# SQL
# INTERVIEW QUESTIONS

Eine Sammlung von SQL-Fragen zur Vorbereitung auf technische Bewerbungsgespräche

**40**
Fragen

# INHALTSVERZEICHNIS

Frage 1

## Was ist SQL und wofür wird es verwendet?

> *SQL (Structured Query Language) ist die Standardsprache für die Interaktion mit relationalen Datenbanken. Sie wird verwendet, um Daten in Datenbanken zu definieren, zu verwalten und abzurufen. Mit SQL kannst du folgende Operationen ausführen:*

- Neue Datenbanken und Tabellen erstellen
- Neue Daten in Tabellen einfügen
- Daten mit Abfragen lesen
- Vorhandene Daten aktualisieren
- Daten löschen
- Zugriffsrechte und Berechtigungen verwalten

Frage 2

# Erkläre die Unterschiede zwischen DDL, DML und DCL in SQL

SQL-Befehle werden in drei Hauptkategorien unterteilt:

DDL (Data Definition Language) — Sprache zur Definition von Daten:

- Wird verwendet, um die Struktur der Datenbank (das Schema) zu definieren.
- Wichtigste Befehle: `CREATE`, `ALTER`, `DROP`, `TRUNCATE`, `RENAME`.

DML (Data Manipulation Language) — Sprache zur Manipulation von Daten:

- Wird verwendet, um mit Daten innerhalb der Tabellen zu arbeiten.
- Wichtigste Befehle: `SELECT`, `INSERT`, `UPDATE`, `DELETE`.

DCL (Data Control Language) — Sprache zur Steuerung des Zugriffs:

- Wird verwendet, um Zugriffsrechte auf die Datenbank zu verwalten.
- Wichtigste Befehle: `GRANT`, `REVOKE`.

Frage 3

# Was sind Primary Key und Foreign Key?

## Primary Key (PRIMARY KEY):

- Eindeutiger Identifikator eines Datensatzes in einer Tabelle.
- Erlaubt keine Duplikate und keine `NULL`-Werte.
- Kann aus einer oder mehreren Spalten bestehen (zusammengesetzter Schlüssel).

```
CREATE TABLE students (
    student_id INT PRIMARY KEY,
    name VARCHAR(100),
    age INT
);
```

## Foreign Key (FOREIGN KEY):

- Eine Spalte oder eine Gruppe von Spalten, die auf den Primary Key einer anderen Tabelle verweisen.
- Stellt die referenzielle Integrität zwischen Tabellen sicher.
- Erlaubt es, Datensätze aus verschiedenen Tabellen zu verknüpfen.

```
CREATE TABLE enrollments (
    enrollment_id INT PRIMARY KEY,
    student_id INT,
    course_id INT,
    FOREIGN KEY (student_id) REFERENCES students(student_id),
    FOREIGN KEY (course_id) REFERENCES courses(course_id)
);
```

Frage 4

# Erkläre das Konzept der Normalisierung und ihre Vorteile

> *Normalisierung ist der Prozess, die Struktur einer Datenbank so zu organisieren, dass Datenredundanz reduziert und die Datenintegrität sichergestellt wird.*

## Hauptziele der Normalisierung:

- Vermeidung von Redundanz: verhindert das Duplizieren von Daten.
- Höhere Datenintegrität: minimiert die Wahrscheinlichkeit inkonsistenter Daten.
- Einfachere Wartung und Aktualisierung: macht die Struktur der Datenbank flexibler und verständlicher.

## Wichtigste Normalformen

- Erste Normalform (1NF): alle Spalten enthalten atomare (unteilbare) Werte.
- Zweite Normalform (2NF): erfüllt 1NF und alle Nichtschlüssel-Spalten hängen vom gesamten Primary Key ab.
- Dritte Normalform (3NF): erfüllt 2NF und es gibt keine transitiven Abhängigkeiten zwischen Nichtschlüssel-Spalten.

## Beispiel

**Vor der Normalisierung:**

| StudentID | StudentName | CourseID | CourseName |
|---|---|---|---|
| 1 | John | 101 | Math |
| 1 | John | 102 | Music |

**Nach der Normalisierung:**

Tabelle Students

| StudentID | StudentName |
|---|---|
| 1 | John |

Tabelle Courses

| CourseID | CourseName |
| --- | --- |
| 101 | Math |
| 102 | Music |

Tabelle Enrollments

| StudentID | CourseID |
| --- | --- |
| 1 | 101 |
| 1 | 102 |

Frage 5

# Was ist JOIN und welche JOIN-Arten kennst du?

> *JOIN ist eine Operation in SQL, mit der du Zeilen aus zwei oder mehr Tabellen anhand verknüpfter Spalten zusammenführen kannst.*

## JOIN-Arten:

- `INNER JOIN`:

Gibt Datensätze zurück, für die in beiden Tabellen passende Einträge existieren.

- `LEFT JOIN (oder LEFT OUTER JOIN)`:

Gibt alle Datensätze aus der linken Tabelle und die passenden Datensätze aus der rechten Tabelle zurück. Gibt es keine Entsprechung, wird `NULL` für die rechte Tabelle zurückgegeben.

- `RIGHT JOIN (oder RIGHT OUTER JOIN)`:

Gibt alle Datensätze aus der rechten Tabelle und die passenden Datensätze aus der linken Tabelle zurück. Gibt es keine Entsprechung, wird `NULL` für die linke Tabelle zurückgegeben.

- `FULL OUTER JOIN`:

Gibt alle Datensätze zurück, sobald es in einer der Tabellen eine Entsprechung gibt.

- `CROSS JOIN`:

Bildet das kartesische Produkt zweier Tabellen, indem jede Zeile der ersten Tabelle mit jeder Zeile der zweiten Tabelle kombiniert wird.

Frage 6

# Was ist ein Subquery und wann wird er verwendet?

> *Ein Subquery ist eine SQL-Abfrage, die innerhalb einer anderen Abfrage verschachtelt ist. Er wird verwendet, um Operationen auszuführen, deren Ergebnis die übergeordnete Abfrage benötigt.*

## Anwendungsfälle für Subqueries:

- **Daten filtern**: Ergebnisse eines Subqueries in `WHERE` - oder `HAVING` -Bedingungen verwenden.
- **Daten auswählen**: Einen Subquery in der Liste der ausgewählten Spalten nutzen.
- **Virtuelle Tabellen erstellen**: Einen Subquery im `FROM` -Teil verwenden.

