

Name:  
 Klasse:  
 Datum:  
 Schuljahr: 2022/23

## Die 3NF kann mich mal!



Die Normalformen dienen dazu, sicher zu stellen, dass die Struktur einem relationalen System entspricht und Anomalien bzw. doppelte Einträge vermieden werden. Für den Entwickler stellt dies aber gerade bei großen Datenbankmodellen und nachträglichen Änderungen immer ein Problem dar.

Andere Systeme, wie „Document Stores“ (z. B. MongoDB) verzichten komplett auf eine Struktur und lassen chaotische Strukturen zu – der Entwickler wird schon wissen was er macht. 😊

Bei relationalen Systemen gibt es einige Bereiche, die es einem erlauben, die starre Struktur etwas auszuhebeln.

## Verwendete DB-Version:



Version MariaDB	Version gm3
10.9.2-MariaDB-1:10.9.2+maria~ubu2204	2022-08-12

## Views – virtuelle Tabellen

Views sind Abfragen, die in der Datenbank fest gespeichert werden. Sie stellen somit virtuelle Tabellen dar. Dabei wird ein entsprechender `SELECT`-Befehl gespeichert (meist schon in übersetzter Form), der dann bei jedem Zugriff ausgeführt wird.

Informieren Sie sich über den `CREATE VIEW`-Befehl und notieren Sie die wichtigsten Parameter. Die wichtigsten Beschränkungen sind dabei (aus dem MariaDB-Handbuch):

- The `SELECT` statement cannot contain a subquery in the `FROM` clause.
- The `SELECT` statement cannot refer to system or user variables.
- Within a stored program, the definition cannot refer to program parameters or local variables.
- The `SELECT` statement cannot refer to prepared statement parameters.
- Any table or view referred to in the definition must exist. However, after a view has been created, it is possible to drop a table or view that the definition refers to. In this case, use of the view results in an error. To check a view definition for problems of this kind, use the `CHECK TABLE` statement.
- The definition cannot refer to a `TEMPORARY` table, and you cannot create a `TEMPORARY` view.
- Any tables named in the view definition must exist at definition time.
- You cannot associate a trigger with a view.
- For valid identifiers to use as view names, see Identifier Names.

Mit `DROP VIEW` wird eine View gelöscht, mit `ALTER VIEW` geändert. Eine View-Definition wird mit `SHOW CREATE VIEW` angezeigt.

Verwendet man sehr häufig bestimmte Abfragen auf eine Tabelle, evtl. sogar über Verknüpfungen, so macht eine View Sinn. Ein zusätzlicher Vorteil ist, dass der SQL-Ausdruck nur bei der Erstellung der View geparkt werden muss und dann „übersetzt“ ist. Ein `SELECT` wird im Normalfall jedes Mal geparkt, wenn dieser nicht zwischengespeichert wird.



## Beispiel

Nachfolgendes Beispiel legt eine View an. Die Abfrage kennen Sie bereits aus dem vorherigen Kapitel (Gesucht sind alle Kunden (Id, Name) incl. Lieferdatum, die vor über einem Monat oder mehr ihre Bestellung erhalten haben, aber noch nicht bezahlt haben.).

```
USE gm3;
CREATE OR REPLACE VIEW kundenichtbezahlt AS
  SELECT k.id, k.name, b.liefdatum
  FROM kunde k INNER JOIN bestellung b ON kid=k.id
  WHERE TimeStampDiff(Month, liefdatum, now()) >= 1 AND bezahlt = 0
  ORDER BY k.name ASC;
```

Ein SHOW TABLES zeigt die View wie eine ganz normale Tabelle an.

```
USE gm3;
SHOW TABLES LIKE 'kundenichtbezahlt';
```

```
+-----+
| Tables_in_gm3 (kundenichtbezahlt) |
+-----+
| kundenichtbezahlt                  |
+-----+
```

Der Aufruf wird dann ganz einfach:

```
USE gm3;
SELECT * FROM kundenichtbezahlt LIMIT 5;
```

```
+----+-----+-----+
| id | name                | liefdatum |
+----+-----+-----+
| 20 | BASIS und Partner   | 2021-10-28 |
| 20 | BASIS und Partner   | 2021-10-31 |
| 8  | Biergarten Waldesruh | 2021-11-25 |
| 15 | Café Maldaner       | 2021-09-10 |
| 15 | Café Maldaner       | 2021-10-08 |
+----+-----+-----+
```

Der Vorteil einer View ist es, dass die Abfragen zwischengespeichert werden. Erkennt das DBMS, dass sich abhängige Werte, Tabellen, ... geändert haben, so wird die View als „dirty“ markiert und bei der nächsten Abfrage werden alle Daten erneut ermittelt. Somit wird immer sicher gestellt, dass auch abhängige oder berechnete Werte immer korrekt sind.

