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
- **Best Practices:** Sichere Schema-Evolution, häufige Fehler vermeiden

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.

Datenbank vorbereiten

Wir starten mit einer einfachen Sandbox-Datenbank. Keine Sorge – alles läuft im Browser, nichts wird dauerhaft gespeichert. Sie können experimentieren, Fehler machen, lernen.

```sql
-- Sandbox initialisieren
CREATE TABLE IF NOT EXISTS demo_test (id INTEGER, name TEXT);
INSERT INTO demo_test VALUES (1, 'Test');
SELECT 'Datenbank bereit!' AS status;
```

```
-- Sandbox initialisieren
CREATE TABLE IF NOT EXISTS demo_test (id INTEGER, name TEXT)
```

ok

```
INSERT INTO demo_test VALUES (1, 'Test')
```

ok

```
SELECT 'Datenbank bereit!' AS status
```

#	status
1	Datenbank bereit!

1 rows

```sql
-- Interaktives Terminal (nutzen Sie es für eigene Experimente)
SELECT * FROM demo_test;
```

```
-- Interaktives Terminal (nutzen Sie es für eigene Experimente)
SELECT * FROM demo_test
```

#	id	name
1	1	Test

1 rows

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	`CREATE`, `ALTER`, `DROP`	**Heute**
Data Manipulation Language	DML	Daten lesen/schreiben	`SELECT`, `INSERT`, `UPDATE`, `DELETE`	**Heute**
Data Control Language	DCL	Zugriffsrechte	`GRANT`, `REVOKE`	Später
Transaction Control Language	TCL	Transaktionen	`BEGIN`, `COMMIT`, `ROLLBACK`	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).

DDL – 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:

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

```
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

```sql
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
);
```

```
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
```

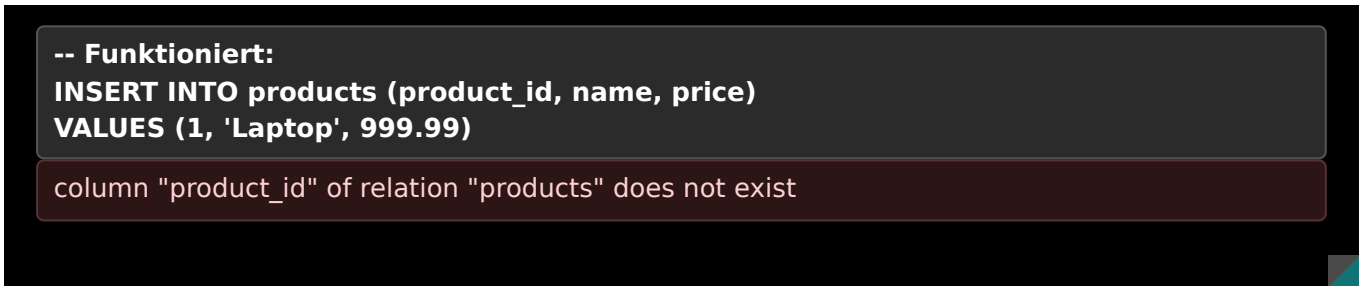
Was ist neu?

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

Testen:

```sql
-- Funktioniert:
INSERT INTO products (product_id, name, price)
VALUES (1, 'Laptop', 999.99);

-- Funktioniert NICHT (price negativ):
INSERT INTO products (product_id, name, price)
VALUES (2, 'Mouse', -10.00);
```



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):

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

Als Constraint (flexibel):

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

Composite Key (mehrere Spalten):

```sql
CREATE TABLE order_items (
  order_id INTEGER,
  product_id INTEGER,
```

```
  quantity INTEGER,
  PRIMARY KEY (order_id, product_id)
);
```

```
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^{31} bis $2^{31}-1$	4 Bytes	IDs, Zähler, ganze Zahlen
`BIGINT`	-2^{63} bis $2^{63}-1$	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?

```
-- FLOAT hat Rundungsfehler:
SELECT 0.1 + 0.2 AS float_sum;  -- Ergebnis: 0.30000000000000004
```

```
SELECT 0.1 + 0.2 AS float_sum;  -- Ergebnis: 0.30000000000000004

-- DECIMAL ist präzise:
SELECT CAST(0.1 AS DECIMAL(10,2)) + CAST(0.2 AS DECIMAL(10,2)) AS
  decimal_sum;
```

