SQL & Datenbanken

by Woche 11

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import sqlite3
np.random.seed(1) # damit die Zufallszahlen reproduzierbar sind
```

Stellt euch vor, ihr arbeitet für ein E-Commerce-Unternehmen und alle Verkaufsdaten liegen in einer großen Tabelle vor:

bestell_id	kunden_name	kunden_email	kunden_stadt	produkt_name	produkt_preis	produkt_kategorie	menge	datum
101	Lisa Schmidt	lisa@example.com	Hamburg	Laptop	899.99	Computer	1	2025-05-15
102	Lisa Schmidt	lisa@example.com	Hamburg	Maus	29.99	Zubehör	2	2025-05-16
103	Max Weber	max@example.com	Berlin	Monitor	299.99	Computer	1	2025-05-17
104	Anna Müller	anna@example.com	München	Tastatur	79.99	Zubehör	1	2025-05-18
105	Lisa Schmidt	lisa@example.com	Hamburg	Kopfhörer	149.99	Audio	1	2025-05-20

Auf den ersten Blick sieht das in Ordnung aus - alles in einer Datei. Aber diese Struktur führt schnell zu massiven Problemen: Vor allem die Redundanz bringt direkt mehrere Probleme mit sich. Da Lisa Schmidt dreimal bestellt hat, muss sie zwar auch dreimal in der Datei als Kundin identifiziert werden, dass dann aber auch ihre E-Mail-Adresse und ihr Wohnort dreimal auftauchen, ist nicht optimal. Auf der einen Seite wird so unnötig viel Speicherplatz verwendet und falls Lisa ihre E-Mail-Adresse ändert, muss das in mehreren Zeilen aktualisiert werden.

Mehrere Tabellen

Die Lösung: Wir teilen die Daten in separate, logische Tabellen auf, die über eindeutige IDs verknüpft sind:

kunden.csv:

kunden_id	name	email	stadt
1	Lisa Schmidt	lisa@example.com	Hamburg
2	Max Weber	max@example.com	Berlin
3	Anna Müller	anna@example.com	München

produkte.csv:

produkt_id	name	preis	kategorie
5	Laptop	899.99	Computer
8	Monitor	299.99	Computer
12	Maus	29.99	Zubehör
15	Tastatur	79.99	Zubehör
20	Kopfhörer	149.99	Audio

bestellungen.csv:

bestell_id	kunden_id	produkt_id	menge	datum
101	1	5	1	2025-05-15
102	1	12	2	2025-05-16
103	2	8	1	2025-05-17
104	3	15	1	2025-05-18
105	1	20	1	2025-05-20

So wird klar, dass wir die Redundanz in den Daten auf ein Minimum reduzieren Statt dreimal Lisa Schmidt zu speichern, haben wir sie nur einmal in der Kundentabelle. Das gleiche gilt für die Produkte. In der Bestellungen-Tabelle verweisen wir dann einfach auf die IDs der Kunden und Produkte.