**Notiz:**



1. Erstellen Sie eine View mit dem Namen `mitarbeiter_alter`, die zusätzlich das Alter des Mitarbeiters anzeigt.

Hinweis: Als Spaltennamen (alias) dürfen Sie nicht `alter` verwenden, da es sonst zu einem Fehler kommen kann - Schlüsselwort `ALTER TABLE...`

Diese soll dann wie folgt abgefragt werden:

```
use gm3;
SELECT id, name, vorname, gebdat, alter_jahre
FROM mitarbeiter_alter
ORDER BY alter_jahre DESC ;
```

Kontrollergebnis - Anzahl Datensätze insgesamt: 41

id	name	vorname	gebdat	alter_jahre
16	Hellmeister	Sepp	1959-08-01	63
21	Michael	Konrad	1962-03-24	60
29	Iwansky	Sonja	1962-10-31	59



## Common Table Expressions - CTE

Common Table Expressions (kurz CTE bzw. Sub-Query Factoring bzw. allgemeiner Tabellenausdruck) ermöglichen es, eine temporäre Ergebnismenge innerhalb einer Abfrage zu erstellen.

Eine CTE ist wie eine View, da sie nicht als Datenbankobjekt gespeichert wird und nur während der Ausführung einer Abfrage existiert. Diese muss daher nicht die 3NF erfüllen.

Im Gegensatz zu einer View kann eine CTE innerhalb einer Abfrage mehrfach referenziert werden und viele Beschränkungen, wie bei der View, entfallen. Darüber hinaus können Sie eine CTE in sich selbst referenzieren - dieses wird als rekursive CTE bezeichnet.

Ein CTE kann verwendet werden, um

- eine Ergebnismenge mehrfach in derselben Anweisung zu referenzieren. 👍
- Einen View zu ersetzen, um die Erstellung von Views zu vermeiden.
- eine rekursive Abfrage zu erstellen.
- eine komplexe Abfrage zu vereinfachen, indem sie in mehrere einfache und logische Bausteine zerlegt wird. 👍

### Syntax:

Einfach ausgedrückt, ist die WITH-Klausel ein optionaler Präfix für einen SELECT.

```
WITH [RECURSIVE] table_reference [(columns_list)] AS (  
    SELECT ...  
)  
[CYCLE cycle_column_list RESTRICT]  
SELECT/UPDATE/DELETE/INSERT ... table_reference
```

Nach dem Schlüsselwort WITH entspricht die Syntax der von CREATE VIEW, es beginnt mit dem Namen gefolgt von einer optionalen Liste, die den Ergebnisspalten Namen zuweist. Danach folgt das Schlüsselwort AS, das die eigentliche Definition, also die Abfrage, einleitet.

Der Vorteil von CTE ist dabei, dass diese nicht wie die View permanent gespeichert werden, was schnell zu einer unverhältnismäßigen Zahl von Views („namespace pollution“) führen kann.

### Notiz:



## Obiges Beispiel kann man entsprechend umbauen:

```
USE gm3;
WITH kundenichtbezahlt (id, name, liefdatum) AS (
    SELECT k.id, k.name, b.liefdatum
    FROM kunde k INNER JOIN bestellung b ON kid=k.id
    WHERE TimeStampDiff(Month, liefdatum, now()) >= 1 AND bezahlt = 0
    ORDER BY k.name ASC
)
SELECT * FROM kundenichtbezahlt LIMIT 5;
```

id	name	liefdatum
1	Diskotheek Blue	2021-07-24
1	Diskotheek Blue	2021-10-08
1	Diskotheek Blue	2021-10-21
1	Diskotheek Blue	2021-11-08
2	Zum Goldenen Ross	2021-10-08

## Komplexe Abfragen vereinfachen

In der vorherigen Aufgabe (ein paar Bereiche vorher) „In dieser Aufgabe soll für jeden Monat (in deutscher Schreibweise) im Jahr 2021, in dem Produkte verkauft wurden, das Produkt (Id, Bezeichnung) ermittelt werden, das am meisten verkauft wurde. Das Ergebnis soll dabei nach dem Monat aufsteigend sortiert sein.“ wurde ein View erstellt und mit vielen subselects die Lösung bestimmt – ziemlich komplex.