-- FLOAT hat Rundungsfehler:
SELECT 0.1 + 0.2 AS float_sum

#	float_sum
1	0.3

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:

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

```
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`	YYYY-MM-DD HH:MM:SS	2025-11-04 14:30:00	Logs, created_at
`INTERVAL`	Duration	'3 days', ‚2 hours‘	Zeitrechnungen

Beispiel: Zeitberechnungen

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

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

-- 3 Tage vor dem Event:
SELECT
  event_name,
  event_date,
  event_date - INTERVAL '3 days' AS reminder_date
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

```sql
CREATE TABLE products_ext (
  product_id INTEGER PRIMARY KEY,
```

```
  name TEXT,
  metadata JSON  -- Flexible Zusatzdaten
);

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

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

DDL – Tabellen ändern (ALTER TABLE)

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.

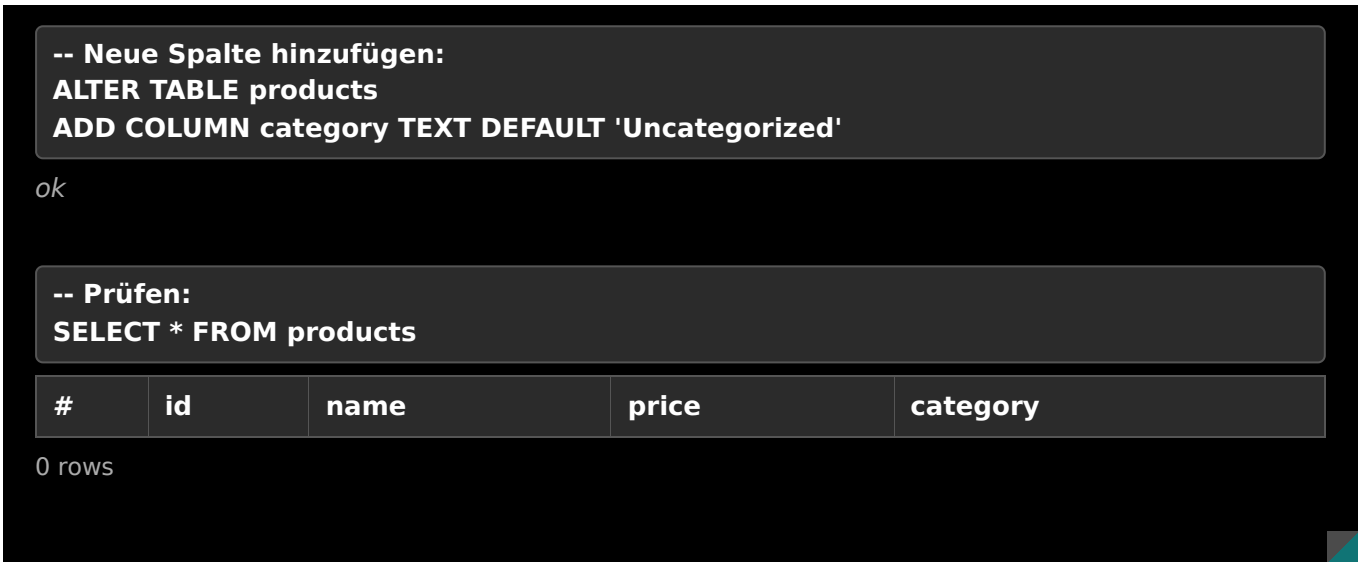
Spalten hinzufügen (ADD COLUMN)

Syntax:

```
ALTER TABLE table_name
ADD COLUMN column_name datatype [constraints];
```

Beispiel:

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



💡 **Best Practice:** Neue Spalten mit DEFAULT oder NULL hinzufügen, um Lock-Probleme zu vermeiden. Spalten ändern ist komplexer. Sie können Datentypen ändern, Defaults setzen, Constraints hinzufügen. Aber nicht alle Datenbanken unterstützen alle Operationen gleich.

Spalten ändern (ALTER COLUMN)

Datentyp ändern:

```
-- In PostgreSQL/PGlite.
ALTER TABLE products
ALTER COLUMN price TYPE DECIMAL(12, 2);

-- In MySQL:
ALTER TABLE products
MODIFY COLUMN price DECIMAL(12, 2);
```

Default setzen/ändern:

```
ALTER TABLE products
ALTER COLUMN stock SET DEFAULT 10;
```

⚠️ **Achtung bei Datentyp-Änderungen:**

- `TEXT` → `INTEGER`: Funktioniert nur, wenn alle Werte Zahlen sind
- `INTEGER` → `BIGINT`: Meist sicher
- Bei großen Tabellen: Kann lange dauern!

