stackfuel

SQL Codeübersicht - Kapitel 3

Hier findest du die wichtigsten Befehle der Lektionen mit Beispielcode. Wenn du dir den Output jeder Query noch einmal anschauen willst, kannst du einen Blick in die Notebooks werfen.

Tabellen erzeugen und verändern

`CREATE TABLE` erzeugt Tabellen. Dabei folgt nach dem Tabellennamen eine Liste von Spalten und Eigenschaften.

```
CREATE TABLE series
(series_id SMALLINT UNSIGNED,
name VARCHAR(100),
num_seasons TINYINT UNSIGNED,
release_date DATE,
finished ENUM('Y', 'N') NOT NULL,
PRIMARY KEY (series_id));
```

`ALTER TABLE` verändert bestehende Tabellen. So können Spalten oder Constraints (z. B. Schlüsseldefinitionen) angepasst werden.

```
ALTER TABLE series
MODIFY series_id SMALLINT UNSIGNED AUTO_INCREMENT,
ADD FOREIGN KEY (cast_id) REFERENCES cast (cast_id);
```

`INSERT INTO` fügt neue Daten in eine Tabelle ein. Dabei müssen Werte für alle Spalten, die nicht NULL sein können, angegeben werden.

```
INSERT INTO series
(series_id, name, num_seasons, release_date, finished)
VALUES
(NULL, 'The Big Drink Family', 16, '2001-11-10', 'N');
```

`UPDATE` ermöglicht die Aktualisierung einzelner Werte.

```
UPDATE series
SET num_seasons = 8
WHERE series_id = 2;
```

`DROP` löscht Tabellen. Dabei dürfen diese keine Foreignkey/Fremdschlüssel-Constraints zu anderen Tabellen enthalten.

```
DROP TABLES series_cast, cast, series;
```

Views

Views ermöglichen das Speichern von Queries, um Tabellen für Nutzende bereitzustellen.
Sie können wie Tabellen genutzt werden. Du kannst sie mit `CREATE VIEW` erstellen und mit `DROP VIEW` wieder löschen.

```
CREATE VIEW customer_overview AS
(SELECT customer_id,
        first_name,
        last_name,
        CONCAT(SUBSTR(email,1, 1), '*********',
               SUBSTR(email, -20)) AS email
FROM customer);
DROP VIEW customer_overview;
```

Subqueries

Subqueries folgen der normalen Querystruktur. Achte darauf, sie in Klammern zu setzen und für `JOIN`s und `UNION`s zu benennen:

```
SELECT customer_id
FROM payment
WHERE amount = (SELECT MAX(amount)
                FROM payment);
```

CTEs

CTEs funktionieren ähnlich zu Subqueries. Beachte, dass sie immer benannt sein müssen. CTEs können nach ihrer Definition beliebig oft innerhalb einer Query genutzt werden.

```
WITH min_rental_duration AS
(SELECT MIN(DATEDIFF(return_date, rental_date)) AS min_duration
 FROM rental)
```

UNION/ UNION ALL

`UNION` verbindet zwei Tabellen miteinander. Dabei werden die Tabellen übereinander ‚gestapelt'. Beide Tabellen müssen dabei die gleichen Spalten enthalten. Standardmäßig werden Duplikate identischer Zeilen gelöscht. Die Modifikation `UNION ALL` sorgt dafür, dass auch alle Duplikate in der Ergebnistabelle auftauchen.

```
SELECT * FROM (
(SELECT c.customer_id, c.store_id,
       SUM(p.amount) AS total_payments
FROM customer AS c
INNER JOIN payment AS p
ON c.customer_id = p.customer_id
WHERE c.store_id = 1
GROUP BY c.customer_id
ORDER BY total_payments DESC
LIMIT 10)
UNION ALL
(SELECT c.customer_id, c.store_id,
        SUM(p.amount) AS total_payments
FROM customer AS c
INNER JOIN payment AS p
ON c.customer_id = p.customer_id
WHERE c.store_id = 2
GROUP BY c.customer_id
ORDER BY total_payments DESC
LIMIT 10)) AS full_table;
```

OUTER JOIN (LEFT/RIGHT)

Verbindet Tabellen anhand eines gemeinsamen Schlüssels (Key), wobei Tabelle 1 um die Werte von Tabelle 2 erweitert wird. Wenn keine Werte existieren, wird `NULL` eingefügt:

```
SELECT *
FROM customer_1 AS c1
LEFT JOIN customer_2 AS c
    ON c1.customer_id = c2.customer_id;
```

CROSS JOIN

Verbindet Tabellen als kartesisches Produkt (alle möglichen Paare der Tabellenzeilen). Hierbei wird kein gemeinsamer Schlüssel (Key) benötigt. Dieser Join ist sehr rechenintensiv und selten sinnvoll.

```
SELECT *
FROM customer_1
CROSS JOIN customer_2;
```

INDEX

Ein Index kann über eine Tabellenveränderung mit `ALTER TABLE` und `ADD INDEX` erstellt werden.

```
ALTER TABLE customer
ADD INDEX idx_first_name (first_name);
```

Einzigartiger INDEX (UNIQUE)

Ein einzigartiger Index kann wie ein normaler Index erstellt werden, indem stattdessen `UNIQUE` genutzt wird.

```
ALTER TABLE customer
ADD UNIQUE idx_email (email);
```