## Beispiele:

1. Subquery in `WHERE` :

```
SELECT name
FROM employees
WHERE department_id = (SELECT id FROM departments WHERE name = 'IT');
```

2. Subquery in `FROM` :

```
SELECT sub.department, COUNT(*)
FROM (
    SELECT department_id AS department
    FROM employees
) sub
GROUP BY sub.department;
```

Frage 7

## Wie entfernt man Duplikate aus dem Ergebnis einer SQL-Abfrage?

Verwende das Schlüsselwort `DISTINCT` im `SELECT`-Statement, um nur eindeutige Datensätze zurückzugeben.

Beispiel:

```
SELECT DISTINCT position
FROM employees;
```

Frage 8

# Erkläre den Unterschied zwischen WHERE und HAVING

## WHERE

- Filtert Zeilen **vor** der Gruppierung der Daten.
- Kann keine Aggregatfunktionen ( `SUM()` , `COUNT()` , `AVG()` usw.) verwenden.
- Wirkt auf einzelne Datensätze der Tabelle.

Beispiel:

```
SELECT department_id, COUNT()
FROM employees
WHERE salary > 50000
GROUP BY department_id;
```

## *HAVING*

- *Filtert Zeilengruppen nach der Gruppierung der Daten.*
- *Kann Aggregatfunktionen verwenden.*
- *Wirkt auf die Ergebnisse von* `GROUP BY` .

*Beispiel:*

```
SELECT department_id, COUNT() AS num_employees
FROM employees
GROUP BY department_id
HAVING COUNT(*) > 5;
```

Frage 9

# Was ist ein Index und wie beeinflusst er die Performance?

> *Ein **Index** ist eine spezielle Datenstruktur, die das Lesen von Daten aus einer Tabelle beschleunigt, indem sie Verweise auf die Daten anlegt.*

Indizes beschleunigen Leseoperationen, können aber Schreiboperationen (Insert, Update, Delete) verlangsamen, da die Indizes bei jeder Datenänderung mit aktualisiert werden müssen.

## Vorteile von Indizes:

- Schnellere Ausführung von `SELECT`-Operationen.
- Bessere Performance beim Sortieren und Suchen von Daten.

## Nachteile von Indizes:

- Zusätzlicher Speicherplatz auf der Platte.
- Langsamere `INSERT`-, `UPDATE`- und `DELETE`-Operationen.

## Beispiel für das Anlegen eines Index:

```
-- Index auf der Spalte name in der Tabelle employees anlegen
CREATE INDEX idx_employees_name ON employees(name);
```

Frage 10

# Was sind die wichtigsten Unterschiede zwischen DELETE und TRUNCATE?

## DELETE:

- Löscht ausgewählte Datensätze aus einer Tabelle.
- Du kannst eine `WHERE` -Bedingung verwenden, um gezielt Datensätze zu entfernen.
- Die Operation wird zeilenweise im Transaktionslog aufgezeichnet.
- `ON DELETE` -Trigger werden ausgelöst.
- Langsamer als `TRUNCATE` .

Beispiel:

```
DELETE FROM employees WHERE salary < 30000;
```

## TRUNCATE:

- Löscht alle Datensätze aus der Tabelle, ohne Möglichkeit zur Wiederherstellung per `ROLLBACK` (in den meisten DBMS).
- `WHERE` kann nicht verwendet werden.
- Die Operation ist schneller, da sie nicht zeilenweise protokolliert wird.
- Setzt Identifier zurück (falls Autoincrement verwendet wird).
- Trigger werden nicht ausgelöst.

Beispiel:

```
TRUNCATE TABLE employees;
```

Frage 11

# Was ist eine Transaktion und welche Eigenschaften hat sie (ACID)?

> *Eine **Transaktion** ist eine Folge von Operationen, die als eine logische Einheit ausgeführt werden — entweder vollständig oder gar nicht.*

## Eigenschaften von Transaktionen (ACID):

**Atomarität (Atomicity):**

- Eine Transaktion wird entweder vollständig ausgeführt oder gar nicht.
- Tritt ein Fehler auf, werden alle Änderungen rückgängig gemacht.

**Konsistenz (Consistency):**

- Eine Transaktion überführt die Datenbank von einem konsistenten Zustand in einen anderen.
- Alle Regeln und Constraints der Datenbank bleiben dabei eingehalten.

**Isolation (Isolation):**

- Die Ergebnisse einer Transaktion sind für andere Transaktionen erst nach deren Abschluss sichtbar.
- So wird die gegenseitige Beeinflussung paralleler Transaktionen verhindert.

**Dauerhaftigkeit (Durability):**

- Nach erfolgreichem Abschluss bleiben die Ergebnisse einer Transaktion auch bei Systemausfällen erhalten.
- Änderungen werden in einen persistenten Speicher geschrieben.

Frage 12

# Was sind Trigger in SQL?

> *Ein **Trigger** ist eine gespeicherte Prozedur, die automatisch ausgeführt wird, sobald ein bestimmtes Ereignis in der Datenbank eintritt — etwa `INSERT`, `UPDATE` oder `DELETE` auf einer bestimmten Tabelle.*

## Trigger-Arten:

- DML-Trigger: reagieren auf `INSERT`-, `UPDATE`- und `DELETE`-Operationen.
- DDL-Trigger: reagieren auf `CREATE`-, `ALTER`- und `DROP`-Operationen.
- Trigger auf Zeilen- oder Statement-Ebene.

## Vorteile von Triggern:

- Automatisierung von Prüfungen und Constraints.
- Logging von Änderungen.
- Sicherung der Datenintegrität.

## Beispiel für einen Trigger:

```
CREATE TRIGGER trg_after_insert_employee
AFTER INSERT ON employees
FOR EACH ROW
BEGIN
    INSERT INTO audit_log (employee_id, action, action_time)
    VALUES (NEW.id, 'INSERT', NOW());
END;
```

Frage 13

# Erkläre, was ein VIEW ist und welche Vorteile er bietet

> *Ein **VIEW (Sicht)** ist eine virtuelle Tabelle, die auf dem Ergebnis einer SQL-Abfrage basiert. Ein View speichert selbst keine Daten, sondern stellt eine bestimmte Sicht auf die Daten einer oder mehrerer Tabellen bereit.*

## Vorteile eines VIEW:

- **Vereinfachung komplexer Abfragen**: Du kannst eine komplexe Abfrage speichern und wie eine einfache Tabelle nutzen.
- **Sicherheit**: Du gibst Nutzern nur Zugriff auf bestimmte Daten und blendest den Rest aus.
- **Aktualisierbarkeit**: In manchen Fällen lassen sich Daten direkt über den View aktualisieren.
- **Datenintegrität**: Ein View kann Daten aus mehreren Tabellen auf eine bestimmte Art zusammenführen.

