

## PRIMARY KEY

### Single-Column vs. Composite Keys

PRIMARY KEY ist der wichtigste Constraint. Er garantiert Eindeutigkeit und NOT NULL. Jede Tabelle sollte einen Primärschlüssel haben – er ist die Identität jeder Zeile.

Single-Column (häufigster Fall):

```
CREATE TABLE users (
    user_id INTEGER PRIMARY KEY,
    username TEXT UNIQUE,
    email TEXT
);
```

Composite Key (mehrere Spalten):

```
1 CREATE TABLE enrollments (
2     student_id INTEGER,
3     course_id INTEGER,
4     enrollment_date DATE,
5     PRIMARY KEY (student_id, course_id) -- Ein Student kann jeden Kurs
6         einmal belegen
);
```

```
CREATE TABLE enrollments (
    student_id INTEGER,
    course_id INTEGER,
    enrollment_date DATE,
    PRIMARY KEY (student_id, course_id) -- Ein Student kann jeden Kurs nur einmal
        belegen
)
```

ok

## Wann Composite Keys?

- Viele-zu-Viele-Beziehungen (Student ↔ Kurs)
- Zeitreihendaten (sensor\_id, timestamp)

## Natürliche vs. künstliche Keys

Natürliche vs. künstliche Keys: Natürlich = aus Daten (E-Mail, ISBN), künstlich = generiert (Auto-Increment ID). Künstliche Keys sind meist besser, weil sie unveränderlich sind.

### Natürlicher Key (aus Daten):

```
CREATE TABLE books (
    isbn TEXT PRIMARY KEY, -- ISBN ist natürlich eindeutig
    title TEXT,
    author TEXT
);
```

### Künstlicher Key (generiert):

```
CREATE TABLE books_auto (
    book_id INTEGER PRIMARY KEY, -- Auto-generiert
    isbn TEXT UNIQUE,
    title TEXT,
    author TEXT
);
```

## Wann was?

Kriterium	Natürlich	Künstlich
Unveränderlich	✗ (z.B. E-Mail ändert sich)	✓
Performance	⚠️ (Text-Keys langsamer)	✓ (Integer schnell)
Lesbarkeit	✓ (ISBN sagt etwas aus)	✗ (ID 4711 ist abstrakt)

💡 Empfehlung: Künstlicher Primärschlüssel + natürlicher UNIQUE Constraint

```
CREATE TABLE users_best (
    user_id INTEGER PRIMARY KEY,
    email TEXT UNIQUE NOT NULL,
    username TEXT
);
```



## FOREIGN KEY

### Referenzielle Integrität

FOREIGN KEY verbindet Tabellen. Er garantiert, dass Beziehungen gültig sind: Jede Bestellung muss einem existierenden Kunden gehören. Das ist referenzielle Integrität.

Beispiel: Kunden und Bestellungen

```
1 * CREATE TABLE customers_fk (
2     customer_id INTEGER PRIMARY KEY,
3     name TEXT NOT NULL
4 );
5
6 * CREATE TABLE orders_fk (
7     order_id INTEGER PRIMARY KEY,
8     customer_id INTEGER NOT NULL,
9     order_date DATE,
10    FOREIGN KEY (customer_id) REFERENCES customers_fk(customer_id)
11 );
12
13 INSERT INTO customers_fk VALUES (1, 'Alice'), (2, 'Bob');
14
15 -- Funktioniert (customer_id 1 existiert):
16 INSERT INTO orders_fk VALUES (101, 1, '2025-11-04');
17
18 -- Fehlschlägt (customer_id 999 existiert nicht):
19 INSERT INTO orders_fk VALUES (102, 999, '2025-11-04');
```



```
CREATE TABLE customers_fk (
    customer_id INTEGER PRIMARY KEY,
    name TEXT NOT NULL
)
```

ok

```
CREATE TABLE orders_fk (
    order_id INTEGER PRIMARY KEY,
    customer_id INTEGER NOT NULL,
    order_date DATE,
    FOREIGN KEY (customer_id) REFERENCES customers_fk(customer_id)
)
```

ok

```
INSERT INTO customers_fk VALUES (1, 'Alice'), (2, 'Bob')
```

ok

```
-- Funktioniert (customer_id 1 existiert):
INSERT INTO orders_fk VALUES (101, 1, '2025-11-04')
```

ok

```
-- Fehlschlägt (customer_id 999 existiert nicht):
INSERT INTO orders_fk VALUES (102, 999, '2025-11-04')
```

insert or update on table "orders\_fk" violates foreign key constraint  
"orders\_fk\_customer\_id\_fkey"

Was passiert bei Verstößen?

- **INSERT**: Fehlschlag, wenn referenzierter Key nicht existiert
- **UPDATE**: Fehlschlag, wenn neuer Wert nicht existiert
- **DELETE**: Abhängig von ON DELETE (siehe unten)

#### ON DELETE / ON UPDATE

ON DELETE und ON UPDATE steuern, was passiert, wenn der referenzierte Datensatz gelöscht oder geändert wird. CASCADE löscht/ändert mit, SET NULL setzt NULL, RESTRICT verhindert die Aktion.