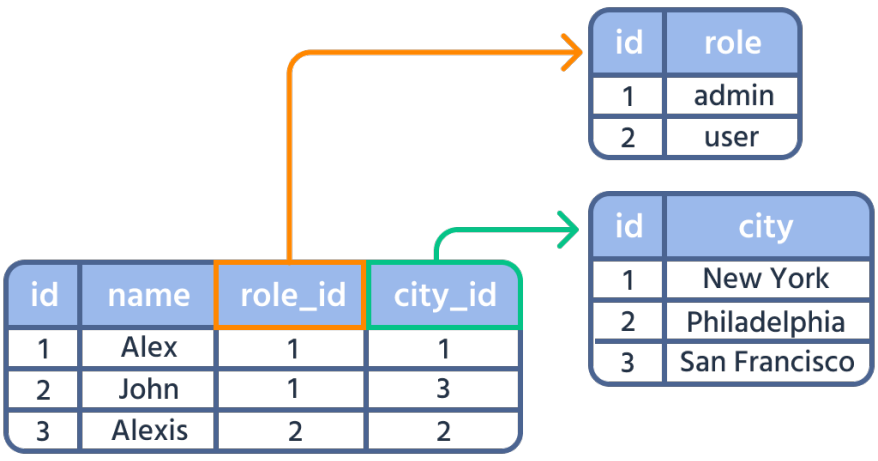
Als nächstes stellt sich die Frage wie man solche Datenstrukturen optimal verwalten kann. Natürlich könnte jede der obigen Tabellen als separate CSV-Datei gespeichert werden, doch das bringt einige Nachteile mit sich. Stattdessen werden in der Praxis normalerweise **Datenbanken** verwendet, die genau für solche Anwendungsfälle entwickelt wurden.

Was ist eine Datenbank?

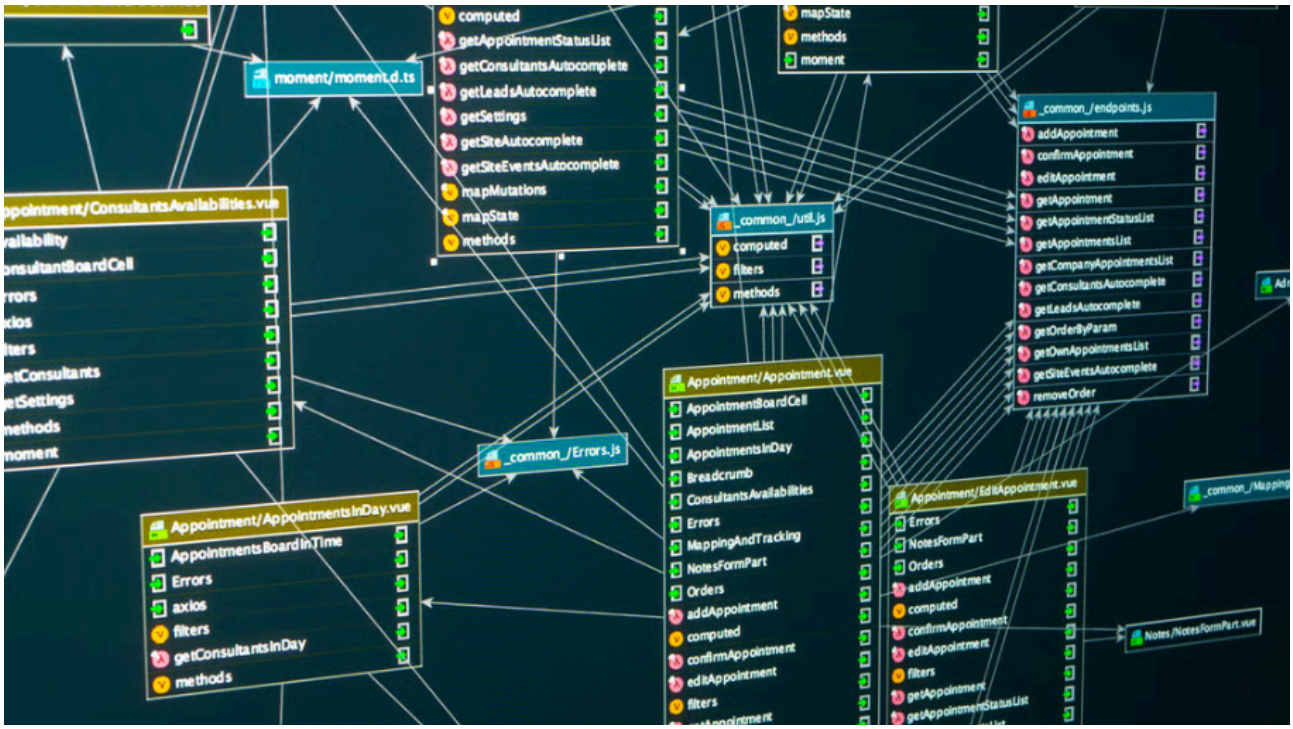
Eine Datenbank ist ein organisiertes System zur elektronischen Datenspeicherung. Sie ist darauf ausgelegt, Daten effizient abzurufen, zu speichern und zu verwalten. Während für kleine Datenmengen eine einfache CSV-Datei oft ausreichend ist, bieten Datenbanken viele Vorteile bei größeren Datenmengen oder komplexeren Anforderungen:

- **Effizienz**: Optimierte Speicherung und schneller Zugriff auf große Datenmengen
- **Datenkonsistenz**: Regeln und Einschränkungen (Constraints) können definiert werden, um die Datenintegrität sicherzustellen
- **Parallelität**: Mehrere Nutzer können gleichzeitig auf Daten zugreifen ohne die Integrität zu gefährden
- **Sicherheit**: Zugriffsrechte können detailliert konfiguriert werden
- **Skalierbarkeit**: Datenbanken können mit wachsenden Anforderungen mitwachsen, und beispielsweise auf mehrere Server verteilt werden

Das am weitesten verbreitete Datenbankmodell ist das *relationale Modell*, bei dem Daten in Tabellen (auch “Relationen” genannt) organisiert werden. Diese Tabellen stehen durch gemeinsame Felder/IDs in Beziehung zueinander - also genau wie unsere drei Tabellen oben.



Quelle: Codefinity

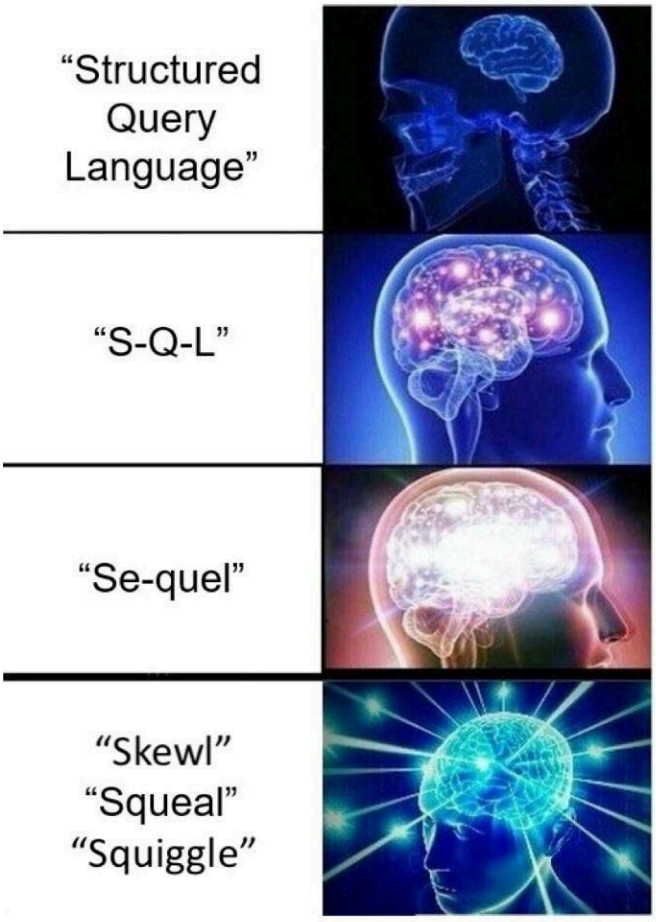


Quelle: Anina Ot

Was ist SQL?

SQL (Structured Query Language) ist eine standardisierte Sprache zum Arbeiten mit relationalen Datenbanken. Sie ermöglicht das Erstellen, Abfragen, Aktualisieren und Löschen von Daten. SQL wurde in den 1970er Jahren entwickelt und ist heute der Industriestandard für Datenbankoperationen.

> **i** APIs und Datenbanken
>