## Beispiel: VIEW anlegen:

```
CREATE VIEW employee_details AS
SELECT e.id, e.name, d.name AS department, e.salary
FROM employees e
JOIN departments d ON e.department_id = d.id;
```

## VIEW verwenden:

```
SELECT * FROM employee_details WHERE salary > 50000;
```

Frage 14

# Wie verwendet man den LIKE-Operator und wofür dient er?

Der `LIKE`-Operator wird in `WHERE`-Bedingungen verwendet, um Strings zu finden, die einem bestimmten Muster entsprechen. In den Mustern kommen Platzhalter zum Einsatz:

- `%` — entspricht einer beliebigen Zeichenfolge (auch der Länge null).
- `_` — entspricht einem einzelnen beliebigen Zeichen.

## Anwendungsbeispiele:

1. Strings finden, die mit „A“ beginnen:

```
SELECT  FROM employees WHERE name LIKE 'A%';
```

*2. Strings finden, die mit „e“ enden:*

```
SELECT  FROM employees WHERE name LIKE '%e';
```

3. Strings finden, bei denen das zweite Zeichen „a“ ist:

```
SELECT * FROM employees WHERE name LIKE '_a%';
```

Frage 15

# Was sind Aggregatfunktionen? Nenne Beispiele

Aggregatfunktionen führen Berechnungen über eine Menge von Werten aus und geben einen einzelnen Wert zurück. Sie werden oft in Kombination mit `GROUP BY` verwendet.

## Wichtigste Aggregatfunktionen

- `COUNT()` — zählt die Anzahl der Zeilen.
- `SUM()` — berechnet die Summe der Werte.
- `AVG()` — berechnet den Durchschnittswert.
- `MAX()` — findet den größten Wert.
- `MIN()` — findet den kleinsten Wert.

## Anwendungsbeispiele

1. Anzahl der Mitarbeiter zählen:

```
SELECT COUNT(*) FROM employees;
```

2. Durchschnittsgehalt pro Abteilung:

```
SELECT department_id, AVG(salary) AS average_salary
FROM employees
GROUP BY department_id;
```

3. Höchstes Gehalt im Unternehmen:

```
SELECT MAX(salary) FROM employees;
```

4. Gesamtumsatz pro Monat:

```
SELECT SUM(amount) FROM sales WHERE sale_date BETWEEN '2023-01-01' AND '2023-01-31';
```

Frage 16

# Erkläre den Unterschied zwischen UNION und UNION ALL

## UNION:

- Führt die Ergebnisse von zwei oder mehr `SELECT`-Abfragen zusammen.
- Entfernt Duplikate aus dem kombinierten Ergebnis.

Syntax

```
SELECT column_list FROM table1
UNION
SELECT column_list FROM table2;
```

## UNION ALL:

- Führt die Ergebnisse von zwei oder mehr `SELECT`-Abfragen zusammen.
- Behält Duplikate im kombinierten Ergebnis.
- Schneller, da keine zusätzliche Duplikatentfernung erfolgt.

Syntax

```
SELECT column_list FROM table1
UNION ALL
SELECT column_list FROM table2;
```

Frage 17

# Was ist eine Stored Procedure und wie unterscheidet sie sich von einer Funktion?

## Stored Procedure:

- Eine Menge von SQL-Befehlen, die auf dem Server zur Wiederverwendung gespeichert werden.
- Kann `SELECT`-, `INSERT`-, `UPDATE`- und `DELETE`-Operationen ausführen.
- Kann mehrere Ergebnismengen zurückgeben oder gar nichts.
- Kann Eingangs- und Ausgangsparameter haben.
- Kann nicht innerhalb einer SQL-Abfrage aufgerufen werden.

Beispiel für eine Stored Procedure:

```
CREATE PROCEDURE GetEmployeeByID(IN emp_id INT)
BEGIN
    SELECT * FROM employees WHERE id = emp_id;
END;
```

Aufruf der Prozedur:

```
CALL GetEmployeeByID(1);
```

## Funktion (Function):

- Gibt einen einzelnen Wert (skalare Funktion) oder eine Tabelle (Tabellenfunktion) zurück.
- Kann in SQL-Ausdrücken verwendet werden (z. B. in SELECT oder WHERE).
- Muss einen Wert zurückgeben.
- Wird üblicherweise für Berechnungen verwendet und liefert ein deterministisches Ergebnis.

Beispiel für eine Funktion:

```
CREATE FUNCTION GetEmployeeSalary(emp_id INT) RETURNS DECIMAL(10,2)
BEGIN
    DECLARE salary DECIMAL(10,2);
    SELECT e.salary INTO salary FROM employees e WHERE e.id = emp_id;
    RETURN salary;
END;
```

Verwendung der Funktion:

```
SELECT name, GetEmployeeSalary(id) FROM employees;
```

Frage 18

# Wie optimiert man die Performance von SQL-Abfragen?

**Indizes einsetzen:**

- Lege Indizes auf Spalten an, die häufig in `WHERE`, `JOIN` und `ORDER BY` vorkommen.
- Vermeide überflüssige Indizes.

* `SELECT` vermeiden:

- **Wähle nur die Spalten aus, die du wirklich brauchst.**
- **Das reduziert die übertragene Datenmenge.**

Bedingungen in `JOIN` und `WHERE` optimieren:

- **Verwende wo möglich Gleichheit (=) statt Ungleichheit.**
- **Vermeide Funktionen und Berechnungen auf indexierten Spalten innerhalb der Bedingungen.**

Ergebnisse begrenzen ( `LIMIT` ):

- **Begrenze die Anzahl der zurückgegebenen Zeilen, wenn du nicht alle Daten brauchst.**

Subqueries vermeiden, wenn ein Join möglich ist:

- **Ersetze korrelierte Subqueries durch `JOIN` oder `EXISTS`.**

Häufig genutzte Daten cachen:

- **Nutze materialisierte Views oder Caching auf Anwendungsebene.**

Abfragen profilen und analysieren:**

- Verwende Tools ( `EXPLAIN`, `EXPLAIN PLAN` ), um den Ausführungsplan der Abfragen zu analysieren.

Frage 19

## Was sind Constraints und welche Arten gibt es?

**Constraints** sichern die Integrität und Zuverlässigkeit der Daten in einer Tabelle, indem sie Regeln für die Werte einzelner Spalten festlegen.

Arten von Constraints:

**NOT NULL:**

- Verbietet `NULL`-Werte in der Spalte.

Beispiel:

```
CREATE TABLE products (
    product_id INT PRIMARY KEY,
    name VARCHAR(100) NOT NULL
);
```

**UNIQUE:**

- Stellt sicher, dass die Werte in einer Spalte oder einer Gruppe von Spalten eindeutig sind.