ALL

`ALL` gibt `TRUE` aus, wenn alle Werte einer Menge die formulierte Bedingung erfüllen. Diese Bedingung kann mit `NOT ALL` ins Gegenteil verkehrt werden.

```
SELECT customer_id, COUNT(*) AS num_rentals
FROM rental
GROUP BY customer_id
HAVING num_rentals > ALL rentals_from_DACH;
```

ANY

`ANY` gibt `TRUE` zurück, wenn irgendein Wert in der Menge die Bedingung erfüllt. Diese Bedingung kann mit `NOT ANY` ins Gegenteil verkehrt werden.

```
SELECT customer_id, COUNT(*) AS num_rentals
FROM rental
GROUP BY customer_id
HAVING num_rentals > ANY rentals_from_DACH;
```

EXISTS

`EXISTS` wird genutzt, um zu überprüfen, ob eine Beziehung existiert, ohne dass wir diese quantifizieren wollen. Dabei nutzt man `EXISTS` meist in Verbindung mit korrelierten Subqueries. Diese Bedingung kann mit `NOT EXISTS` ins Gegenteil verkehrt werden.

```
SELECT c.first_name, c.last_name
FROM customer AS c
WHERE EXISTS(SELECT * FROM rental AS r
             WHERE r.customer_id = c.customer_id AND
             DATE(r.rental_date) = (SELECT MIN(DATE(rental_date)
                                    FROM rental));
```

CASE

`CASE` prüft in jedem `WHEN`-Statement eine Bedingung, die an eine Spalte gebunden ist und, erzeugt einen Wert, wenn diese `TRUE` ist.

```
SELECT first_name, last_name,
       CASE
        WHEN active = 1 THEN 'ACTIVE'
        ELSE 'INACTIVE'
       END AS activity_type
FROM customer;
```

STRINGFUNKTIONEN

REGEXP

`REGEXP` ermöglicht das Filtern von Strings mit Regular Expressions und der damit einhergehenden höheren Flexibilität als mit Wildcard-Operatoren.

```
SELECT first_name, last_name,
FROM customer
WHERE first_name REGEXP '^[XYZ]';
```

Gruppierungen:

```
[abc]          Findet irgendeines der Zeichen in den eckigen Klammern, hier 'a', 'b'
               oder 'b'

[a-z]          Findet irgendein Zeichen zwischen a und z (in der ASCII Reihenfolge)
```

Spezielle Zeichen:

```
^              Findet den Beginn eines Strings oder einer Zeile

\d             Findet eine Ziffer (0-9)

\s             Findet Leerzeichen
```

Quantifizierer: Geben an, wie oft ein bestimmtes Zeichen oder eine Abfolge von Zeichen gematched werden soll.

```
*               Findet null oder mehr Übereinstimmungen, so viele wie möglich

+               Findet eine oder mehr Übereinstimmungen, so viele wie möglich

?               Findet genau eine Übereinstimmung
```

CONCAT()

`CONCAT()` fügt mehrere Strings zu einer einzigen zusammen. Beachte, dass Leerzeichen zwischen den Strings explizit eingefügt werden müssen.

```
SELECT CONCAT(first_name, ' ',last_name) AS actor_name
FROM actor;
```

LENGTH()

`LENGTH()` bestimmt die Anzahl der Zeichen eines Strings.

```
SELECT customer_id,
       email,
       LENGTH(email) AS email_length
FROM customer;
```

DATUMSFUNKTIONEN

DATE_ADD()

`DATE_ADD()` addiert zu einem Ausgangsdatum bzw. einer Ausgangszeit ein Zeitintervall. Hierfür wird außerdem das Keyword `INTERVAL` genutzt.

```
SELECT COUNT(DISTINCT customer_id)
FROM rental
WHERE rental_date BETWEEN (SELECT MIN(rental_date
                           FROM rental)
                  AND DATE_ADD((SELECT MIN(rental_date
                  FROM rental), INTERVAL 1 WEEK);
```

DATEDIFF ()

`DATEDIFF()` gibt die Differenz zwischen zwei Datumsangaben in Tagen aus.

```
SELECT DATEDIFF(r.return_date,
                p.payment_date) AS time_rental_to_payment
FROM rental AS r
INNER JOIN payment AS p
    ON r.rental_id = p.rental_id
```

stackfuel

Es gibt noch viele weitere Datumsfunktionen, die einzelne Teile aus einer Datums- und Zeitangabe extrahieren können. Einige findest du in der folgenden Tabelle

Funktion	Erklärung
DATE()	Gibt das Datum aus DATETIME aus
DAY()/DAYOFMONTH()	Gibt den Tag im Monat aus (01-31)
DAYNAME()	Gibt den Namen des Wochentages aus
DAYOFWEEK()	Gibt den Index des Wochentages aus
WEEK()	Gibt die Wochennummer aus
MONTH()	Gibt den Monat als Zahl aus
MONTHNAME()	Gibt den Monatsnamen aus
QUARTER()	Gibt das Quartal aus
YEAR()	Gibt das Jahr aus
TIME()	Gibt die Zeit aus DATETIME aus
HOUR()	Gibt die Stunde aus
MINUTE()	Gibt die Minute aus,