Option	Bei DELETE	Bei UPDATE
CASCADE	Abhängige Zeilen werden auch gelöscht	Abhängige Zeilen werden aktualisiert
SET NULL	FK wird auf NULL gesetzt	FK wird auf NULL gesetzt
RESTRICT	Löschen/Ändern wird verhindert	Löschen/Ändern wird verhindert
NO ACTION	Wie RESTRICT (Standard)	Wie RESTRICT (Standard)

### Beispiel: ON DELETE CASCADE

```

1 * CREATE TABLE authors (
2     author_id INTEGER PRIMARY KEY,
3     name TEXT
4 );
5
6 * CREATE TABLE books_cascade (
7     book_id INTEGER PRIMARY KEY,
8     title TEXT,
9     author_id INTEGER,
10    FOREIGN KEY (author_id) REFERENCES authors(author_id) ON DELETE CAS
11 );
12
13 INSERT INTO authors VALUES (1, 'Tolkien');
14 INSERT INTO books_cascade VALUES (1, 'Hobbit', 1), (2, 'LOTR', 1);
15
16 -- Autor löschen → Bücher werden auch gelöscht:
17 DELETE FROM authors WHERE author_id = 1;
18
19 SELECT * FROM books_cascade; -- Leer!

```

```
CREATE TABLE authors (
    author_id INTEGER PRIMARY KEY,
    name TEXT
)
```

ok

```
CREATE TABLE books_cascade (
    book_id INTEGER PRIMARY KEY,
    title TEXT,
    author_id INTEGER,
    FOREIGN KEY (author_id) REFERENCES authors(author_id) ON DELETE CASCADE
)
```

ok

```
INSERT INTO authors VALUES (1, 'Tolkien')
```

ok

```
INSERT INTO books_cascade VALUES (1, 'Hobbit', 1), (2, 'LOTR', 1)
```

ok

```
-- Autor löschen → Bücher werden auch gelöscht:  
DELETE FROM authors WHERE author_id = 1
```

ok

```
SELECT * FROM books_cascade
```

#	book_id	title	author_id
0 rows			

```
-- Leer!
```

ok

Beispiel: ON DELETE SET NULL

```
CREATE TABLE books_setnull (
    book_id INTEGER PRIMARY KEY,
    title TEXT,
    author_id INTEGER,
    FOREIGN KEY (author_id) REFERENCES authors(author_id) ON DELETE SET NULL
```

```
);

-- Autor löschen → author_id wird NULL:
DELETE FROM authors WHERE author_id = 1;
-- Bücher bleiben, aber ohne Autor
```

### 💡 Wann was nutzen?

- **CASCADE:** Abhängige Daten sind ohne Parent sinnlos (z.B. Bestellpositionen ohne Bestellung)
- **SET NULL:** Beziehung optional (z.B. Autor gelöscht, Buch bleibt)
- **RESTRICT:** Keine Löschung, solange Abhängigkeiten bestehen (Standard, sicher)

## Self-Referencing (Hierarchien)

Self-Referencing Foreign Keys sind nützlich für hierarchische Daten: Jeder Mitarbeiter hat einen Manager, der selbst ein Mitarbeiter ist. Jede Kategorie kann eine übergeordnete Kategorie haben.

### Beispiel: Mitarbeiter-Hierarchie

```
1 CREATE TABLE employees (
2     employee_id INTEGER PRIMARY KEY,
3     name TEXT,
4     manager_id INTEGER,
5     FOREIGN KEY (manager_id) REFERENCES employees(employee_id)
6 );
7
8 INSERT INTO employees VALUES
9     (1, 'CEO', NULL),           -- Kein Manager (Top)
10    (2, 'CTO', 1),             -- Manager: CEO
11    (3, 'Dev Lead', 2),       -- Manager: CTO
12    (4, 'Developer', 3);      -- Manager: Dev Lead
13
14 -- Wer ist der Manager von Developer?
15 SELECT
16     e.name AS employee,
17     m.name AS manager
18 FROM employees e
19 LEFT JOIN employees m ON e.manager_id = m.employee_id
20 WHERE e.name = 'Developer';
```

```
CREATE TABLE employees (
    employee_id INTEGER PRIMARY KEY,
    name TEXT,
    manager_id INTEGER,
    FOREIGN KEY (manager_id) REFERENCES employees(employee_id)
)
```

ok

```
INSERT INTO employees VALUES
    (1, 'CEO', NULL),          -- Kein Manager (Top)
    (2, 'CTO', 1),            -- Manager: CEO
    (3, 'Dev Lead', 2),       -- Manager: CTO
    (4, 'Developer', 3)
```

ok

-- Manager: Dev Lead

-- Wer ist der Manager von Developer?

```
SELECT
    e.name AS employee,
    m.name AS manager
FROM employees e
LEFT JOIN employees m ON e.manager_id = m.employee_id
WHERE e.name = 'Developer'
```

#	employee	manager
1	Developer	Dev Lead

1 rows

## Use Cases:

- Organisationshierarchien
- Kategorie-Bäume (Produkte → Elektronik → Laptops)
- Threads/Kommentare (parentcommentid)

**UNIQUE**, **NOT NULL**, **CHECK**, **DEFAULT**

## UNIQUE – Eindeutigkeit ohne Primary Key

Diese Constraints sind einfacher, aber nicht weniger wichtig. UNIQUE verhindert Duplikate, NOT NULL erzwingt Werte, CHECK validiert Bedingungen, DEFAULT setzt Standardwerte.