Beispiel:

```
CREATE TABLE users (
    user_id INT PRIMARY KEY,
    email VARCHAR(100) UNIQUE
);
```

**PRIMARY KEY:**

- Kombination aus `NOT NULL` und `UNIQUE`.
- Identifiziert jeden Datensatz in der Tabelle eindeutig.

**FOREIGN KEY:**

- Stellt die referenzielle Integrität zwischen Tabellen sicher.
- Der Wert muss einem existierenden Wert des Primary Keys in der referenzierten Tabelle entsprechen.

Beispiel:

```
CREATE TABLE orders (
    order_id INT PRIMARY KEY,
    user_id INT,
    FOREIGN KEY (user_id) REFERENCES users(user_id)
);
```

**CHECK:**

- Definiert eine Bedingung, die die Werte in einer Spalte erfüllen müssen.

Beispiel:

```
CREATE TABLE employees (
    id INT PRIMARY KEY,
    age INT CHECK (age >= 18)
);
```

**DEFAULT:**

- Legt einen Standardwert für eine Spalte fest, wenn beim Einfügen kein Wert angegeben wird.

```
CREATE TABLE tasks (
    task_id INT PRIMARY KEY,
    status VARCHAR(20) DEFAULT 'Pending'
);
```

Frage 20

# Was ist eine SQL-Injection und wie schützt man sich davor?

> *Eine **SQL-Injection** ist eine Angriffsmethode auf eine Datenbank, bei der ein Angreifer über Eingabefelder schädlichen SQL-Code einschleust und so nicht autorisierte SQL-Abfragen ausführen kann.*

## Folgen einer SQL-Injection:

- Diebstahl von Daten.
- Löschen oder Verändern von Daten.
- Erlangen administrativer Zugriffsrechte.

## Schutz vor SQL-Injections:

### 1. Prepared Statements (parametrisierte Abfragen):

- Verwende Parameter statt String-Konkatenation.
- Das DBMS escapt Sonderzeichen automatisch.

Beispiel (in Java mit JDBC)

```
String sql = "SELECT * FROM users WHERE username = ? AND password = ?";
PreparedStatement stmt = connection.prepareStatement(sql);
stmt.setString(1, username);
stmt.setString(2, password);
ResultSet rs = stmt.executeQuery();
```

### 2. ORM (Object-Relational Mapping) einsetzen:

- ORM-Bibliotheken bringen häufig schon Schutzmechanismen gegen SQL-Injections mit.

### 3. Eingabedaten prüfen und filtern:

- Prüfe Daten auf das erwartete Format.
- Setze Validierung auf Server- und Client-Seite ein.

### 4. Zugriffsrechte einschränken:

- Vergib Datenbanknutzern nur die minimal nötigen Rechte.
- Beschränke den Zugriff auf Systemtabellen und sensible Operationen.

### 5. Stored Procedures verwenden:

- Die Datenlogik ist in der Prozedur gekapselt.

- Nutzer greifen nur auf die Prozeduren zu, nicht direkt auf die Tabellen.

Frage 21

# Was ist eine relationale Datenbank?

> ***Eine relationale Datenbank*** *ist eine Datenbank, die auf dem relationalen Datenmodell basiert. In einer solchen Datenbank werden die Daten in Tabellen gespeichert, und die Beziehungen zwischen ihnen werden über Schlüssel definiert.*

## Wichtigste Eigenschaften:

- Tabellen (Relationen): Die Daten sind in Tabellen organisiert, die aus Zeilen und Spalten bestehen.
- Zeilen (Datensätze): Jede Zeile steht für einen einzelnen Datensatz.
- Spalten (Attribute): Jede Spalte enthält Daten eines bestimmten Typs.
- Schlüssel: dienen dazu, Datensätze eindeutig zu identifizieren und Beziehungen zwischen Tabellen herzustellen.

## Vorteile relationaler Datenbanken:

- Flexibilität: Neue Tabellen und Spalten lassen sich leicht ergänzen.
- Datenintegrität: Constraints sorgen dafür, dass die Daten konsistent bleiben.
- SQL: eine standardisierte Sprache für die Datenverwaltung.

Frage 22

# Erkläre den Unterschied zwischen INNER JOIN und OUTER JOIN

## INNER JOIN:

- Liefert nur die Datensätze, für die es in beiden verknüpften Tabellen einen Treffer gibt.
- Gibt es keinen Treffer, taucht der Datensatz nicht im Ergebnis auf.

Beispiel:

```
SELECT
FROM employees e
INNER JOIN departments d ON e.department_id = d.id;
```

## *OUTER JOIN:*

- *Liefert sowohl Datensätze mit Treffer als auch Datensätze aus einer der Tabellen, für die es keinen passenden Eintrag gibt.*

### *Arten von OUTER JOIN:*

#### *LEFT OUTER JOIN (LEFT JOIN):*

- *Liefert alle Datensätze aus der linken Tabelle und die passenden Datensätze aus der rechten Tabelle.*
- *Gibt es keinen Treffer, sind die Spalten der rechten Tabelle* `NULL`.

*Beispiel:*

```
SELECT
FROM employees e
LEFT JOIN departments d ON e.department_id = d.id;
```

#### RIGHT OUTER JOIN (RIGHT JOIN):

- Liefert alle Datensätze aus der rechten Tabelle und die passenden Datensätze aus der linken Tabelle.
- Gibt es keinen Treffer, sind die Spalten der linken Tabelle `NULL`.

Beispiel:

```
SELECT
FROM employees e
RIGHT JOIN departments d ON e.department_id = d.id;
```

***FULL OUTER JOIN (FULL JOIN):***

- *Liefert alle Datensätze, sobald es in einer der Tabellen einen Treffer gibt.*
- *Fehlt der Treffer auf der jeweils anderen Seite, sind die entsprechenden Spalten* `NULL`.

```
SELECT
FROM employees e
FULL OUTER JOIN departments d ON e.department_id = d.id;
```

Frage 23

# Was ist NULL und wie arbeitet man damit in SQL?

> ***NULL** ist ein spezieller Wert in SQL, der einen fehlenden oder unbekannten Wert kennzeichnet.*

## Besonderheiten von NULL:

- `NULL` ist nicht dasselbe wie ein leerer String oder die Zahl Null.
- Operationen mit `NULL` liefern `NULL`.
- Der Vergleich `NULL` = `NULL` ergibt `FALSE`.