> Übrigens: Die APIs, die wir im letzten Kapitel kennengelernt haben, greifen in der Realität oft auf Datenbanken zu. Wenn ihr eine API-Anfrage stellt, wird diese häufig intern in SQL umgewandelt, um die entsprechenden Daten aus der Datenbank zu extrahieren und an euch zurückzusenden.



Wichtige Datenbanksysteme

Es gibt verschiedene relationale Datenbankverwaltungssysteme (RDBMS), die SQL unterstützen:

- **SQLite**: Eine leichtgewichtige, dateibasierte Datenbank, ideal für eingebettete Systeme oder lokale Anwendungen
- **MySQL/MariaDB**: Open-Source-Datenbank, häufig bei Webanwendungen eingesetzt
- **PostgreSQL**: Leistungsstarke Open-Source-Datenbank mit erweiterten Funktionen
- **Microsoft SQL Server**: Kommerzielles RDBMS von Microsoft
- **Oracle Database**: Kommerzielle Datenbanklösung für Unternehmen

All diese Datenbanken unterstützen die grundlegenden SQL-Befehle, unterscheiden sich hinter den Kulissen jedoch in ihrer Implementierung und ihren erweiterten Funktionen. Die für Data Scientists relevanten Grundfunktionen sind in diesen Systemen jedoch ähnlich.

Tabellenstruktur in relationalen Datenbanken

In einer relationalen Datenbank werden Daten in Tabellen gespeichert, die aus Zeilen (Records) und Spalten (Feldern) bestehen:

- **Tabelle**: Eine Sammlung verwandter Daten (z.B. “Kunden”, “Bestellungen”)
- **Spalte**: Ein benanntes Attribut mit einem bestimmten Datentyp (z.B. “Name” als Text)
- **Zeile**: Ein einzelner Datensatz in der Tabelle (z.B. alle Informationen zu einem bestimmten Kunden)
- **Primärschlüssel**: Ein eindeutiger Identifikator für jede Zeile (z.B. “Kundennummer”)
- **Fremdschlüssel**: Ein Feld, das auf den Primärschlüssel einer anderen Tabelle verweist, um Beziehungen zwischen verschiedenen Tabellen herzustellen (z.B. “Kundennummer” in der Bestellungen-Tabelle, die auf einen Kunden in der Kunden-Tabelle verweist)

Grundlegende SQL-Befehle

Stellt euch vor, ihr wollt aus unseren drei Tabellen herausfinden: “Welche Kunden aus Hamburg haben Laptops gekauft?” Mit CSV-Dateien müsstet ihr:

1. Die Kundentabelle nach Hamburg filtern
2. Die Bestellungen-Tabelle laden
3. Die Produkte-Tabelle laden
4. Drei separate Pandas-Merge-Operationen durchführen
5. Nach Laptops filtern

Mit SQL ist das eine einzige, lesbare Abfrage:

```sql
SELECT k.name
FROM Kunden k
JOIN Bestellungen b ON k.kunden_id = b.kunden_id
JOIN Produkte p ON b.produkt_id = p.produkt_id
WHERE k.stadt = 'Hamburg' AND p.name = 'Laptop';
```

SQL ist darauf optimiert, genau solche Fragen effizient zu beantworten. Die Datenbank-Engine weiß automatisch, wie sie die Tabellen am schnellsten verknüpft und filtert.

Betrachten wir aber erstmal kleinere Beispiele, um die grundlegenden SQL-Befehle zu verstehen.

SELECT: Daten abfragen

Der wichtigste SQL-Befehl ist `SELECT` - er holt Daten aus Tabellen:

```sql
SELECT name, email FROM Kunden;
```

Das liest sich fast wie ein deutscher Satz: “Wähle Name und E-Mail aus der Kunden-Tabelle.” Hier die Bestandteile:

- **SELECT**: “Ich möchte folgende Spalten sehen”
- **name, email**: Die spezifischen Spalten, die wir wollen
- **FROM Kunden**: “Aus dieser Tabelle”

```sql
SELECT * FROM Kunden;
```

Das Sternchen (`*`) bedeutet “alle Spalten” - wie ein Platzhalter. Es ist praktisch zum Erkunden, aber in echten Anwendungen solltet ihr spezifische Spalten nennen.

WHERE: Daten filtern

Mit `WHERE` fügen wir Bedingungen hinzu:

```sql
SELECT * FROM Kunden WHERE stadt = 'Hamburg';
```

Das entspricht `df[df['stadt'] == 'Hamburg']` in Pandas. Ihr könnt verschiedene Operatoren verwenden:

```sql
-- Gleichheit
SELECT * FROM Produkte WHERE kategorie = 'Computer';

-- Größer/kleiner
```

```sql
SELECT * FROM Produkte WHERE preis > 100;

-- Mehrere Bedingungen
SELECT * FROM Produkte WHERE kategorie = 'Computer' AND preis < 500;
```

> **i** SQL-Kommentare
>
> In SQL werden Kommentare mit `--` geschrieben, nicht mit `#` wie in Python. Alles nach `--` in einer Zeile wird ignoriert.

ORDER BY: Sortieren

Um Ergebnisse zu sortieren, verwenden wir `ORDER BY`:

```sql
-- Aufsteigend sortieren (Standard)
SELECT * FROM Produkte ORDER BY preis;

-- Absteigend sortieren
SELECT * FROM Produkte ORDER BY preis DESC;
```

Das entspricht `df.sort_values('preis')` bzw. `df.sort_values('preis', ascending=False)` in Pandas.

JOIN: Tabellen verbinden

Hier wird es interessant - und das ist der große Vorteil von SQL. Mit `JOIN` verbinden wir mehrere Tabellen:

```sql
SELECT k.name, p.name as produkt_name
FROM Kunden k
JOIN Bestellungen b ON k.kunden_id = b.kunden_id
JOIN Produkte p ON b.produkt_id = p.produkt_id;
```

Das ist das Äquivalent zu mehreren `pd.merge()`-Operationen in Pandas. Hier passiert folgendes:

1. **FROM Kunden k**: Beginne mit der Kunden-Tabelle (das “k” ist ein Alias, also Abkürzung)
2. **JOIN Bestellungen b ON k.kunden_id = b.kunden_id**: Füge die Bestellungen hinzu, verknüpft über die Kunden-ID
3. **JOIN Produkte p ON b.produkt_id = p.produkt_id**: Füge die Produkte hinzu, verknüpft über die Produkt-ID

GROUP BY: Aggregieren

Für Zusammenfassungen verwenden wir `GROUP BY`:

```sql
SELECT kategorie, COUNT(*) as anzahl, AVG(preis) as durchschnittspreis
FROM Produkte
GROUP BY kategorie;
```

Das entspricht `df.groupby('kategorie').agg({'preis': ['count', 'mean']})` in Pandas.

SQL bietet einige Aggregatfunktionen:

- `COUNT(*)`: Anzahl der Zeilen
- `SUM(spalte)`: Summe
- `AVG(spalte)`: Durchschnitt
- `MIN(spalte)`: Minimum
- `MAX(spalte)`: Maximum

Weitere SQL-Befehle

Der Vollständigkeit halber seien hier noch andere wichtige SQL-Befehle erwähnt, die jedoch für Data Scientists meist weniger relevant sind, da wir in der Regel keine Datenstrukturen ändern möchten:

```sql
-- Neue Daten einfügen
INSERT INTO Kunden (name, email, anschrift)
VALUES ('Lisa Schmidt', 'lisa@example.com', 'Testweg 3, 12345 München');

-- Bestehende Daten aktualisieren
UPDATE Kunden SET email = 'max.neu@example.com' WHERE kunden_id = 1;

-- Daten löschen
DELETE FROM Bestellungen WHERE bestell_id = 103;

-- Tabellen erstellen
CREATE TABLE Produkte (
    produkt_id INTEGER PRIMARY KEY,
    name TEXT NOT NULL,
    preis DECIMAL(10,2),
    bestand INTEGER DEFAULT 0
);
```

SQL in Python

Als Data Scientist wirst du SQL typischerweise in Kombination mit Python verwenden. Die Standardbibliothek von Python bietet bereits Unterstützung für SQLite, und für andere Datenbanksysteme gibt es spezifische Bibliotheken. SQLite ist eine leichtgewichtige Datenbank, die keine separate Server-Installation erfordert und direkt in Anwendungen eingebettet werden kann. Die Datenbank wird in einer einzelnen Datei gespeichert, was sie ideal für lokale Anwendungen macht.

Man interagiert mit SQLite Datenbanken in Python, indem man die SQL-Befehle in Strings verpackt und sie an die Datenbank sendet. Wollen wir also diesen Befehl in Python ausführen:

```sql
SELECT * FROM Kunden WHERE stadt = 'Hamburg';
```

So tut man dies in der Regel auf einer der folgenden Arten:

```python
cursor.execute("SELECT * FROM Kunden WHERE stadt = 'Hamburg'")
```

```python
cursor.execute('''
SELECT * FROM Kunden WHERE stadt = 'Hamburg'
''')
```

Datenbank erstellen und verwenden

Genug Theorie - wir erstellen nun eine echte Datenbank mit unseren E-Commerce-Daten.

Schritt 1: Verbindung zur Datenbank

```
import sqlite3

# Verbindung zur Datenbank erstellen
conn = sqlite3.connect('shop.db')
cursor = conn.cursor()

print("Datenbank-Verbindung erfolgreich!")
```

```
Datenbank-Verbindung erfolgreich!
```

Was passiert hier?

- `sqlite3.connect('shop.db')`: Erstellt oder öffnet eine SQLite-Datenbankdatei namens ‘shop.db’
- `cursor`: Ein Objekt, mit dem wir SQL-Befehle ausführen können - wie ein Zeiger in der Datenbank

Schritt 2: Tabellen erstellen

Jetzt erstellen wir unsere drei Tabellen. In SQL definieren wir dabei auch die Datentypen und Beziehungen:

```
# Kunden-Tabelle erstellen
cursor.execute('''
CREATE TABLE IF NOT EXISTS Kunden (
    kunden_id INTEGER PRIMARY KEY,
    name TEXT NOT NULL,
    email TEXT,
    stadt TEXT
)
''')

# Produkte-Tabelle erstellen
cursor.execute('''
CREATE TABLE IF NOT EXISTS Produkte (
    produkt_id INTEGER PRIMARY KEY,
    name TEXT NOT NULL,
    preis REAL,
    kategorie TEXT
)
''')

# Bestellungen-Tabelle erstellen
cursor.execute('''
CREATE TABLE IF NOT EXISTS Bestellungen (
    bestell_id INTEGER PRIMARY KEY,
    kunden_id INTEGER,
    produkt_id INTEGER,
    menge INTEGER,
    datum TEXT,
    FOREIGN KEY (kunden_id) REFERENCES Kunden (kunden_id),
    FOREIGN KEY (produkt_id) REFERENCES Produkte (produkt_id)
)
''')
```

```
print("Tabellen erfolgreich erstellt!")
```

```
<sqlite3.Cursor object at 0x0000022A4F4684C0>
<sqlite3.Cursor object at 0x0000022A4F4684C0>
<sqlite3.Cursor object at 0x0000022A4F4684C0>
Tabellen erfolgreich erstellt!
```

Hier die wichtigsten Konzepte:

- **IF NOT EXISTS**: Erstelle die Tabelle nur, wenn sie noch nicht existiert
- **PRIMARY KEY**: Eindeutige Identifikation für jede Zeile
- **NOT NULL**: Diese Spalte darf nicht leer sein
- **FOREIGN KEY**: Verweis auf eine andere Tabelle - das garantiert Datenintegrität
- **Datentypen**:
  - `INTEGER`: Ganzzahlen
  - `TEXT`: Strings
  - `REAL`: Dezimalzahlen

Schritt 3: Daten einfügen

```
# Kundendaten einfügen
kunden = [
    (1, "Lisa Schmidt", "lisa@example.com", "Hamburg"),
    (2, "Max Weber", "max@example.com", "Berlin"),
    (3, "Anna Müller", "anna@example.com", "München"),
    (4, "Tom Fischer", "tom@example.com", "Hamburg")
]

# Produktdaten einfügen
produkte = [
    (5, "Laptop", 899.99, "Computer"),
    (8, "Monitor", 299.99, "Computer"),
    (12, "Maus", 29.99, "Zubehör"),
    (15, "Tastatur", 79.99, "Zubehör"),
    (20, "Kopfhörer", 149.99, "Audio")
]

# Bestelldaten einfügen
bestellungen = [
    (101, 1, 5, 1, "2025-05-15"),  # Lisa: Laptop
    (102, 1, 12, 2, "2025-05-16"), # Lisa: 2x Maus
    (103, 2, 8, 1, "2025-05-17"),  # Max: Monitor
    (104, 3, 15, 1, "2025-05-18"), # Anna: Tastatur
    (105, 4, 5, 1, "2025-05-19"),  # Tom: Laptop
    (106, 1, 20, 1, "2025-05-20")  # Lisa: Kopfhörer
]

# Daten in die Tabellen einfügen
cursor.executemany("INSERT OR REPLACE INTO Kunden VALUES (?, ?, ?, ?)", kunden)
cursor.executemany("INSERT OR REPLACE INTO Produkte VALUES (?, ?, ?, ?)", produkte)
cursor.executemany("INSERT OR REPLACE INTO Bestellungen VALUES (?, ?, ?, ?, ?)",
bestellungen)

# Änderungen speichern
conn.commit()
print("Daten erfolgreich eingefügt!")
```

```
<sqlite3.Cursor object at 0x0000022A4F4684C0>
<sqlite3.Cursor object at 0x0000022A4F4684C0>
<sqlite3.Cursor object at 0x0000022A4F4684C0>
Daten erfolgreich eingefügt!
```

Wichtige Punkte:

- `executemany()`: Führt den gleichen SQL-Befehl für viele Datensätze aus
- Die Fragezeichen (`?`) sind Platzhalter für die Werte
- `INSERT OR REPLACE`: Füge ein oder ersetze, falls schon vorhanden
- `commit()`: Speichert alle Änderungen in der Datenbank

Pandas und SQL

Pandas bietet einfache Möglichkeiten, mit SQL-Datenbanken zu arbeiten. Mit der Funktion `read_sql` können wir SQL-Abfragen direkt in DataFrames laden:

```python
# SQL-Abfrage mit Pandas ausführen
df = pd.read_sql('''
    SELECT k.name, p.name as produkt, p.preis, b.datum
    FROM Kunden k
    JOIN Bestellungen b ON k.kunden_id = b.kunden_id
    JOIN Produkte p ON b.produkt_id = p.produkt_id
    ORDER BY b.datum
''', conn)
print(df)
```

```
           name    produkt   preis       datum
0  Lisa Schmidt     Laptop  899.99  2025-05-15
1  Lisa Schmidt       Maus   29.99  2025-05-16
2     Max Weber    Monitor  299.99  2025-05-17
3   Anna Müller   Tastatur   79.99  2025-05-18
4   Tom Fischer     Laptop  899.99  2025-05-19
5  Lisa Schmidt  Kopfhörer  149.99  2025-05-20
```

Genauso einfach kann man Pandas DataFrames in SQL-Tabellen speichern:

```python
# Beispiel-DataFrame erstellen
verkaufsdaten = pd.DataFrame({
    'produkt': ['Monitor', 'Tastatur', 'Webcam'],
    'kategorie': ['Hardware', 'Eingabegerät', 'Peripherie'],
    'preis': [249.99, 59.99, 39.99],
    'bestand': [15, 30, 10]
})
# DataFrame in eine SQLite-Tabelle schreiben
verkaufsdaten.to_sql('Produkte', conn, if_exists='replace', index=False)
# Überprüfen ob die Daten gespeichert wurden
print(pd.read_sql("SELECT * FROM Produkte", conn))
```

```
3
    produkt     kategorie   preis  bestand
0   Monitor      Hardware  249.99       15
1  Tastatur  Eingabegerät   59.99       30
2    Webcam    Peripherie   39.99       10
```

i Datenbankverbindungen richtig verwalten

Ein wichtiger Unterschied zwischen Datenbanken und beispielsweise CSV-Dateien: Bei CSV-Dateien lesen wir die Daten einmalig ein oder schreiben sie einmalig raus. Bei Datenbanken hingegen stellen wir eine **Verbindung** her, die während unserer gesamten Arbeit bestehen bleibt.

Warum eine dauerhafte Verbindung?

- Datenbanken sind oft auf separaten Servern und benötigen Authentifizierung
- Mehrere Operationen (mehrere SQL-Abfragen) sollen über dieselbe Verbindung laufen
- Die Datenbank muss wissen, wer gerade zugreift (für Sicherheit und Parallelität)

Problem: Vergessene Verbindungen

Wenn ihr vergesst, Verbindungen zu schließen, können diese Probleme auftreten:

- Ressourcen-Verschwendung: Jede offene Verbindung verbraucht Speicher
- Verbindungslimits**: Datenbanken haben meist ein Maximum an gleichzeitigen Verbindungen
- Sperren: Manche Operationen können andere blockieren

Lösung 1: Manuell schließen

```
conn = sqlite3.connect('shop.db')
# ... arbeiten mit der Datenbank ...
conn.close()  # Wichtig: Nicht vergessen!
```

Lösung 2: Context Manager (besser!)

```
with sqlite3.connect('shop.db') as conn:
    df = pd.read_sql("SELECT * FROM Kunden", conn)
    # Verbindung wird automatisch geschlossen, auch bei Fehlern
```

Der Context Manager (`with`-Statement) ist sicherer, weil die Verbindung garantiert geschlossen wird - selbst wenn ein Fehler auftritt.