Mit CTE lässt sich das Problem in kleine einfache Einzelschritte zerlegen und so schrittweise lösen.

```
USE gm3;
WITH
    xbestellungen AS (
        SELECT pid, month(bestdatum) AS monat, sum(menge) AS summe
        FROM bestpos INNER JOIN bestellung b ON b.id=bid
        WHERE year(bestdatum)=2021
        GROUP by 1,2
        ORDER by 2
    ),
    xmonatmenge AS (
        SELECT monat, max(summe) AS menge
        FROM xbestellungen
        GROUP BY 1
    ),
    xmonatproduktsumme AS (
        SELECT month(bestdatum) AS monat, p.id as pid, sum(menge) AS menge,
        p.bez, DATE_FORMAT(bestdatum, '%M', 'de_DE') AS monatDeutsch
        FROM bestpos
        INNER JOIN bestellung b ON b.id=bid
        INNER JOIN produkt p ON p.id=bestpos.pid
        GROUP BY pid,1
    ),
    xmonatdeutsch AS (
        SELECT x.monat, x.monatDeutsch as Monatsname, x.pid , x.bez, m.menge
        FROM xmonatproduktsumme x, xmonatmenge m
        WHERE m.monat=x.monat AND m.menge=x.menge
    )
SELECT Monatsname, pid, bez, menge
FROM xmonatdeutsch
ORDER BY monat;
```



## Kontrollergebnis

Monatsname	pid	bez	menge
Juni	119	Gerolsteiner Classic	48
Juli	123	Odenwald Quelle Classic	90
August	221	Odenwald-Quelle Apfel-Johannisbeer Pet	116
September	31	Bitburger Pils Stubbi	91
Oktober	220	Odenwald-Quelle Apfel-Kirsch Pet	205
November	53	Schöffelhofer Weizenbier Hefe dunkel	87

Zum Vergleich, die Lösung mit der View.

## Kontrollergebnis

Monatname	pid	bez	menge
Juni	119	Gerolsteiner Classic	48
Juli	123	Odenwald Quelle Classic	90
August	221	Odenwald-Quelle Apfel-Johannisbeer Pet	116
September	31	Bitburger Pils Stubbi	91
Oktober	220	Odenwald-Quelle Apfel-Kirsch Pet	205
November	53	Schöffelhofer Weizenbier Hefe dunkel	87

CTE bietet noch weitere Möglichkeiten, lesen Sie dazu einfach mal in der Dokumentation nach.



## Generierte Spalten

Bei generierten Spalten gibt es zwei Varianten:

**virtual** Die Spalte wird „on the fly“ berechnet, wenn die Spalte bei einer Abfrage verwendet wird und benötigt keinen Speicher in der Tabelle.

**stored/persistent** Die Spalte wird berechnet, wenn der Datensatz geschrieben oder verändert wird und als „reguläre“ Spalte gespeichert. Hierbei dürfen keine Funktionen verwendet werden, wie beispielsweise `curdate()`, die unabhängig von den anderen Werten der Tabelle sind.

Mehr dazu in der Dokumentation [Generated \(Virtual and Persistent/Stored\) Columns](#).

```
USE gm3;
CREATE OR REPLACE TABLE xmitarbeiter (
  id int(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
  name varchar(50),
  vorname varchar(50),
  gebdat date,
  printname varchar(102) AS (concat(name, ', ', vorname))          STORED,
  alter_jahre INT(3) AS (timestampdiff(year, gebdat, curdate())) VIRTUAL
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
INSERT INTO xmitarbeiter (name,vorname,gebdat)
  VALUES ('Kobold','Pumukel','1961-01-01');
SELECT * FROM xmitarbeiter;
```

### Kontrollergebnis

id	name	vorname	gebdat	printname	alter_jahre
1	Kobold	Pumukel	1961-01-01	Kobold, Pumukel	61

Die Tabellenstruktur sieht dabei wie folgt aus:

```
USE gm3;
DESC xmitarbeiter;
```

Field	Type	Null	Key	Default	Extra
id	int(11)	NO	PRI	null	auto_increment
name	varchar(50)	YES		null	
vorname	varchar(50)	YES		null	
gebdat	date	YES		null	
printname	varchar(102)	YES		null	STORED GENERATED
alter_jahre	int(3)	YES		null	VIRTUAL GENERATED



## Window Funktionen

Seit Version 10.2 stellt MariaDB die Window Functions zur Verfügung.