Syntax:

```
1 CREATE TABLE users_unique (  
2     user_id INTEGER PRIMARY KEY,  
3     email TEXT UNIQUE NOT NULL,  
4     username TEXT UNIQUE  
5 );
```



```
CREATE TABLE users_unique (  
    user_id INTEGER PRIMARY KEY,  
    email TEXT UNIQUE NOT NULL,  
    username TEXT UNIQUE  
)
```

ok

Unterschied zu PRIMARY KEY:

- PRIMARY KEY: Eindeutig + NOT NULL + nur 1 pro Tabelle
- UNIQUE: Eindeutig, aber NULL erlaubt (mehrere!), mehrere pro Tabelle

NULL-Verhalten:

```
1 INSERT INTO users_unique VALUES (1, 'alice@example.com', 'alice');  
2 INSERT INTO users_unique VALUES (2, 'bob@example.com', NULL); -- OK  
3 INSERT INTO users_unique VALUES (3, 'charlie@example.com', NULL); -- 0  
    (NULL != NULL)
```

```
INSERT INTO users_unique VALUES (1, 'alice@example.com', 'alice')
```

ok

```
INSERT INTO users_unique VALUES (2, 'bob@example.com', NULL)
```

ok

```
-- OK  
INSERT INTO users_unique VALUES (3, 'charlie@example.com', NULL)
```

ok

```
-- OK (NULL != NULL)
```

ok

## Composite UNIQUE:

```
1 CREATE TABLE reservations (
2   reservation_id INTEGER PRIMARY KEY,
3   room_number INTEGER,
4   date DATE,
5   UNIQUE (room_number, date) -- Raum kann pro Tag nur 1x gebucht werden
6 );
```

```
CREATE TABLE reservations (
  reservation_id INTEGER PRIMARY KEY,
  room_number INTEGER,
  date DATE,
  UNIQUE (room_number, date) -- Raum kann pro Tag nur 1x gebucht werden
)
```

ok

NOT NULL ist der einfachste Constraint, aber extrem wichtig. Er verhindert NULL-Werte in Spalten, die immer einen Wert haben müssen.

## NOT NULL – Pflichtfelder

Syntax:

```
1 CREATE TABLE products_nn (
2   product_id INTEGER PRIMARY KEY,
3   name TEXT NOT NULL,
4   price DECIMAL(10,2) NOT NULL,
5   description TEXT -- NULL erlaubt
6 );
```

```
CREATE TABLE products_nn (
  product_id INTEGER PRIMARY KEY,
  name TEXT NOT NULL,
  price DECIMAL(10,2) NOT NULL,
  description TEXT -- NULL erlaubt
)
```

ok

Warum wichtig?

- NULL ist nicht 0, nicht leerer String – es ist „unbekannt“
- Berechnungen mit NULL geben NULL zurück
- WHERE-Bedingungen können scheitern

Beispiel:

```
1 -- Fehlschlägt (name ist NOT NULL):
2 INSERT INTO products_nn (product_id, price) VALUES (1, 99.99);
3
4 -- Funktioniert:
5 INSERT INTO products_nn (product_id, name, price) VALUES (1, 'Laptop',
.99);
```

-- Fehlschlägt (name ist NOT NULL):  
INSERT INTO products\_nn (product\_id, price) VALUES (1, 99.99)

null value in column "name" of relation "products\_nn" violates not-null constraint

CHECK ermöglicht benutzerdefinierte Validierung. Sie können Bereiche prüfen, Muster validieren, Bedingungen zwischen Spalten definieren.

## CHECK – Benutzerdefinierte Validierung

Syntax:

```
CHECK (condition)
```

Beispiele:

```
1 CREATE TABLE products_check (
2     product_id INTEGER PRIMARY KEY,
3     name TEXT NOT NULL,
4     price DECIMAL(10,2) CHECK (price >= 0),
5     discount_percent INTEGER CHECK (discount_percent BETWEEN 0 AND 100),
6     stock INTEGER CHECK (stock >= 0),
7     release_date DATE CHECK (release_date >= CURRENT_DATE)
8 );
```

```
CREATE TABLE products_check (
    product_id INTEGER PRIMARY KEY,
    name TEXT NOT NULL,
    price DECIMAL(10,2) CHECK (price >= 0),
    discount_percent INTEGER CHECK (discount_percent BETWEEN 0 AND 100),
    stock INTEGER CHECK (stock >= 0),
    release_date DATE CHECK (release_date >= CURRENT_DATE)
)
```

ok

Multi-Column Checks:

```
1 CREATE TABLE discounts (
2     discount_id INTEGER PRIMARY KEY,
3     start_date DATE,
4     end_date DATE,
5     CHECK (end_date > start_date) -- Ende muss nach Start sein
6 );
```

```
CREATE TABLE discounts (
    discount_id INTEGER PRIMARY KEY,
    start_date DATE,
    end_date DATE,
    CHECK (end_date > start_date) -- Ende muss nach Start sein
)
```

ok

Enum-Simulation:

```
1 CREATE TABLE orders_status (
2     order_id INTEGER PRIMARY KEY,
3     status TEXT CHECK (status IN ('pending', 'shipped', 'delivered',
4         'cancelled'))
5 );
6 -- Fehlschlägt (ungültiger Status):
7 INSERT INTO orders_status VALUES (1, 'in_transit');
```

```
CREATE TABLE orders_status (
    order_id INTEGER PRIMARY KEY,
    status TEXT CHECK (status IN ('pending', 'shipped', 'delivered', 'cancelled'))
)
```

ok

```
-- Fehlschlägt (ungültiger Status):
INSERT INTO orders_status VALUES (1, 'in_transit')
```

new row for relation "orders\_status" violates check constraint "orders\_status\_status\_check"

## DEFAULT – Standardwerte

DEFAULT setzt Standardwerte, wenn beim INSERT kein Wert angegeben wird. Praktisch für Zeitstempel, Flags, Status.