## Mit NULL arbeiten

Um auf `NULL` zu prüfen, nutzt du den Operator `IS NULL` oder `IS NOT NULL`.

```
-- Datensätze mit unbekanntem Geburtsdatum finden
SELECT  FROM employees WHERE birth_date IS NULL;

-- Datensätze mit bekanntem Geburtsdatum finden
SELECT  FROM employees WHERE birth_date IS NOT NULL;
```

## Funktionen für die Arbeit mit NULL

**COALESCE** Liefert das erste Element der Liste, das nicht `NULL` ist.

```
COALESCE(val1[, val2, ...., val_n])
```

**ISNULL** Liefert 1 oder 0, je nachdem, ob der Ausdruck `NULL` ist.

```
ISNULL(value)
```

**IFNULL** Liefert den ersten Wert, sofern dieser nicht `NULL` ist. Andernfalls wird der zweite Wert zurückgegeben.

```
IFNULL(value, alternative_value)
```

Frage 24

# Wie verwendet man den CASE-Operator in SQL?

Der Operator `CASE` ist das Mittel der Wahl, um bedingte Logik direkt in einer SQL-Abfrage abzubilden. Damit gibst du Werte basierend auf Bedingungen zurück, ganz ähnlich wie mit `IF-ELSE`.

```
CASE
    WHEN condition1 THEN result1
    WHEN condition2 THEN result2
    ...
    ELSE default_result
END
```

## Beispiele

- Kategorien anhand des Gehalts vergeben:

```
SELECT name,
    salary,
    CASE
        WHEN salary >= 80000 THEN 'Hoch'
        WHEN salary >= 50000 THEN 'Mittel'
        ELSE 'Niedrig'
    END AS salary_category
FROM employees;
```

Frage 25

# Erkläre die Transaktionsbefehle COMMIT und ROLLBACK.

## COMMIT

- Schreibt die aktuelle Transaktion endgültig fest.
- Alle Änderungen aus der Transaktion werden persistent und für andere Nutzer sichtbar.
- Nach einem COMMIT lassen sich die Änderungen nicht mehr rückgängig machen.

```
BEGIN TRANSACTION;

UPDATE accounts SET balance = balance - 100 WHERE account_id = 1;
UPDATE accounts SET balance = balance + 100 WHERE account_id = 2;

COMMIT;
```

## ROLLBACK

- Bricht die aktuelle Transaktion ab.
- Alle in der Transaktion vorgenommenen Änderungen werden zurückgerollt.
- Die Datenbank kehrt in den Zustand vor Beginn der Transaktion zurück.

```
BEGIN TRANSACTION;

DELETE FROM orders WHERE order_date < '2022-01-01';

-- Falls du es dir anders überlegt hast
ROLLBACK;
```

## Einsatz beim Transaktionsmanagement:

- `BEGIN TRANSACTION` oder `START TRANSACTION` : Beginn einer Transaktion.
- `COMMIT` : Festschreiben der Transaktion.
- `ROLLBACK` : Zurückrollen der Transaktion.

Frage 26

# Erkläre die Unterschiede zwischen CHAR und VARCHAR

## CHAR(n):

- Speichert Strings mit fester Länge `n`.
- Ist die eingegebene Zeichenkette kürzer als `n`, wird sie mit Leerzeichen auf `n` aufgefüllt.
- Geeignet für Daten gleicher Länge (z. B. Ländercodes, Postleitzahlen).

## VARCHAR(n):

- Speichert Strings variabler Länge mit bis zu `n` Zeichen.
- Belegt tatsächlich nur so viel Platz, wie der String Zeichen hat, plus einen kleinen Overhead für die Längenangabe.
- Geeignet für Daten mit variabler Länge.

## Wichtigste Unterschiede:

Speicher und Performance:

- `CHAR` belegt immer eine feste Menge an Speicher.
- `VARCHAR` ist beim Speicherverbrauch effizienter, kann aber beim Zugriff einen Hauch langsamer sein.

Einsatzgebiete:

- `CHAR` ist gut für Daten mit vorhersagbarer Länge geeignet.
- `VARCHAR` ist gut für Daten mit variabler Länge geeignet.

Beispiel:

```
CREATE TABLE products (
    code CHAR(10),     -- Produktcode mit fester Länge
    name VARCHAR(100)  -- Produktname mit variabler Länge
);
```

Daten einfügen:

```
INSERT INTO products (code, name)
VALUES ('A123', 'Lenovo Notebook');
```

In der Spalte `code` wird der Wert mit Leerzeichen auf 10 Zeichen aufgefüllt.

Frage 27

# Was ist eine temporäre Tabelle in SQL?

> ***Eine temporäre Tabelle*** *ist eine Tabelle, die nur innerhalb der aktuellen Session bzw. Verbindung existiert und automatisch gelöscht wird, sobald die Session endet oder die Verbindung geschlossen wird.*

## Eine temporäre Tabelle anlegen

```
CREATE TEMPORARY TABLE TempTable (
    id INT,
    name VARCHAR(100)
);
```

### Mit einer temporären Tabelle arbeiten

```
-- Daten in die temporäre Tabelle einfügen
INSERT INTO #TempTable (id, name)
VALUES (1, 'Iwan'), (2, 'Peter');

-- Daten aus der temporären Tabelle lesen
SELECT * FROM #TempTable;

-- Die temporäre Tabelle wird nach dem Ende der Session automatisch entfernt
```

## Einsatzbereiche temporärer Tabellen

- Zwischenergebnisse in komplexen Queries ablegen.
- Große Datenmengen in Batch-Operationen verarbeiten.
- Konflikte vermeiden, wenn mehrere Benutzer gleichzeitig arbeiten.

Frage 28

# Was sind Window-Funktionen in SQL?

> ***Window-Funktionen*** *sind Funktionen, die eine Berechnung über eine Menge von Zeilen (das "Fenster") ausführen, die mit der aktuellen Zeile zusammenhängen, und für jede Zeile ein Ergebnis liefern, ohne die Daten zu gruppieren.*

## Wichtige Window-Funktionen:

### Aggregatfunktionen:

- `SUM` — berechnet die Gesamtsumme der Werte
- `COUNT` — zählt die Anzahl der Einträge in einer Spalte
- `AVG` — berechnet den arithmetischen Mittelwert
- `MAX` — findet den größten Wert
- `MIN` — findet den kleinsten Wert

### Ranking-Funktionen:

- `ROW_NUMBER` : vergibt eine fortlaufende Nummer pro Zeile innerhalb des Fensters
- `RANK` : vergibt einen Rang innerhalb des Fensters und überspringt Ränge bei gleichen Werten
- `DENSE_RANK` : vergibt einen Rang ohne Lücken

### Versatz-Funktionen:

- `LAG` : liefert den Wert aus der vorherigen Zeile
- `LEAD` : liefert den Wert aus der nächsten Zeile
- `FIRST_VALUE` : liefert den ersten Wert im Fenster
- `LAST_VALUE` : liefert den letzten Wert im Fenster

Eine ausführliche Erklärung zur Funktionsweise von Window-Funktionen findest du in unserem Kurs.