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.

Spalten löschen (DROP COLUMN)

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 fehlschlagen

Alternative: Soft Delete

Statt Spalte zu löschen:

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

-- Oder in Views weglassen:
CREATE VIEW products_view AS
SELECT product_id, name, price FROM products;
```

Tabellen können umbenannt werden. Das ist nützlich, wenn Sie Schema-Migrationen machen oder alte Versionen als Backup behalten wollen.

Tabelle umbenennen (RENAME TO)

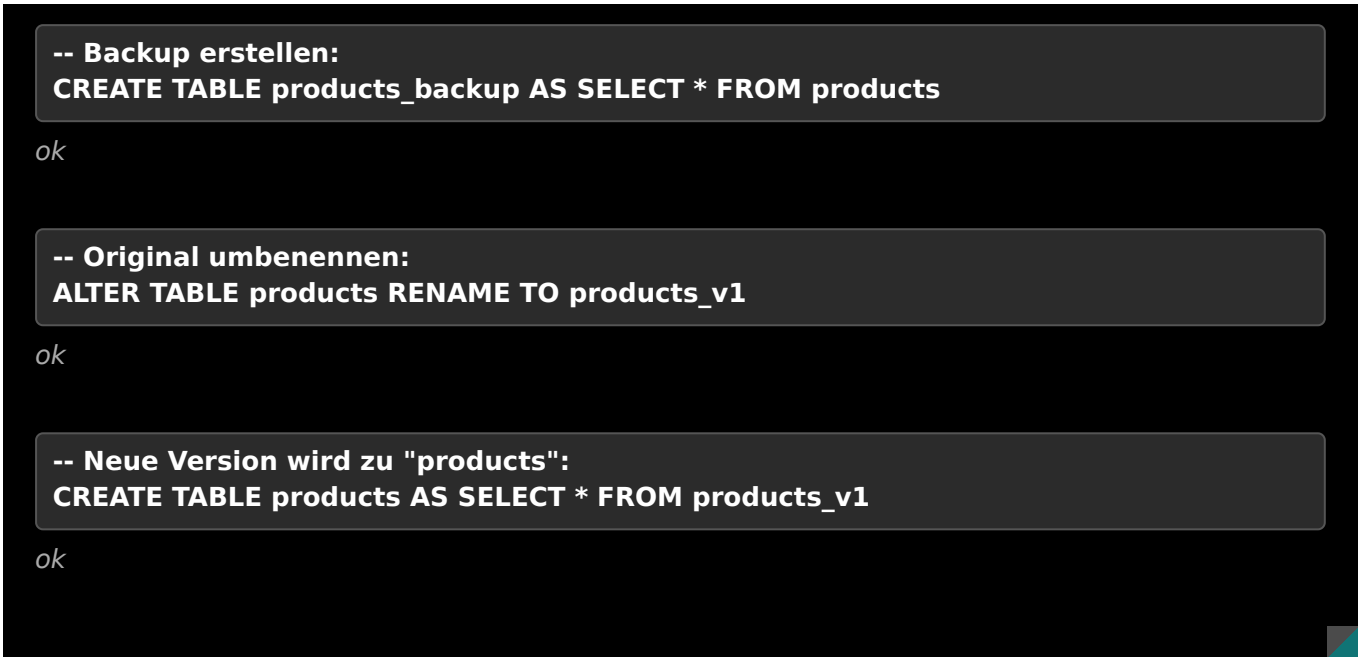
Syntax:

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

Beispiel:

```
1  -- Backup erstellen:
2  CREATE TABLE products_backup AS SELECT * FROM products;
3
4  -- Original umbenennen:
```

```
5  ALTER TABLE products RENAME TO products_v1;
6
7  -- Neue Version wird zu "products":
8  CREATE TABLE products AS SELECT * FROM products_v1;
```



DDL – Tabellen löschen (DROP TABLE)

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.

Grundsyntax

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):

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

CASCADE (Vorsicht!):

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

Beispiel:

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

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

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

-- Funktioniert (löscht auch FOREIGN KEY Constraint):
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

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.

Warum Constraints?

Ohne Constraints:

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

-- Alles erlaubt:
INSERT INTO orders_bad VALUES (1, NULL, -100);   -- ❌ Kein Kunde, negativer
INSERT INTO orders_bad VALUES (1, 999, 50);      -- ❌ Duplikat-ID, nicht
  -existierender Kunde
```