Syntax:

```
CREATE TABLE ... DEFAULT ...
```

```
1 CREATE TABLE products_default (
2     product_id INTEGER PRIMARY KEY,
3     name TEXT NOT NULL,
4     price DECIMAL(10,2) NOT NULL,
5     stock INTEGER DEFAULT 0,
6     is_active BOOLEAN DEFAULT TRUE,
7     created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
8 );
```

```
CREATE TABLE products_default (
    product_id INTEGER PRIMARY KEY,
    name TEXT NOT NULL,
    price DECIMAL(10,2) NOT NULL,
    stock INTEGER DEFAULT 0,
    is_active BOOLEAN DEFAULT TRUE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
)
```

ok

Nutzung:

```
1 -- Ohne stock, is_active, created_at:
2 INSERT INTO products_default (product_id, name, price)
3 VALUES (1, 'Laptop', 999.99);
4
5 SELECT * FROM products_default;
6 -- → stock = 0, is_active = TRUE, created_at = jetzt
```

```
-- Ohne stock, is_active, created_at:
INSERT INTO products_default (product_id, name, price)
VALUES (1, 'Laptop', 999.99)
```

ok

```
SELECT * FROM products_default
```

#	product_id	name	price	stock	is_active	created_at
1	1	Laptop	999.99	0	true	2026-02-13T10:23:10.964Z

1 rows

```
-- → stock = 0, is_active = TRUE, created_at = jetzt
```

ok

Funktionen als Default:

```
1 CREATE TABLE logs (
2   log_id INTEGER PRIMARY KEY,
3   message TEXT,
4   timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
5   random_id TEXT DEFAULT (gen_random_uuid()::TEXT)
6 );
```

```
CREATE TABLE logs (
  log_id INTEGER PRIMARY KEY,
  message TEXT,
  timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  random_id TEXT DEFAULT (gen_random_uuid()::TEXT)
)
```

ok

## DML: Data Manipulation Language

### INSERT

#### Einzelne Zeile einfügen

Jetzt verlassen wir DDL und gehen zu DML: Daten manipulieren. INSERT fügt neue Zeilen ein. Sie können einzelne Zeilen einfügen, mehrere gleichzeitig, oder Daten aus anderen Tabellen kopieren.

Syntax:

```
INSERT INTO table_name (column1, column2, ...)
VALUES (value1, value2, ...);
```

Beispiel:

```
1 CREATE TABLE customers_insert (
2   customer_id INTEGER PRIMARY KEY,
3   name TEXT NOT NULL,
4   email TEXT,
5   registered_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
6 );
7
8 INSERT INTO customers_insert (customer_id, name, email)
9 VALUES (1, 'Alice', 'alice@example.com');
10
11 SELECT * FROM customers_insert;
```

```
CREATE TABLE customers_insert (
    customer_id INTEGER PRIMARY KEY,
    name TEXT NOT NULL,
    email TEXT,
    registered_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
)
```

ok

```
INSERT INTO customers_insert (customer_id, name, email)
VALUES (1, 'Alice', 'alice@example.com')
```

ok

```
SELECT * FROM customers_insert
```

#	customer_id	name	email	registered_at
1	1	Alice	alice@example.com	2026-02-13T10:23:03.834Z

1 rows

Alle Spalten (Reihenfolge wie in CREATE TABLE):

```
INSERT INTO customers_insert
VALUES (2, 'Bob', 'bob@example.com', CURRENT_TIMESTAMP);
```

### Mehrere Zeilen gleichzeitig (Bulk Insert)

Bulk Insert ist effizienter als viele einzelne INSERTs. Statt 100 Befehle schreiben Sie einen mit 100 Wertepaaren.

Syntax:

```
INSERT INTO table_name (columns)
VALUES
(values1),
(values2),
(values3),
...;
```

Beispiel:

```
1  INSERT INTO customers_insert (customer_id, name, email) VALUES
2      (3, 'Charlie', 'charlie@example.com'),
3      (4, 'Diana', 'diana@example.com'),
4      (5, 'Eve', 'eve@example.com');
```

```
6 SELECT * FROM customers_insert;
```

```
INSERT INTO customers_insert (customer_id, name, email) VALUES
(3, 'Charlie', 'charlie@example.com'),
(4, 'Diana', 'diana@example.com'),
(5, 'Eve', 'eve@example.com')
```

ok

```
SELECT * FROM customers_insert
```

#	customer_id	name	email	registered_at
1	1	Alice	alice@example.com	2026-02-13T10:23:03.834Z
2	3	Charlie	charlie@example.com	2026-02-13T10:23:04.777Z
3	4	Diana	diana@example.com	2026-02-13T10:23:04.777Z
4	5	Eve	eve@example.com	2026-02-13T10:23:04.777Z

4 rows

Performance-Vorteil:

- 1 INSERT mit 1000 Zeilen: ~10ms
- 1000 einzelne INSERTs: ~1000ms

### INSERT ... SELECT

INSERT ... SELECT kopiert Daten aus einer anderen Tabelle. Praktisch für Backups, Datenmigrationen, berechnete Tabellen.