```
# Verbindung schließen
conn.close()
```

SQL vs. Pandas: Wann was verwenden?

Jetzt habt ihr beide Welten gesehen. Hier eine praktische Entscheidungshilfe:

Verwende SQL wenn:

- **Große Datenmengen** (Millionen von Zeilen): SQL ist oft 10-100x schneller
- **Einfache Aggregationen**: `GROUP BY` und `SUM` sind sehr effizient
- **Mehrere Tabellen** verknüpft werden müssen: JOINs sind SQL's Stärke
- **Mehrere Nutzer** gleichzeitig auf die Daten zugreifen
- **Daten bereits in einer Datenbank** liegen

Verwende Pandas wenn

- **Komplexe Datenmanipulationen**: Pivotierung, Reshaping, komplexe Transformationen
- **Machine Learning**: Direkte Integration mit scikit-learn
- **Explorative Datenanalyse**: Flexibles Erkunden und Visualisieren
- **Kleine bis mittlere Datensätze** (unter 1 Million Zeilen)

- **Prototyping**: Schnelle Experimente und Tests

In der Praxis kombiniert ihr oft beide Ansätze.

> Weitere Ressourcen
>
> - SQL Databases with Pandas and Python - A Complete Guide
>
> **Optional:**
>
> - SQL-Tutorial: W3Schools SQL Tutorial
> - Interaktive Übungen: SQLBolt
> - SQLite Documentation: SQLite.org

Übungen

Übung 1

In dieser Übung arbeitet ihr mit einer echten, externen Musik-Datenbank (Chinook).

Setup und Exploration (vorgegeben - einfach ausführen):

```
# SQL Übung: Chinook Musik-Datenbank Analyse
import sqlite3
import pandas as pd
import matplotlib.pyplot as plt
import urllib.request
import os

print("=== SQL ÜBUNG: CHINOOK MUSIK-DATENBANK ===\n")

# TEIL 1: SETUP (VORGEGEBEN - EINFACH AUSFÜHREN)
def setup_database():
    """Lädt die Chinook-Datenbank herunter, falls sie noch nicht existiert."""
    db_file = 'Chinook_Sqlite.sqlite'

    # Prüfen ob Datenbankdatei bereits lokal vorhanden ist
    if not os.path.exists(db_file):
        print(" Lade Chinook-Datenbank herunter...")
        # URL zur offiziellen Chinook-Datenbank auf GitHub
        url = "https://github.com/lerocha/chinook-database/releases/download/v1.4.5/
Chinook_Sqlite.sqlite"
        # Datei von URL herunterladen und lokal speichern
        urllib.request.urlretrieve(url, db_file)
        print(f" Datenbank wurde als {db_file} gespeichert.")
    else:
        print(f" Datenbank {db_file} existiert bereits.")

    return db_file

# Datenbank setup ausführen
db_file = setup_database()
# SQLite-Verbindung zur Datenbank aufbauen
conn = sqlite3.connect(db_file)

# TEIL 2: DATENBANK ERKUNDEN (VORGEGEBEN)
print("\n DATENBANKSTRUKTUR ERKUNDEN")
print("-" * 40)
```

```
# Welche Tabellen gibt es?
# Mit SELECT name wählen wir nur die Spalte 'name' aus.
# FROM sqlite_master bedeutet, dass wir aus der speziellen Systemtabelle sqlite_master
lesen,
# die Informationen über alle Objekte in der Datenbank enthält.
# WHERE type='table' filtert nur die Einträge heraus, die Tabellen sind (nicht Views
oder Indizes).
# ORDER BY name sortiert die Ergebnisse alphabetisch nach dem Tabellennamen.
tables = pd.read_sql("""
    SELECT name FROM sqlite_master WHERE type='table' ORDER BY name
""", conn)
print("Verfügbare Tabellen:")
print(tables['name'].tolist())

print("\n WICHTIGE TABELLEN FÜR DIESE ÜBUNG:")
print("• Customer: Kundendaten")
print("• Track: Musikstücke")
print("• Genre: Musikrichtungen")

# Schauen wir uns ein paar Beispieldaten an:
print("\n BEISPIELDATEN:")
print("Customer-Tabelle (erste 3 Zeilen):")
# Mit SELECT * wählen wir alle Spalten (*) aus der Tabelle Customer.
# LIMIT 3 beschränkt das Ergebnis auf nur die ersten 3 Zeilen, um die Ausgabe
übersichtlich zu halten.
print(pd.read_sql("SELECT * FROM Customer LIMIT 3", conn))

print("\n Genre-Tabelle (alle Genres):")
# Mit SELECT * FROM Genre holen wir alle Spalten und alle Zeilen aus der Genre-Tabelle.
# ORDER BY Name sortiert die Genres alphabetisch nach ihrem Namen.
print(pd.read_sql("SELECT * FROM Genre ORDER BY Name", conn))
```

Aufgaben:

1. Extrahiere eine Tabelle als DataFrame mit allen Kunden aus Deutschland.
2. Erstelle ein Balkendiagramm der Top 10 Musikrichtungen nach Anzahl Songs.

- (A) Geschafft