Window Funktionen ermöglichen dabei die Durchführung von Berechnungen über eine Reihe von Zeilen hinweg, die sich auf die aktuelle Zeile beziehen. Vereinfacht ausgedrückt ähneln Window Funktionen Aggregatsfunktionen, da sie Berechnungen über eine Reihe von Zeilen hinweg durchführen. Im Gegensatz zu Aggregatsfunktionen wird die Ausgabe jedoch nicht in einer einzelnen Zeile gruppiert.

Nicht aggregierte Window Funktionen:

CUME\_DIST  
DENSE\_RANK  
FIRST\_VALUE  
LAG  
LAST\_VALUE  
LEAD  
MEDIAN  
NTH\_VALUE  
NTILE  
PERCENT\_RANK  
PERCENTILE\_CONT  
PERCENTILE\_DISC  
RANK, ROW\_NUMBER

Aggregatsfunktionen, die auch als Window Funktionen verwendet werden können:

AVG  
BIT\_AND  
BIT\_OR  
BIT\_XOR  
COUNT  
MAX  
MIN  
STD  
STDDEV  
STDDEV\_POP  
STDDEV\_SAMP  
SUM  
VAR\_POP  
VAR\_SAMP  
VARIANCE





## RANK, ROW

RANK () ist eine Window Funktion, die die Nummer einer bestimmten Zeile anzeigt, beginnend bei eins und nach der ORDER BY-Sequenz der Window Funktion, wobei identische Werte dasselbe Ergebnis erhalten. Sie ähnelt der Funktion ROW\_NUMBER (), außer dass in dieser Funktion identische Werte für jedes Ergebnis eine andere Zeilennummer erhalten.

```
USE gm3;
SELECT m.name, m.vorname, g.gehalt, year(eingestellt) AS 'Einstellungsjahr',
       RANK() OVER (ORDER BY g.gehalt DESC) AS 'Rank'
FROM mitarbeiter m, gehalt g
WHERE m.id=g.id
LIMIT 10;
```

name	vorname	gehalt	Einstellungsjahr	Rank
Reibach	Bernd	10490.77	2010	1
Kaiser	Ralf	4765.00	2005	2
Humpe	Sybille	3770.00	2002	3
Kamp	Klaus-Dieter	3650.00	2005	4
Hagen	Friedhelm	3600.00	2000	5
Wieland	Brunhilde	3590.00	2006	6
Hoffmann	Theresa	3366.00	2006	7
Richter	Hans-Otto	3250.00	2004	8
Santer	Claudia-Maria	3172.00	2010	9
Lauterbach	Wilma	2510.00	2001	10

## Partition results – Partitionsergebnisse

Sie können die Ergebnisse in der OVER-Klausel partitionieren, d. h. der „Rank“ wird bei Änderung des Einstellungsjahres wieder bei „1“ gestartet. Im nachfolgenden Beispiel soll der Gehaltsrang für jedes Jahr ermittelt werden.

```
USE gm3;
SELECT m.name, m.vorname, g.gehalt, year(eingestellt) AS 'Einstellungsjahr',
       ROW_NUMBER() OVER (PARTITION BY year(eingestellt)
                          ORDER BY g.gehalt DESC) AS 'Rank'
FROM mitarbeiter m, gehalt g
WHERE m.id=g.id
LIMIT 10;
```

name	vorname	gehalt	Einstellungsjahr	Rank
Hagen	Friedhelm	3600.00	2000	1
Michael	Konrad	2444.00	2000	2
Collmar-Schmidt	Nadine	2353.00	2000	3
Gehrke	Anna-Marie	2150.00	2000	4
Lorenz	Sophia	1250.00	2000	5
Lauterbach	Wilma	2510.00	2001	1
Kaufmann	Sonja	2000.00	2001	2
Humpe	Sybille	3770.00	2002	1
Soerens	Helge	2255.00	2002	2
Berger	Ludwig	1850.00	2002	3