Syntax:

```
INSERT INTO target_table (columns)
SELECT columns FROM source_table WHERE condition;
```

Beispiel: Backup erstellen

```
1 CREATE TABLE customers_backup (
2     customer_id INTEGER,
3     name TEXT,
4     email TEXT,
5     backup_date DATE DEFAULT CURRENT_DATE
6 );
7
8 INSERT INTO customers_backup (customer_id, name, email)
9 SELECT customer_id, name, email FROM customers_insert;
10
```

```
11 SELECT * FROM customers_backup;
```

```
CREATE TABLE customers_backup (
    customer_id INTEGER,
    name TEXT,
    email TEXT,
    backup_date DATE DEFAULT CURRENT_DATE
)
```

ok

```
INSERT INTO customers_backup (customer_id, name, email)
SELECT customer_id, name, email FROM customers_insert
```

ok

```
SELECT * FROM customers_backup
```

#	customer_id	name	email	backup_date
1	3	Charlie	charlie@example.com	2026-02-13
2	4	Diana	diana@example.com	2026-02-13
3	5	Eve	eve@example.com	2026-02-13
4	1	Alice	alice_updated@example.com	2026-02-13

4 rows

### Beispiel: Gefilterte Kopie

```
-- Nur Kunden mit E-Mail:
INSERT INTO customers_backup (customer_id, name, email)
SELECT customer_id, name, email
FROM customers_insert
WHERE email IS NOT NULL;
```

### INSERT ... ON CONFLICT (Upsert)

Upsert (INSERT ... ON CONFLICT) ist ein fortgeschrittenes Pattern: „Füge ein, oder update, wenn schon vorhanden.“ Praktisch für Daten-Synchronisation.

**Problem:** Was, wenn die ID schon existiert?

```
1 -- Fehlschlägt (customer_id 1 existiert schon):
2 INSERT INTO customers_insert (customer_id, name, email)
3 VALUES (1, 'Alice Updated', 'alice_new@example.com');
```

```
-- Fehlschlägt (customer_id 1 existiert schon):
INSERT INTO customers_insert (customer_id, name, email)
VALUES (1, 'Alice Updated', 'alice_new@example.com')
```

```
duplicate key value violates unique constraint "customers_insert_pkey"
```

Lösung: ON CONFLICT DO UPDATE

```
1 INSERT INTO customers_insert (customer_id, name, email)
2 VALUES (1, 'Alice Updated', 'alice_new@example.com')
3 ON CONFLICT (customer_id) DO UPDATE
4 SET
5     name = EXCLUDED.name,
6     email = EXCLUDED.email;
7
8 SELECT * FROM customers_insert WHERE customer_id = 1;
```

```
INSERT INTO customers_insert (customer_id, name, email)
VALUES (1, 'Alice Updated', 'alice_new@example.com')
ON CONFLICT (customer_id) DO UPDATE
SET
    name = EXCLUDED.name,
    email = EXCLUDED.email
```

ok

```
SELECT * FROM customers_insert WHERE customer_id = 1
```

#	customer_id	name	email	registered_at
1	1	Alice Updated	alice_new@example.com	2026-02-13T10:23:03.834Z

1 rows

ON CONFLICT DO NOTHING:

```
1 -- Ignoriere Duplikate:
2 INSERT INTO customers_insert (customer_id, name, email)
3 VALUES (1, 'Alice', 'alice@example.com')
4 ON CONFLICT (customer_id) DO NOTHING;
```

```
-- Ignriere Duplike:  
INSERT INTO customers_insert (customer_id, name, email)  
VALUES (1, 'Alice', 'alice@example.com')  
ON CONFLICT (customer_id) DO NOTHING
```

ok

### 💡 Use Cases:

- Daten-Sync aus externen Systemen:

Beim regelmäßigen Import (z.B. aus einer API oder CSV) sorgt ON CONFLICT dafür, dass vorhandene Datensätze aktualisiert statt dupliziert werden.

- Idempotente Pipelines (mehrfaches Ausführen = gleiches Ergebnis)

Das Upsert-Muster garantiert, dass mehrfaches Ausführen des Imports immer zum gleichen, korrekten Ergebnis führt – keine Duplikate, immer aktuelle Daten.

## UPDATE

### UPDATE mit WHERE

UPDATE ändert bestehende Daten. Der gefährlichste Befehl ist UPDATE ohne WHERE – dann werden ALLE Zeilen geändert. Immer mit WHERE filtern!

Syntax:

```
UPDATE table_name  
SET column1 = value1, column2 = value2, ...  
WHERE condition;
```

Beispiel:

```
1 -- Einzelne Zeile ändern:  
2 UPDATE customers_insert  
3 SET email = 'alice_updated@example.com'  
4 WHERE customer_id = 1;  
5  
6 SELECT * FROM customers_insert WHERE customer_id = 1;
```

```
-- Einzelne Zeile ändern:  
UPDATE customers_insert  
SET email = 'alice_updated@example.com'  
WHERE customer_id = 1
```

ok

```
SELECT * FROM customers_insert WHERE customer_id = 1
```

#	customer_id	name	email	registered_at
1	1	Alice	alice_updated@example.com	2026-02-13T10:23:03.834Z

1 rows

### ⚠ GEFAHR: UPDATE ohne WHERE

```
-- ALLE Zeilen werden geändert!  
UPDATE customers_insert  
SET name = 'Unknown';  
  
-- Jetzt heißen ALLE Kunden "Unknown"!
```

💡 Best Practice: Immer WHERE nutzen, außer Sie wollen wirklich alle Zeilen ändern.