Mit Constraints:

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

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)
);

-- Schutz aktiviert:
INSERT INTO orders_good VALUES (1, NULL, 50);     -- ❌ customer_id NOT NULL
INSERT INTO orders_good VALUES (1, 999, 50);      -- ❌ customer_id existier
INSERT INTO orders_good VALUES (1, 1, -100);      -- ❌ total CHECK fehlschl
```

PRIMARY KEY

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 vs. Composite Keys

Single-Column (häufigster Fall):

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

Composite Key (mehrere Spalten):

```sql
CREATE TABLE enrollments (
  student_id INTEGER,
  course_id INTEGER,
  enrollment_date DATE,
  PRIMARY KEY (student_id, course_id)  -- Ein Student kann jeden Kurs 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: Natürlich = aus Daten (E-Mail, ISBN), künstlich = generiert (Auto-Increment ID). Künstliche Keys sind meist besser, weil sie unveränderlich sind.

Natürliche vs. künstliche Keys

Natürlicher Key (aus Daten):

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

Künstlicher Key (generiert):

```sql
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

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

FOREIGN KEY

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

Referenzielle Integrität

Beispiel: Kunden und Bestellungen

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

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)
);

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

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

-- Fehlschlägt (customer_id 999 existiert nicht):
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 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.

ON DELETE / ON UPDATE

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

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

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

INSERT INTO authors VALUES (1, 'Tolkien');
INSERT INTO books_cascade VALUES (1, 'Hobbit', 1), (2, 'LOTR', 1);

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

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

Self-Referencing (Hierarchien)

Beispiel: Mitarbeiter-Hierarchie

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

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

```
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 (parent*comment*id)

UNIQUE, NOT NULL, CHECK, DEFAULT

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

UNIQUE – Eindeutigkeit ohne Primary Key

Syntax:

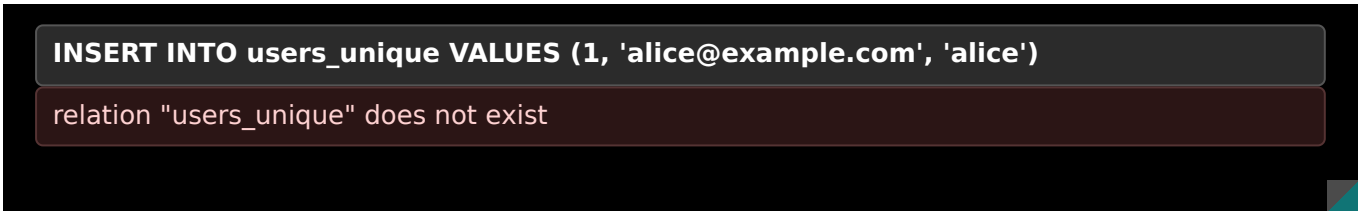
```
CREATE TABLE users_unique (
  user_id INTEGER PRIMARY KEY,
  email TEXT UNIQUE NOT NULL,
  username TEXT UNIQUE
);
```

Unterschied zu PRIMARY KEY:

- PRIMARY KEY: Eindeutig + NOT NULL + nur 1 pro Tabelle
- UNIQUE: Eindeutig, aber NULL erlaubt (mehrfach!), 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); -- O
     (NULL != NULL)
```



Composite UNIQUE:

```
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
);
```

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:

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

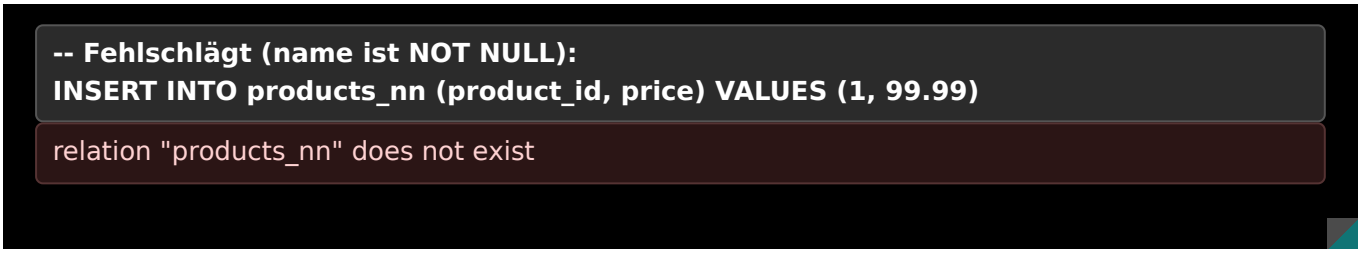
Warum wichtig?

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

Beispiel:

```
-- Fehlschlägt (name ist NOT NULL):
INSERT INTO products_nn (product_id, price) VALUES (1, 99.99);

-- Funktioniert:
INSERT INTO products_nn (product_id, name, price) VALUES (1, 'Laptop',
  .99);
```



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

CHECK – Benutzerdefinierte Validierung

Syntax:

```
CHECK (condition)
```

Beispiele:

```
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)
);
```

```
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:

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

Enum-Simulation:

```
1  CREATE TABLE orders_status (
2    order_id INTEGER PRIMARY KEY,
3    status TEXT CHECK (status IN ('pending', 'shipped', 'delivered',
       'cancelled'))
4  );
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')
```

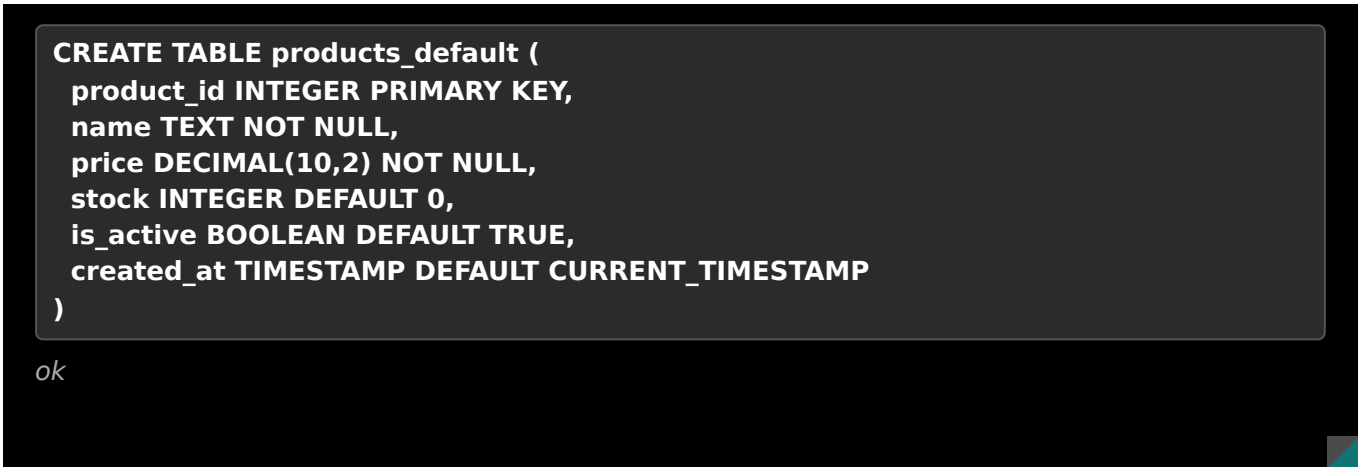
new row for relation "orders_status" violates check constraint "orders_status_status_check"

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

DEFAULT – Standardwerte

Syntax:

```sql
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
);
```



Nutzung:

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

SELECT * FROM products_default;
-- → 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**	2026-02-13T10:25:15.656Z

1 rows

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

ok

Funktionen als Default:

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

DML – INSERT

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.