Frage 29

# Erkläre das Konzept der CTE (Common Table Expression)

> ***Eine CTE (Common Table Expression)*** *ist eine temporäre, benannte Ergebnismenge, die innerhalb einer SQL-Query mit dem Schlüsselwort* `WITH` *definiert wird.*

Sie verbessert die Lesbarkeit und die Struktur komplexer Queries.

## Syntax

```
WITH CTEName (column1, column2, ...)
AS (
    -- Deine Query
    SELECT ...
)
SELECT * FROM CTEName;
```

## Vorteile von CTEs

- Verbessert die Lesbarkeit des Codes.
- Erlaubt es, komplexe Queries in logische Teile zu zerlegen.
- Unterstützt rekursive Queries (rekursive CTEs).

Eine ausführliche Erklärung zur Funktionsweise von CTEs findest du in unserem Kurs.

Frage 30

## Wie löscht man eine Tabelle samt ihrer Daten?

Dafür gibt es den Befehl `DROP TABLE`. Er entfernt die Tabelle samt aller darin gespeicherten Daten aus der Datenbank.

### Syntax

```
DROP TABLE table_name;
```

### Wichtige Hinweise:

- Alle Daten, Indizes, Trigger und Berechtigungen, die zur Tabelle gehören, werden entfernt.
- Die Aktion ist unumkehrbar, sofern kein Backup oder ein Wiederherstellungsmechanismus eingerichtet ist.

Frage 31

# Was ist ein FOREIGN KEY und wie sichert er die Datenintegrität?

> ***Ein FOREIGN KEY*** *ist ein Constraint, das eine Beziehung zwischen einer Spalte oder einer Spaltengruppe einer Tabelle und einer Spalte (bzw. Spalten) einer anderen Tabelle herstellt — in der Regel zum Primary Key der anderen Tabelle.*

Er sorgt für referenzielle Integrität, indem er sicherstellt, dass die Werte in der Fremdschlüssel-Spalte zu vorhandenen Werten in der referenzierten Tabelle passen.

## So wird die Datenintegrität gewahrt:

- **Einfügen ungültiger Daten wird verhindert.**

Es ist nicht möglich, einen Wert in die Fremdschlüssel-Spalte einzutragen, wenn es diesen Wert in der referenzierten Tabelle nicht gibt.

- **Löschen verknüpfter Datensätze wird eingeschränkt.**

Ein Datensatz in der übergeordneten Tabelle kann nicht einfach gelöscht werden, solange ihn Datensätze der untergeordneten Tabelle referenzieren — dafür braucht es zusätzliche Regeln.

**Beispiel für einen Foreign Key:**

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    Name VARCHAR(100),
    DepartmentID INT,
    FOREIGN KEY (DepartmentID) REFERENCES Departments(DepartmentID)
);
```

Frage 32

# Wie fügt man einer bestehenden Tabelle eine neue Spalte hinzu?

Dafür nutzt du den Befehl `ALTER TABLE` zusammen mit dem Operator `ADD`, um der Tabelle eine neue Spalte hinzuzufügen.

## Syntax

```
ALTER TABLE table_name
ADD column_name data_type [constraints];
```

## Beispiel

Angenommen, es gibt eine Tabelle `employees`, und du möchtest die Spalte `email` vom Typ `VARCHAR(255)` ergänzen.

```
ALTER TABLE employees
ADD email VARCHAR(255);
```

Spalte mit `NOT NULL`-Constraint und Default-Wert hinzufügen:

```
ALTER TABLE employees
ADD date_of_birth DATE NOT NULL DEFAULT '1900-01-01';
```

**Wichtig:** Wenn du eine Spalte mit `NOT NULL`-Constraint ergänzt und die Tabelle bereits Daten enthält, musst du einen Default-Wert angeben, sonst bekommst du einen Fehler.

Frage 33

# Was ist eine korrelierte Subquery?

> ***Eine korrelierte Subquery*** *ist eine Subquery, die von der äußeren Query abhängt. Sie wird für jede Zeile der äußeren Query erneut ausgeführt und nutzt dabei die Werte aus dieser Zeile.*

## Besonderheiten

- Die Subquery referenziert Spalten aus der äußeren Query.
- Sie kann durch die wiederholte Ausführung weniger performant sein.

## Beispiel

Es gibt die Tabellen `employees` und `departments`.

```
SELECT e.name, e.salary
FROM employees e
WHERE e.salary > (
    SELECT AVG(salary)
    FROM employees
    WHERE department_id = e.department_id
);
```

- Die Subquery berechnet das Durchschnittsgehalt der Abteilung jedes Mitarbeitenden.
- Die Hauptquery liefert die Mitarbeitenden, deren Gehalt über dem Durchschnitt ihrer Abteilung liegt.

Mehr Infos zu korrelierten Subqueries findest du in unserem Kurs.

Frage 34

# Erkläre die Unterschiede zwischen DELETE, TRUNCATE und DROP

## DELETE

- Entfernt ausgewählte Datensätze aus einer Tabelle.
- Mit `WHERE` kannst du gezielt einzelne Datensätze löschen.
- Die Operation wird zeilenweise im Transaktionslog protokolliert.
- `ON DELETE`-Trigger werden ausgelöst.

### Syntax

```
DELETE FROM table_name [WHERE condition];
```

## TRUNCATE

- Entfernt sämtliche Daten aus einer Tabelle, ohne Möglichkeit zur Wiederherstellung.
- `WHERE` ist hier nicht erlaubt.
- Deutlich schneller, da nicht jede gelöschte Zeile geloggt wird.
- Setzt Auto-Increment-Werte zurück.
- Trigger werden nicht ausgelöst.

### Syntax

```
TRUNCATE TABLE table_name;
```

## DROP

- Entfernt die komplette Tabelle samt Daten, Struktur, Indizes und Constraints.
- Die Aktion ist unumkehrbar.

### Syntax