### Mehrere Spalten & berechnete Updates

Sie können mehrere Spalten gleichzeitig ändern und berechnete Updates machen.

#### Mehrere Spalten:

```
UPDATE customers_insert  
SET  
    name = 'Alice Smith',  
    email = 'alice.smith@example.com'  
WHERE customer_id = 1;
```

#### Berechnete Updates:

```
1 CREATE TABLE products_update (  
2     product_id INTEGER PRIMARY KEY,  
3     name TEXT,  
4     price DECIMAL(10,2),  
5     stock INTEGER  
6 );  
7  
8 INSERT INTO products_update VALUES  
9     (1, 'Laptop', 1000.00, 50),  
10    ...
```

```
10      (2, 'Mouse', 25.00, 200);
11
12  -- Preiserhöhung um 10%:
13  UPDATE products_update
14  SET price = price * 1.10;
15
16  -- Stock reduzieren:
17  UPDATE products_update
18  SET stock = stock - 5
19  WHERE product_id = 1;
20
21  SELECT * FROM products_update;
```

```
CREATE TABLE products_update (
    product_id INTEGER PRIMARY KEY,
    name TEXT,
    price DECIMAL(10,2),
    stock INTEGER
)
```

ok

```
INSERT INTO products_update VALUES
(1, 'Laptop', 1000.00, 50),
(2, 'Mouse', 25.00, 200)
```

ok

```
-- Preiserhöhung um 10%:
UPDATE products_update
SET price = price * 1.10
```

ok

```
-- Stock reduzieren:
UPDATE products_update
SET stock = stock - 5
WHERE product_id = 1
```

ok

```
SELECT * FROM products_update
```

#	product_id	name	price	stock
1	2	Mouse	27.50	200
2	1	Laptop	1100.00	45

2 rows

## UPDATE mit Subquery (Fortgeschritten)

UPDATE mit Subqueries oder Joins ist fortgeschritten, aber sehr mächtig. Sie können Werte aus anderen Tabellen holen und einfügen.

Beispiel: Preis basierend auf Kategorie anpassen

```
1 ▾ CREATE TABLE categories_update (
2     category_id INTEGER PRIMARY KEY,
3     name TEXT,
4     discount_percent DECIMAL(5,2)
5 )
```



```
5  );
6
7 * CREATE TABLE products_cat (
8     product_id INTEGER PRIMARY KEY,
9     name TEXT,
10    price DECIMAL(10,2),
11    category_id INTEGER,
12
13    FOREIGN KEY (category_id) REFERENCES categories_update(category_id)
14 );
15
16 INSERT INTO categories_update VALUES
17     (1, 'Electronics', 10.00),
18     (2, 'Books', 5.00);
19
20 INSERT INTO products_cat VALUES
21     (1, 'Laptop', 1000.00, 1),
22     (2, 'Novel', 20.00, 2);
23
24 -- Preis mit Kategorie-Discount reduzieren:
25 UPDATE products_cat
26 * SET price = price * (1 - (
27     SELECT discount_percent / 100
28     FROM categories_update
29     WHERE categories_update.category_id = products_cat.category_id
30 ));
31
32 SELECT * FROM products_cat;
```

```
CREATE TABLE categories_update (
    category_id INTEGER PRIMARY KEY,
    name TEXT,
    discount_percent DECIMAL(5,2)
)
```

*ok*

```
CREATE TABLE products_cat (
    product_id INTEGER PRIMARY KEY,
    name TEXT,
    price DECIMAL(10,2),
    category_id INTEGER,

    FOREIGN KEY (category_id) REFERENCES categories_update(category_id)
)
```

*ok*

```
INSERT INTO categories_update VALUES
    (1, 'Electronics', 10.00),
    (2, 'Books', 5.00)
```

*ok*

```
INSERT INTO products_cat VALUES
    (1, 'Laptop', 1000.00, 1),
    (2, 'Novel', 20.00, 2)
```

*ok*

```
-- Preis mit Kategorie-Discount reduzieren:
UPDATE products_cat
SET price = price * (1 - (
    SELECT discount_percent / 100
    FROM categories_update
    WHERE categories_update.category_id = products_cat.category_id
))
```

*ok*

```
SELECT * FROM products_cat
```

#	product_id	name	price	category_id
1	1	Laptop	900.00	1
2	2	Novel	19.00	2

2 rows

## DELETE

### DELETE mit WHERE

DELETE entfernt Zeilen. Wie bei UPDATE gilt: Immer mit WHERE, außer Sie wollen wirklich alles löschen.  
DELETE ist reversibel (via Transaktion), TRUNCATE nicht.