## Named windows

Window Funktionen lassen sich auch mit einem Namen speichern und dann beliebig oft in der Abfrage verwenden. Obiges Beispiel wird entsprechend umgebaut, so dass die Partitionierung nun unter dem Namen `w` verwendet wird.

```
USE gm3;
SELECT m.name, m.vorname, g.gehalt, year(eingestellt) AS 'Einstellungsjahr',
       ROW_NUMBER() OVER w AS 'Rank'
FROM mitarbeiter m, gehalt g
WHERE m.id=g.id
WINDOW w AS (PARTITION BY year(eingestellt) ORDER BY g.gehalt DESC)
LIMIT 10;
```

name	vorname	gehalt	Einstellungsjahr	Rank
Hagen	Friedhelm	3600.00	2000	1
Michael	Konrad	2444.00	2000	2
Collmar-Schmidt	Nadine	2353.00	2000	3
Gehrke	Anna-Marie	2150.00	2000	4
Lorenz	Sophia	1250.00	2000	5
Lauterbach	Wilma	2510.00	2001	1
Kaufmann	Sonja	2000.00	2001	2
Humpe	Sybille	3770.00	2002	1
Soerens	Helge	2255.00	2002	2
Berger	Ludwig	1850.00	2002	3

## First, last und nth Werte

Über ein definiertes Fenster kann jeweils der erste, der n-te und der letzte Wert ermittelt werden. Dabei wird immer von der aktuellen Zeile ausgegangen und daraus resultierend die Werte bestimmt. Dies bedeutet, nach dem ersten Datensatz in der Gruppe ist „first“ festgelegt. Der dritte Wert ist erst ab der dritten Zeile definiert, vorher `null`. Der Wert „last“ wird jeweils in der entsprechenden Zeile definiert - dies bedeutet, erst beim letzten Wert in der Gruppe, hat dieser in unserem Beispiel den kleinsten Wert.

```
USE gm3;
SELECT m.name, g.gehalt, year(eingestellt) AS 'E-Jahr',
       ROW_NUMBER() OVER w AS 'Rank',
       FIRST_VALUE(g.gehalt) OVER w AS 'first',
       NTH_VALUE(g.gehalt,3 ) OVER w AS 'drei',
       LAST_VALUE(g.gehalt) OVER w AS 'last'
FROM mitarbeiter m, gehalt g
WHERE m.id=g.id
WINDOW w AS (PARTITION BY year(eingestellt) ORDER BY g.gehalt DESC)
LIMIT 5;
```

name	gehalt	E-Jahr	Rank	first	drei	last
Hagen	3600.00	2000	1	3600.00	null	3600.00
Michael	2444.00	2000	2	3600.00	null	2444.00
Collmar-Schmidt	2353.00	2000	3	3600.00	2353.00	2353.00
Gehrke	2150.00	2000	4	3600.00	2353.00	2150.00
Lorenz	1250.00	2000	5	3600.00	2353.00	1250.00



## weitere Funktionen

Es können auch weitere Funktionen (siehe Liste am Anfang) verwendet werden.

```
USE gm3;
SELECT m.name, g.gehalt, year(eingestellt) AS 'E-Jahr',
       RANK() OVER w AS 'Rank',
       MIN(g.gehalt) OVER w AS 'min',
       AVG(g.gehalt) OVER w AS 'avg',
       MAX(g.gehalt) OVER w AS 'max'
FROM mitarbeiter m, gehalt g
WHERE m.id=g.id
WINDOW w AS (ORDER BY g.gehalt DESC)
LIMIT 10;
```

name	gehalt	E-Jahr	Rank	min	avg	max
Reibach	10490.77	2010	1	10490.77	10490.770000	10490.77
Kaiser	4765.00	2005	2	4765.00	7627.885000	10490.77
Humpe	3770.00	2002	3	3770.00	6341.923333	10490.77
Kamp	3650.00	2005	4	3650.00	5668.942500	10490.77
Hagen	3600.00	2000	5	3600.00	5255.154000	10490.77
Wieland	3590.00	2006	6	3590.00	4977.628333	10490.77
Hoffmann	3366.00	2006	7	3366.00	4747.395714	10490.77
Richter	3250.00	2004	8	3250.00	4560.221250	10490.77
Santer	3172.00	2010	9	3172.00	4405.974444	10490.77
Lauterbach	2510.00	2001	10	2510.00	4216.377000	10490.77

siehe auch:

- [MariaDB 10.2 Window Function Examples](#)
- [Übersicht der Fensterfunktionen](#)
- [Six Examples Using MySQL Window Functions](#)
- [Optimizing Queries Using CTEs and Window Functions](#) 👍

**Views und Co. anwenden** [AUF-09-1-1](#)

► [SQL-view](#)