Einzelne Zeile einfügen

Syntax:

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

Beispiel:

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

```
);

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

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	2026-02-13T10:25:11.820Z

1 rows

Alle Spalten (Reihenfolge wie in CREATE TABLE):

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

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

Mehrere Zeilen gleichzeitig (Bulk Insert)

Syntax:

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

Beispiel:

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

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	2026-02-13T10:25:11.820Z
2	3	Charlie	charlie@example.com	2026-02-13T10:25:12.808Z
3	4	Diana	diana@example.com	2026-02-13T10:25:12.808Z
4	5	Eve	eve@example.com	2026-02-13T10:25:12.808Z

4 rows

Performance-Vorteil:

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

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

INSERT ... SELECT

Syntax:

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

Beispiel: Backup erstellen

```
CREATE TABLE customers_backup (
  customer_id INTEGER,
  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;
```

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

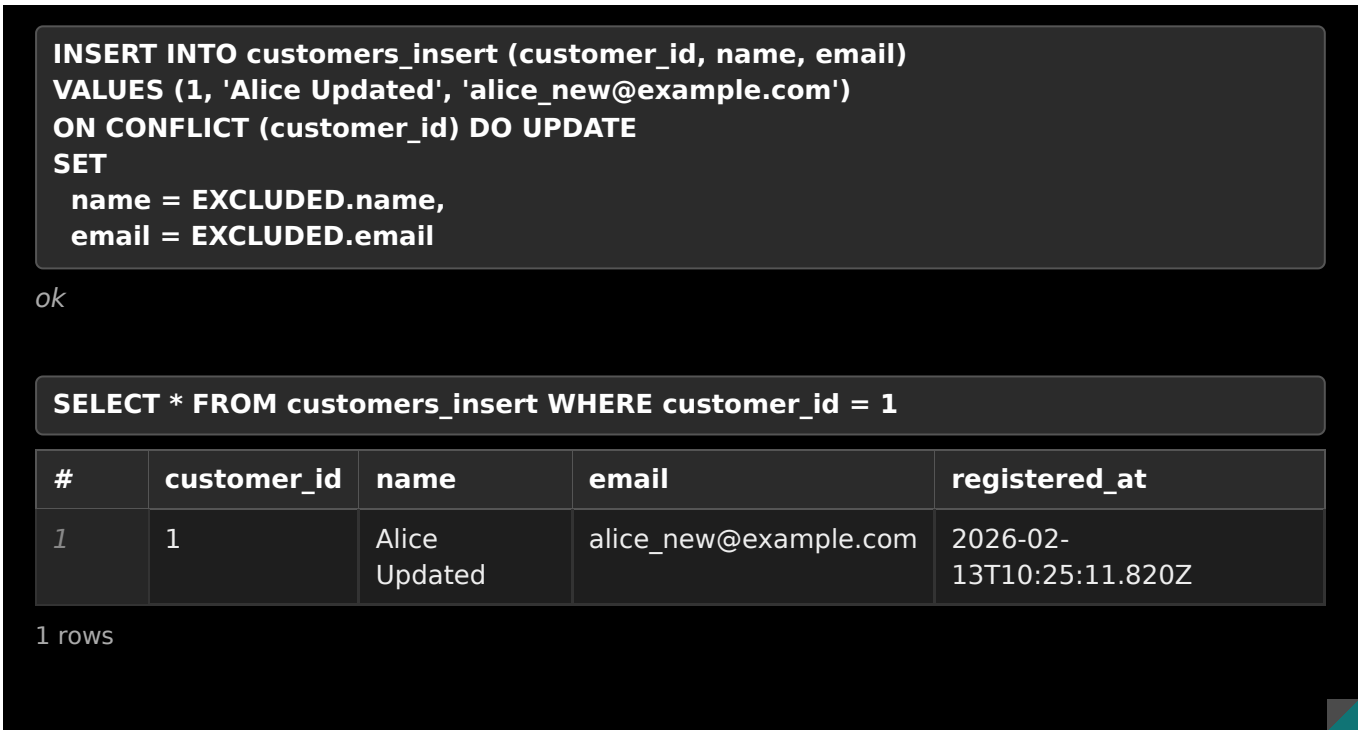
INSERT … ON CONFLICT (Upsert)

Problem: Was, wenn die ID schon existiert?

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

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	2026-02-13T10:25:11.820Z

1 rows

ON CONFLICT DO NOTHING:

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

💡 **Use Cases:**

- Daten-Sync aus externen Systemen
- Idempotente Pipelines (mehrfaches Ausführen = gleiches Ergebnis)

DML – UPDATE

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

UPDATE mit WHERE

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	2026-02-13T10:25:11.820Z

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.

Sie können mehrere Spalten gleichzeitig ändern und berechnete Updates machen.

Mehrere Spalten & berechnete Updates

Mehrere Spalten:

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

Berechnete Updates:

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

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

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

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

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

UPDATE mit Subquery (Fortgeschritten)

Beispiel: Preis basierend auf Kategorie anpassen

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