Syntax:

```
DELETE FROM table_name  
WHERE condition;
```

Beispiel:

```
1 -- Einzelne Zeile löschen:  
2 DELETE FROM customers_insert  
3 WHERE customer_id = 5;  
4  
5 -- Mehrere Zeilen:  
6 DELETE FROM customers_insert  
7 WHERE email IS NULL;  
8  
9 SELECT * FROM customers_insert;
```

```
-- Einzelne Zeile löschen:  
DELETE FROM customers_insert  
WHERE customer_id = 5
```

ok

```
-- Mehrere Zeilen:  
DELETE FROM customers_insert  
WHERE email IS NULL
```

ok

```
SELECT * FROM customers_insert
```

#	customer_id	name	email	registered_at
1	3	Charlie	charlie@example.com	2026-02-13T10:23:04.777Z
2	4	Diana	diana@example.com	2026-02-13T10:23:04.777Z
3	1	Alice	alice_updated@example.com	2026-02-13T10:23:03.834Z

3 rows

### ⚠ GEFAHR: DELETE ohne WHERE

```
-- ALLE Zeilen werden gelöscht!  
DELETE FROM customers_insert;  
  
-- Tabelle ist jetzt leer!
```



## TRUNCATE vs. DELETE

TRUNCATE vs. DELETE: TRUNCATE ist schneller, aber weniger flexibel. DELETE kann mit WHERE filtern und ist in Transaktionen reversibel.

Feature	DELETE	TRUNCATE
WHERE-Klausel	✓ Ja	✗ Nein (alle Zeilen)
Performance	⚠ Langsamer (Zeile für Zeile)	✓ Schneller (gesamte Tabelle)
Rollback	✓ In Transaktion möglich	⚠ Meist nicht (DB-abhängig)
Triggers	✓ Werden ausgelöst	✗ Meist nicht
Auto-Increment Reset	✗ Nein	✓ Ja (zurück auf 1)

Beispiel:

```
-- DELETE: Kann WHERE nutzen
DELETE FROM products_update WHERE price < 50;

-- TRUNCATE: Löscht alles
TRUNCATE TABLE products_update;
```

### 💡 Wann was?

- **DELETE:** Selektives Löschen, Transaktionen wichtig
- **TRUNCATE:** Komplettes Leeren, Performance wichtig

### Soft Delete Pattern

Soft Delete ist ein Pattern, bei dem Sie Daten nicht wirklich löschen, sondern nur als „gelöscht“ markieren. Praktisch für Audit-Trails und Wiederherstellung.

**Problem:** Gelöschte Daten sind weg – kein Audit-Trail, keine Wiederherstellung.

**Lösung:** Status-Flag

```
1 * CREATE TABLE users_soft (
2     user_id INTEGER PRIMARY KEY,
3     username TEXT,
4     email TEXT,
5     is_deleted BOOLEAN DEFAULT FALSE,
6     deleted_at TIMESTAMP
7 );
8
9 INSERT INTO users_soft (user_id, username, email) VALUES
10    (1, 'alice', 'alice@example.com'),
11    (2, 'bob', 'bob@example.com');
12
13 -- Statt DELETE:
14 UPDATE users_soft+
```

```
14 UPDATE users_soft
15 SET is_deleted = TRUE, deleted_at = CURRENT_TIMESTAMP
16 WHERE user_id = 1;
17
18 -- View für aktive User:
19 CREATE VIEW active_users AS
20 SELECT * FROM users_soft WHERE is_deleted = FALSE;
21
22 SELECT * FROM active_users;
```

```
CREATE TABLE users_soft (
    user_id INTEGER PRIMARY KEY,
    username TEXT,
    email TEXT,
    is_deleted BOOLEAN DEFAULT FALSE,
    deleted_at TIMESTAMP
)
```

ok

```
INSERT INTO users_soft (user_id, username, email) VALUES
(1, 'alice', 'alice@example.com'),
(2, 'bob', 'bob@example.com')
```

ok

```
-- Statt DELETE:
UPDATE users_soft
SET is_deleted = TRUE, deleted_at = CURRENT_TIMESTAMP
WHERE user_id = 1
```

ok

```
-- View für aktive User:
CREATE VIEW active_users AS
SELECT * FROM users_soft WHERE is_deleted = FALSE
```

ok

```
SELECT * FROM active_users
```

#	user_id	username	email	is_deleted	deleted_at
1	2	bob	bob@example.com	false	null

1 rows

Vorteile:

- Wiederherstellung möglich (SET is\_deleted = FALSE)
- Audit-Trail (wann wurde gelöscht?)
- Analytics über gelöschte Daten

#### Nachteile:

- Tabelle wird größer
- Queries komplexer (immer WHERE is\_deleted = FALSE)

## Zusammenfassung

Was haben Sie gelernt? DDL für Schema-Design: CREATE TABLE mit Datentypen und Constraints, ALTER TABLE für Änderungen, DROP TABLE zum Löschen. DML für Datenmanipulation: INSERT zum Einfügen, UPDATE zum Ändern, DELETE zum Löschen. Constraints für Integrität: PRIMARY KEY, FOREIGN KEY, UNIQUE, NOT NULL, CHECK, DEFAULT. Und Best Practices für sichere Schema-Evolution.

Konzept	Befehl	Zweck
DDL	<code>CREATE TABLE</code>	Tabelle erstellen
	<code>ALTER TABLE</code>	Tabelle ändern
	<code>DROP TABLE</code>	Tabelle löschen
Constraints	<code>PRIMARY KEY</code>	Eindeutigkeit + NOT NULL
	<code>FOREIGN KEY</code>	Referenzielle Integrität
	<code>UNIQUE</code>	Eindeutigkeit (NULL erlaubt)
	<code>NOT NULL</code>	Pflichtfeld
	<code>CHECK</code>	Benutzerdefinierte Validierung
	<code>DEFAULT</code>	Standardwert
DML	<code>INSERT</code>	Daten einfügen
	<code>UPDATE</code>	Daten ändern
	<code>DELETE</code>	Daten löschen

Die wichtigsten Takeaways: Nutzen Sie Constraints – sie schützen Ihre Daten. Immer WHERE bei UPDATE/DELETE – außer Sie wollen wirklich alles ändern. Künstliche Primary Keys sind meist besser als natürliche. FOREIGN KEY mit ON DELETE/UPDATE steuert Kaskaden. Und: Schema-Evolution ist ein Prozess, keine einmalige Aktion.