```
DROP TABLE table_name;
```

## Wann welcher Befehl?

- `DELETE` nutzt du, wenn du nur bestimmte Datensätze entfernen willst.

- `TRUNCATE` ist die schnelle Variante, um alle Daten aus einer Tabelle zu entfernen, die Struktur aber zu behalten.
- `DROP` setzt du ein, wenn die Tabelle komplett aus der Datenbank verschwinden soll.

Frage 35

# Was ist ein Self-Join und wann wird er eingesetzt?

> ***Ein Self-Join*** *ist ein Join-Typ in SQL, bei dem eine Tabelle mit sich selbst verknüpft wird. Das ist nützlich, wenn du Zeilen derselben Tabelle miteinander vergleichen oder hierarchische Daten verarbeiten möchtest.*

## Wann ein Self-Join Sinn ergibt:

- **Hierarchische Strukturen.**

Etwa eine Mitarbeitertabelle, in der jeder Mitarbeitende eine Managerin oder einen Manager haben kann, die selbst wieder ein Eintrag in derselben Tabelle sind.

- **Datensätze vergleichen.**

Zum Aufspüren von Duplikaten oder zum Vergleichen von Werten zwischen Zeilen derselben Tabelle.

## Beispiel

| employeeId | name | managerId |
|---|---|---|
| 1 | John | |
| 2 | Michail | 1 |
| 3 | Alisa | 1 |
| 4 | Max | 2 |

Um eine Liste der Mitarbeitenden samt zugehörigem Manager zu erhalten:

```
SELECT e.name AS Employee, m.name AS Manager
FROM Employee e
LEFT JOIN Employee m ON e.managerId = m.employeeId;
```

In dieser Query verknüpfen wir die Tabelle `Employee` mit sich selbst, um jedem Mitarbeitenden den passenden Manager zuzuordnen.

Frage 36

# Wie führst du Backup und Restore einer Datenbank durch?

Backup und Restore einer Datenbank sind kritische Aufgaben, um Daten zu sichern und sie im Fall eines Ausfalls oder Datenverlusts wiederherstellen zu können.

## So legst du ein Backup einer Datenbank an:

Die Backup-Methode hängt vom eingesetzten Datenbankmanagementsystem (DBMS) ab.

Hier ein paar Beispiele für gängige Systeme.

### MySQL

Backup mit dem Tool `mysqldump`:

```
mysqldump -u username -p mydatabase > backup.sql
```

- `username` — der Benutzername der Datenbank.
- `mydatabase` — der Name der zu sichernden Datenbank.
- `backup.sql` — die Datei, in die das Backup geschrieben wird.

### PostgreSQL

Backup mit dem Tool `pg_dump`:

```
pg_dump -U username mydatabase > backup.sql
```

- `-U username` — der Benutzername der Datenbank.
- `mydatabase` — der Name der Datenbank.
- `backup.sql` — die Ausgabedatei des Backups.

## So stellst du eine Datenbank aus einem Backup wieder her:

### MySQL

Restore aus der Datei `backup.sql`:

```
mysql -u username -p mydatabase < backup.sql
```

### PostgreSQL

Restore mit dem Tool `psql`:

```
psql -U username mydatabase < backup.sql
```

## Empfehlungen

- **Berechtigungen.**

Stelle sicher, dass du die nötigen Rechte für Backup- und Restore-Operationen hast.

- **Regelmäßigkeit.**

Richte automatisierte, regelmäßige Backups ein, um das Risiko eines Datenverlusts zu minimieren.

- **Aufbewahrung.**

Lagere Backups an einem sicheren und zuverlässigen Ort, am besten außerhalb des Hauptservers.

- **Tests.**

Spiele Backups regelmäßig auf einem Testserver ein, um sicherzustellen, dass sie konsistent und nutzbar sind.

Frage 37

# Wie setzt man eine Many-to-Many-Beziehung in SQL um?

**Many-to-Many**-Beziehungen entstehen in relationalen Datenbanken, wenn ein Datensatz aus der ersten Tabelle zu mehreren Datensätzen der zweiten Tabelle gehören kann — und umgekehrt.

In SQL setzt du solche Beziehungen über eine Zwischentabelle um (auch Verknüpfungs- oder Junction-Tabelle genannt), die beide Haupttabellen über Foreign Keys verbindet.

## So setzt du eine Many-to-Many-Beziehung in SQL um

- **Lege die beiden Haupttabellen an**, die miteinander verknüpft werden sollen.
- **Erstelle eine Zwischentabelle** mit Foreign Keys auf die Primary Keys beider Haupttabellen.
- **Definiere die Foreign Keys und einen zusammengesetzten Primary Key** in der Zwischentabelle, um referenzielle Integrität und eindeutige Beziehungspaare sicherzustellen.

## Beispielumsetzung

Stell dir ein Szenario mit den Tabellen `Student` und `Course` vor: ein Student kann sich für mehrere Kurse anmelden, und ein Kurs kann von mehreren Studierenden besucht werden.

- Studierenden-Tabelle anlegen:

```
CREATE TABLE Student (
    StudentID INT PRIMARY KEY,
    Name VARCHAR(100)
);
```

- Kurs-Tabelle anlegen:

```
CREATE TABLE Course (
    CourseID INT PRIMARY KEY,
    Title VARCHAR(100)
);
```

- Zwischentabelle für die Many-to-Many-Beziehung anlegen:

```
CREATE TABLE StudentCourse (
    StudentID INT,
    CourseID INT,
    PRIMARY KEY (StudentID, CourseID),
    FOREIGN KEY (StudentID) REFERENCES Student(StudentID),
    FOREIGN KEY (CourseID) REFERENCES Course(CourseID)
);
```

Ergebnis:

- Die **Zwischentabelle** `StudentCourse` hält Paare aus `StudentID` und `CourseID`, die jeweils eine Verknüpfung zwischen Studierenden und Kursen abbilden.
- Der **zusammengesetzte Primary Key** (`StudentID`, `CourseID`) sorgt dafür, dass jedes Paar einzigartig bleibt und Duplikate gar nicht erst entstehen.
- Die **Foreign Keys** sichern die Datenintegrität, indem sie auf die passenden Einträge in den Tabellen `Student` und `Course` verweisen.

Frage 38

# Wie funktionieren die Befehle REVOKE und GRANT?

Die Befehle `GRANT` und `REVOKE` dienen in SQL dazu, die Zugriffsrechte von Benutzern auf Datenbankobjekte zu steuern. Damit erteilst oder entziehst du Benutzern oder Rollen bestimmte Privilegien und sorgst so für Sicherheit und Kontrolle darüber, wer welche Aktionen in der Datenbank ausführen darf.

## GRANT

Mit `GRANT` weist du Benutzern oder Rollen bestimmte Privilegien auf Datenbankobjekte zu.

**Syntax:**

```
GRANT privileges ON object TO user [WITH GRANT OPTION];
```

- `privileges` : die erlaubten Aktionen (z. B. SELECT, INSERT, UPDATE, DELETE, ALL PRIVILEGES).
- `object` : Datenbank, Tabelle, View, Prozedur usw.
- `user` : Name des Benutzers oder der Rolle, der die Rechte zugewiesen werden.
- `WITH GRANT OPTION` (optional): erlaubt es dem Empfänger, die Privilegien an andere weiterzugeben.

**Beispiel:** So gibst du dem Benutzer `user1` das Recht, Daten aus der Tabelle `employees` zu lesen:

```
GRANT SELECT ON employees TO user1;
```

## REVOKE

Mit `REVOKE` ziehst du zuvor erteilte Privilegien von Benutzern oder Rollen wieder ein.

**Syntax:**

```
REVOKE privileges ON object FROM user;
```

**Beispiel:** So entziehst du `user1` das Recht, aus der Tabelle `employees` zu lesen:

```
REVOKE SELECT ON employees FROM user1;
```

Frage 39

# Erkläre den Einsatz von Hash-Funktionen in SQL

Hash-Funktionen werden in SQL eingesetzt, um Eingabedaten beliebiger Länge in einen String fester Länge umzuwandeln. Diesen Vorgang nennt man **Hashing**. Er wird breit eingesetzt, um Sicherheit zu erhöhen, Datenintegrität sicherzustellen und Vergleichs- bzw. Suchoperationen zu beschleunigen.

## Wichtige Einsatzbereiche für Hash-Funktionen in SQL:

### 1. Speichern von Passwörtern

- Statt Passwörter im Klartext zu speichern, legst du sie als Hash ab. Das erhöht die Sicherheit deutlich: Selbst wenn die Datenbank kompromittiert wird, lassen sich die Originalpasswörter nicht wiederherstellen.
- Eingesetzt werden kryptografische Hash-Funktionen wie `SHA-256` oder `SHA-512`.

### 2. Prüfung der Datenintegrität

Mit Hash-Funktionen kannst du erkennen, ob Daten verändert wurden. Du bildest einen Hash über die Originaldaten und vergleichst ihn mit dem aktuellen Hash — passt etwas nicht, hat sich etwas geändert.

### 3. Indexierung und schnellere Suche

Hash-Werte eignen sich gut, um große Datenmengen schnell zu vergleichen oder Indizes für eine performantere Suche aufzubauen.

### 4. Eindeutige IDs erzeugen:

Hashing hilft dabei, eindeutige Identifier für Datensätze auf Basis ihres Inhalts zu generieren.

## Beispiel für den Einsatz von Hash-Funktionen in SQL:

```
-- Tabelle für Benutzer anlegen
CREATE TABLE Users (
    UserID INT PRIMARY KEY,
    Username VARCHAR(50) NOT NULL,
    PasswordHash VARBINARY(512) NOT NULL
);

-- Beim Anlegen eines neuen Benutzers hashen wir das Passwort
INSERT INTO Users (UserID, Username, PasswordHash)
VALUES (1, 'user1', HASHBYTES('SHA2_512', 'password123'));

-- Prüfen, ob das eingegebene Passwort zum gespeicherten Hash passt
SELECT UserID FROM Users
WHERE Username = 'user1' AND PasswordHash = HASHBYTES('SHA2_512', 'password123');
```

## Empfehlungen und Hinweise

### Wahl der Hash-Funktion

- Greife zu modernen, robusten Algorithmen wie `SHA-256` oder `SHA-512`.
- Vermeide veraltete Funktionen wie `MD5` oder `SHA1` — sie haben bekannte Schwachstellen.

### Sichere Speicherung

Speichere Hashes binär ( `VARBINARY` ) statt als String ( `VARCHAR` ), das schont Speicher und vermeidet Genauigkeitsverluste.

### Hashes sind nicht umkehrbar

Denk dran: Hashing ist eine Einwegfunktion. Aus einem Hash lassen sich die Originaldaten nicht rekonstruieren.

Frage 40

# Wie funktioniert ein SQL-Trigger?

## Was ist ein SQL-Trigger

> ***Ein SQL-Trigger*** *ist ein Datenbankobjekt — eine spezielle Form einer Stored Procedure, die automatisch ausgeführt wird, sobald in der Datenbank ein bestimmtes Ereignis eintritt. Solche Ereignisse können* `INSERT` *-,* `UPDATE` *- oder* `DELETE` *-Operationen auf Tabellen oder Views sein.*

## Wie ein SQL-Trigger funktioniert

Ein Trigger wird automatisch aktiviert, sobald das definierte Ereignis eintritt (zum Beispiel das Einfügen eines neuen Datensatzes in eine Tabelle).

### Trigger-Typen

- `BEFORE` : Wird vor der eigentlichen Operation ausgelöst.

Praktisch, um Daten vor dem Speichern zu prüfen oder zu verändern.

- `AFTER` : Wird nach der Operation ausgelöst.

Häufig im Einsatz für Logging oder zum Aktualisieren verknüpfter Tabellen.

- `INSTEAD OF` : Ersetzt das Standardverhalten der Operation.

Nützlich, wenn du das Default-Verhalten überschreiben willst — zum Beispiel bei Views.

Innerhalb eines Triggers kannst du über die speziellen Pseudo-Tabellen `OLD` und `NEW` auf die alten und neuen Werte zugreifen.

## Beispiel für einen Trigger

Du hast eine Tabelle `Employees` und möchtest die Gehaltshistorie der Mitarbeitenden bei jeder Änderung automatisch in einer Tabelle `SalaryHistory` festhalten.

**Tabellen anlegen:**

```
-- Tabelle der Mitarbeitenden
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    Name VARCHAR(100),
    Salary DECIMAL(10, 2)
);

-- Tabelle für die Gehaltshistorie
CREATE TABLE SalaryHistory (
    EmployeeID INT,
    OldSalary DECIMAL(10, 2),
    NewSalary DECIMAL(10, 2),
    ChangeDate DATETIME DEFAULT CURRENT_TIMESTAMP
);
```

**Trigger anlegen:**

```
DELIMITER $$

CREATE TRIGGER trg_AfterSalaryUpdate
AFTER UPDATE ON Employees
FOR EACH ROW
BEGIN
    IF OLD.Salary <> NEW.Salary THEN
        INSERT INTO SalaryHistory (EmployeeID, OldSalary, NewSalary)
        VALUES (NEW.EmployeeID, OLD.Salary, NEW.Salary);
    END IF;
END$$

DELIMITER ;
```

## Wann du Trigger einsetzen solltest

- **Logging und Audit:**

Datenänderungen automatisch protokollieren, um Benutzeraktionen nachzuvollziehen.

- **Datenintegrität:**

Komplexe Geschäftsregeln durchsetzen, die sich mit reinen Constraints nicht abbilden lassen.

- **Datensynchronisation:**

Verknüpfte Tabellen automatisch aktualisieren, wenn sich Daten ändern.

- **Werte berechnen:**

Aggregierte oder abgeleitete Werte automatisch berechnen und auf dem neuesten Stand halten.