```sql
);

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

INSERT INTO categories_update VALUES (1, 'Electronics', 10.00), (2, '
  , 5.00);
INSERT INTO products_cat VALUES
  (1, 'Laptop', 1000.00, 1),
  (2, 'Novel', 20.00, 2);

-- 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
));

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
)
```

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

DML – DELETE

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

DELETE mit WHERE

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	2026-02-13T10:25:12.808Z
2	4	Diana	diana@example.com	2026-02-13T10:25:12.808Z
3	1	Alice	alice_updated@example.com	2026-02-13T10:25:11.820Z

3 rows

⚠️ **GEFAHR: DELETE ohne WHERE**

```
-- ALLE Zeilen werden gelöscht!
DELETE FROM customers_insert;

-- Tabelle ist jetzt leer!
```

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

TRUNCATE vs. DELETE

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

Soft Delete Pattern

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

Lösung: Status-Flag

```
1 CREATE TABLE users_soft (
2   user id INTEGER PRIMARY KEY,
```

```
  username TEXT,
  email TEXT,
  is_deleted BOOLEAN DEFAULT FALSE,
  deleted_at TIMESTAMP
);

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

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

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

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)

Schema-Evolution & Best Practices

Schemas ändern sich im Lauf der Zeit. Neue Features erfordern neue Spalten, Refactorings ändern Strukturen. Wie machen Sie das sicher, ohne Downtime, ohne Datenverlust?

Migrations-Konzept

Problem: Schema-Änderungen müssen nachvollziehbar und wiederholbar sein.

Lösung: Migrations (Up/Down)

```
-- Migration 001: Initial Schema
-- UP:
CREATE TABLE users (
  user_id INTEGER PRIMARY KEY,
  username TEXT NOT NULL
);

-- DOWN:
DROP TABLE users;
```

```
-- Migration 002: Add Email
-- UP:
ALTER TABLE users ADD COLUMN email TEXT;

-- DOWN:
ALTER TABLE users DROP COLUMN email;
```

Tools:

- **Flyway** (Java): SQL-basiert, einfach
- **Liquibase** (Java): XML/YAML, komplex aber mächtig
- **Alembic** (Python): Code-basiert, für SQLAlchemy
- **Migrate** (Go): Einfach, Library

Workflow:

1. Entwicklung: Neue Migration schreiben
2. Review: Migration prüfen (Syntax, Logik)
3. Test: Auf Testdatenbank anwenden
4. Produktion: Rollout mit Monitoring

Sichere Schema-Änderungen vermeiden Downtime. ADD COLUMN ist meist sicher, DROP COLUMN riskant. Große Tabellen erfordern besondere Vorsicht.

Sichere Schema-Änderungen

✅ **Sicher (keine Downtime):**

```
-- Spalte mit DEFAULT hinzufügen:
ALTER TABLE products ADD COLUMN category TEXT DEFAULT 'Uncategorized';

-- Index erstellen (CONCURRENT in PostgreSQL):
CREATE INDEX CONCURRENTLY idx_products_category ON products(category);
```

⚠️ **Riskant (Lock/Downtime):**

```
-- Datentyp ändern (gesamte Tabelle wird gesperrt):
ALTER TABLE products ALTER COLUMN price TYPE DECIMAL(12,2);

-- Spalte löschen (Lock):
ALTER TABLE products DROP COLUMN description;
```

💡 **Best Practices:**

1. **ADD COLUMN mit DEFAULT:** Schnell, keine Lock-Probleme
2. **NOT NULL schrittweise:** - Schritt 1: Spalte als NULL hinzufügen - Schritt 2: Werte füllen (UPDATE) - Schritt 3: NOT NULL Constraint hinzufügen
3. **Große Tabellen:** Off-Peak-Zeiten nutzen
4. **Indexes:** CONCURRENT erstellen (PostgreSQL)

Rückwärtskompatibilität ist wichtig, wenn mehrere App-Versionen parallel laufen. Neue Spalten sollten optional sein, alte Spalten nicht sofort gelöscht werden.

Rückwärtskompatibilität

Problem: App v1 läuft noch, aber DB-Schema ist für App v2.