## Best Practices: Checkliste

Zum Abschluss eine Checkliste, die Sie bei jedem Schema-Design durchgehen sollten.

### Schema-Design:

- Jede Tabelle hat einen PRIMARY KEY
- Künstliche Keys (INTEGER) statt natürliche (TEXT) für Performance
- FOREIGN KEYS für alle Beziehungen definiert
- ON DELETE/UPDATE explizit gewählt (CASCADE/RESTRICT/SET NULL)
- NOT NULL für alle Pflichtfelder
- CHECK Constraints für Validierung (z.B. price >= 0)
- DEFAULT für sinnvolle Standardwerte (z.B. created\_at)

### Datenmanipulation:

- INSERT: Spalten explizit benennen, nicht auf Reihenfolge verlassen
- UPDATE: Immer mit WHERE (außer wirklich alle Zeilen ändern)
- DELETE: Immer mit WHERE (außer wirklich alle Zeilen löschen)
- Bulk Operations nutzen (1 INSERT mit 100 Zeilen statt 100 INSERTs)

### Schema-Evolution:

- Jede Änderung hat UP + DOWN Migration
- Rückwärtskompatibilität beachten (Expand-Contract)
- Große Änderungen off-peak ausführen

### Sicherheit:

- Keine DDL/DML in Produktion ohne Backup
  - Transaktionen für multi-step Operationen (ab Session 11)
  - Testen auf Testdatenbank vor Produktion
-

# Quiz: Testen Sie Ihr Wissen

**Frage 1: Was ist der Unterschied zwischen PRIMARY KEY und UNIQUE?**

- Kein Unterschied
- PRIMARY KEY ist eindeutig + NOT NULL, UNIQUE erlaubt NULL
- UNIQUE ist schneller als PRIMARY KEY
- PRIMARY KEY kann mehrfach pro Tabelle vorkommen

**Frage 2: Was passiert bei ON DELETE CASCADE?**

- Löschen wird verhindert
- Abhängige Zeilen werden auch gelöscht
- Foreign Key wird auf NULL gesetzt
- Nichts

**Frage 3: Was macht TRUNCATE im Vergleich zu DELETE?**

- TRUNCATE ist schneller, löscht alle Zeilen, kein WHERE möglich
- TRUNCATE ist langsamer als DELETE
- TRUNCATE löscht nur eine Zeile
- Kein Unterschied

**Frage 4: Warum sollten Sie UPDATE ohne WHERE vermeiden?**

- Es ist langsamer
- Es funktioniert nicht
- Es ändert ALLE Zeilen in der Tabelle
- Es ist unsicher (SQL Injection)

**Frage 5: Was ist der Vorteil von künstlichen Primary Keys (INTEGER) gegenüber natürlichen (z.B. E-Mail)?**

- Künstliche Keys sind lesbarer
  - Künstliche Keys sind unveränderlich und schneller
  - Natürliche Keys sind besser
  - Kein Unterschied
-

# Übungsaufgaben

Zeit für Praxis! Probieren Sie diese Aufgaben selbst aus.

## Aufgabe 1: Tabelle erstellen

Erstellen Sie eine **students** Tabelle mit:

- **student\_id** (Primary Key, Integer)
- **first\_name** und **last\_name** (NOT NULL)
- **email** (UNIQUE, NOT NULL)
- **enrollment\_date** (DEFAULT: aktuelles Datum)
- **gpa** (CHECK: zwischen 0.0 und 4.0)

1  
2  
3  
4  
5  
6  
7  
8



► Lösung anzeigen

## Aufgabe 2: Foreign Key

Erstellen Sie eine **enrollments** Tabelle, die **students** mit **courses** verbindet:

- Composite Primary Key (**studentid, courseid**)
- Foreign Keys zu beiden Tabellen
- ON DELETE CASCADE für beide

1  
2  
3  
4  
5  
6  
7  
8



► Lösung anzeigen

### Aufgabe 3: INSERT & UPDATE

1. Fügen Sie 3 Students ein
2. Fügen Sie 2 Courses ein
3. Enrollen Sie Students in Courses
4. Aktualisieren Sie den GPA eines Students

1  
2  
3  
4  
5  
6  
7  
8



► Lösung anzeigen

## Ausblick: Was kommt als Nächstes?

Sie können jetzt Schemas erstellen, Daten einfügen, ändern, löschen. Aber Ihre Queries sind noch auf eine Tabelle beschränkt. Was, wenn Sie Daten aus mehreren Tabellen kombinieren wollen? Das sind Joins – unser nächstes großes Thema.

Kommende Sessions:

- **Session 9:** SQL Filtering & Operators (BETWEEN, IN, LIKE, CASE)
- **Session 10:** SQL Joins & Combining Data (INNER, LEFT, RIGHT, FULL, CROSS)
- **Session 11+:** Transaktionen & ACID (nach Joins)
- **Session 12:** Aggregation & Window Functions
- **Session 15:** Performance Optimization & Indexing

 Glückwunsch! Sie beherrschen jetzt DDL & DML – das Fundament jeder Datenbank-Arbeit!