Strategie: Expand-Contract

1. **Expand:** Neue Spalte hinzufügen (optional)
2. **Migrate:** App v2 deployed, nutzt neue Spalte
3. **Contract:** Nach Rollout alte Spalte löschen

Beispiel:

```
-- Phase 1: EXPAND (neue Spalte hinzufügen)
ALTER TABLE users ADD COLUMN email_new TEXT;

-- Phase 2: MIGRATE
```

```
-- App v2 schreibt in email_new
-- App v1 schreibt weiter in email

-- Phase 3: CONTRACT (nach vollständigem Rollout)
ALTER TABLE users DROP COLUMN email;
ALTER TABLE users RENAME COLUMN email_new TO email;
```

💡 **Best Practice:** Niemals breaking changes ohne Übergangsphase!

Ausblick: Transaktionen

Ein letzter Punkt, den wir heute nur kurz anreißen: Transaktionen. Sie haben INSERT, UPDATE, DELETE gelernt – aber was, wenn Sie mehrere Operationen atomar ausführen wollen? „Entweder alles oder nichts"? Das sind Transaktionen.

Warum Transaktionen?

Problem:

```
-- Geldtransfer:
UPDATE accounts SET balance = balance - 100 WHERE account_id = 1;  -- ✅ OK
-- ❌ Fehler! Server-Crash!
UPDATE accounts SET balance = balance + 100 WHERE account_id = 2;  -- Wird
  ausgeführt
-- → 100 Euro verschwunden!
```

Lösung: Transaktion

```
BEGIN;
  UPDATE accounts SET balance = balance - 100 WHERE account_id = 1;
  UPDATE accounts SET balance = balance + 100 WHERE account_id = 2;
COMMIT;  -- Beide oder keine
```

Wenn Fehler:

```
BEGIN;
  UPDATE accounts SET balance = balance - 100 WHERE account_id = 1;
  -- Fehler hier!
ROLLBACK;  -- Alles rückgängig
```

Transaktionen lernen Sie ausführlich in Session 11+, nach Joins. Warum später? Weil Transaktionen erst bei Multi-Table-Operations richtig relevant werden. Für heute reicht: Sie existieren, sie garantieren ACID (Atomicity, Consistency, Isolation, Durability), und wir kommen darauf zurück.

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	`CREATE TABLE`	Tabelle erstellen
	`ALTER TABLE`	Tabelle ändern
	`DROP TABLE`	Tabelle löschen
Constraints	`PRIMARY KEY`	Eindeutigkeit + NOT NULL
	`FOREIGN KEY`	Referenzielle Integrität
	`UNIQUE`	Eindeutigkeit (NULL erlaubt)
	`NOT NULL`	Pflichtfeld
	`CHECK`	Benutzerdefinierte Validierung
	`DEFAULT`	Standardwert
DML	`INSERT`	Daten einfügen
	`UPDATE`	Daten ändern
	`DELETE`	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:

- [ ] Migrations-System nutzen (Flyway, Liquibase, Alembic)
- [ ] 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)

****

` sql CREATE TABLE students ( student_id INTEGER PRIMARY KEY, first_name TEXT NOT NULL, last_name TEXT NOT NULL, email TEXT UNIQUE NOT NULL, enrollment*date DATE DEFAULT CURRENT*DATE, gpa DECIMAL(3,2) CHECK (gpa BETWEEN 0.0 AND 4.0) ); ` LIA: terminal **********

Aufgabe 2: Foreign Key

Erstellen Sie eine `enrollments` Tabelle, die `students` mit `courses` verbindet: - Composite Primary Key (student*id, course*id) - Foreign Keys zu beiden Tabellen - ON DELETE CASCADE für beide

` sql INSERT INTO students (student*id, first*name, last_name, email, gpa) VALUES (1, ‚Alice‘, ‚Smith‘, 'alice@university.edu', 3.8), (2, ‚Bob‘, ‚Jones‘, 'bob@university.edu', 3.5), (3, ‚Charlie‘, ‚Brown‘, 'charlie@university.edu', 3.9); INSERT INTO courses (course_id, title) VALUES (1, ‚Databases‘), (2, ‚Algorithms‘); INSERT INTO enrollments (student*id, course*id, grade) VALUES (1, 1, ‚A‘), (1, 2, ‚B‘), (2, 1, ‚A-‘); UPDATE students SET gpa = 3.85 WHERE student_id = 1; SELECT * FROM students; SELECT * FROM enrollments; ` LIA: terminal **********

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!