

© 2025 AUTEUR/ TOM VANHOUTTE

Cursus MySQL

Intro

*In deze cursus leer je hoe je gegevens in een database kunt aanmaken, bewerken, opvragen en verwijderen. Dit doen we met behulp van **MySQL**, één van de populairste relationele databasesystemen ter wereld.*

Wat is SQL?

SQL staat voor *Structured Query Language*. Het is de standaardtaal om relationele databases te beheren. Met SQL kun je tabellen aanmaken, relaties leggen en data manipuleren.

Wat is MySQL?

MySQL is een implementatie van SQL. Het ondersteunt de standaard SQL-commando's, maar voegt ook eigen uitbreidingen toe (zoals specifieke functies of optimalisaties). Andere implementaties van SQL zijn bijvoorbeeld Oracle, PostgreSQL en Microsoft SQL Server.

Hoe leer je in deze cursus?

- Belangrijke begrippen en commando's worden stap voor stap uitgelegd.
- Bij elk onderdeel vind je voorbeelden en oefeningen.
- Aan het einde van elke module krijg je opdrachten die je individueel oplost om je kennis te toetsen.

Database

Een **database** is een georganiseerde verzameling gegevens die op een gestructureerde manier wordt opgeslagen en beheerd door een computersysteem. Dankzij een database kun je grote hoeveelheden data efficiënt bewaren, terugvinden en bewerken.

Relationeel versus niet-relationeel

Er bestaan verschillende soorten databases:

- **Relationele databases (RDBMS)**

Hierbij worden gegevens opgeslagen in **tabellen** (vergelijkbaar met Excel). Tabellen kunnen onderling met elkaar verbonden worden via sleutels (primary en foreign keys). Voorbeelden: **MySQL**, **PostgreSQL**, **Oracle Database**, **Microsoft SQL Server**, **MariaDB**.

- **Niet-relationele databases (NoSQL)**

Deze zijn flexibeler qua structuur en worden vaak gebruikt voor grote hoeveelheden ongestructureerde data (zoals JSON-documenten of key-value stores).

Voorbeelden: **MongoDB**, **CouchDB**, **Redis**.

In deze cursus focussen we op **relationele databases**, met MySQL als praktische omgeving.

Waarom relationele databases?

Relationele databases hebben verschillende voordelen:

- ⚡ **Snel terugvinden van informatie** dankzij indexing en query-optimalisatie.
- ➔ **Relaties tussen tabellen** via sleutels (bijv. een klant met meerdere bestellingen).
- 🔍 **Efficiënt opslaan van data** zonder onnodige duplicatie.
- 🔒 **Beveiliging en rechtenstructuur** voor verschillende gebruikers.
- 🌐 **Gestandaardiseerde taal (SQL)**: dezelfde basiscommando's werken in vrijwel alle relationele databasesystemen.

Installatie

*E*en MySQL database wordt gebruikt in combinatie met webtoepassingen. Om lokaal te kunnen oefenen, hebben we een webserveromgeving nodig waarin zowel MySQL als PHPmyadmin beschikbaar zijn.

Installatie van MySQL en phpMyAdmin

Een MySQL-database wordt vaak gebruikt in combinatie met webtoepassingen. Om lokaal te kunnen oefenen, hebben we een **webserveromgeving** nodig waarin zowel MySQL als phpMyAdmin beschikbaar zijn.

Stap 1: Kies een serverpakket

De eenvoudigste manier om dit lokaal op je computer te installeren, is via een kant-en-klaar pakket dat Apache (webserver), MySQL (database) en PHP combineert. Enkele populaire opties zijn:

- **WampServer** (Windows) → <https://www.wampserver.com>
- **XAMPP** (Windows, macOS, Linux) → <https://www.apachefriends.org>
- **MAMP** (macOS, Windows) → <https://www.mamp.info>

In deze cursus gebruiken we **WampServer** als voorbeeld.

Stap 2: Installatie

1. Download de juiste versie (meestal de **64-bit versie**) van WampServer.
2. Voer het installatiebestand uit en volg de instructies.
3. Na installatie verschijnt een icoontje in de taakbalk rechtsonder.

● = server is uitgeschakeld ● = server is gedeeltelijk actief (Apache of MySQL niet gestart) ● = alles is correct gestart

Stap 3: phpMyAdmin openen

- Klik met de linkermuisknop op het groene WampServer-icoon.
- Kies **phpMyAdmin** in het menu.
- Er opent een browservenster waar je kunt inloggen.

Standaard log je in met:

- **Gebruikersnaam:** root
- **Wachtwoord:** leeg (druk gewoon op Enter)

Belangrijk: dit is enkel veilig in een **lokale ontwikkelomgeving**. Zodra je met een echte online server werkt, moet je altijd een sterk wachtwoord instellen.

Stap 4: Eerste kennismaking

Na het inloggen kom je in de **phpMyAdmin-interface** terecht. Hier kun je:

- nieuwe databases aanmaken,
- tabellen en velden definiëren,
- queries uitvoeren,
- gebruikersrechten beheren.

Onderaan het startscherm zie je informatie zoals:

- de **webserver** die draait (Apache),
- de **databaseserver** (MySQL),
- de **huidige beheeromgeving** (phpMyAdmin).

Belangrijk over storage engines

MySQL ondersteunt verschillende **storage engines**. De belangrijkste zijn:

- **InnoDB** → de standaard sinds MySQL 5.5, ondersteunt relaties (foreign keys), transacties en is de aanbevolen keuze.
- **MyISAM** → oude engine zonder transacties of foreign keys. Wordt vandaag **niet meer gebruikt** voor nieuwe projecten.

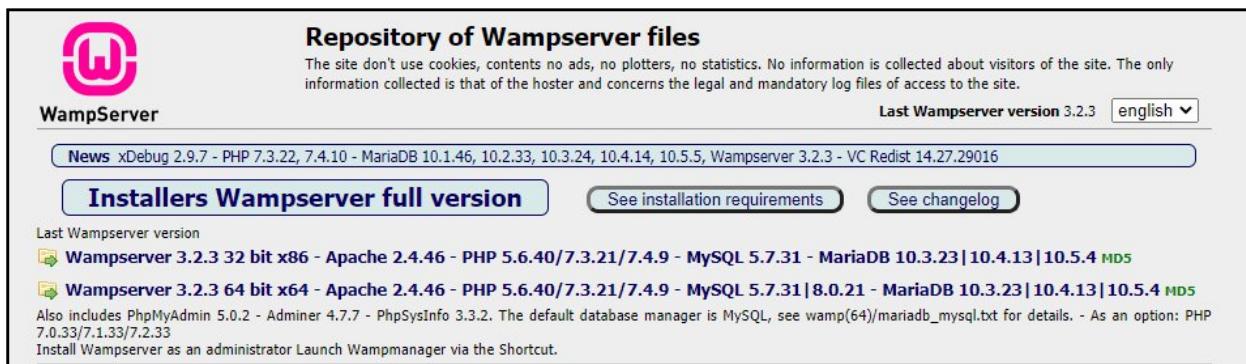
Zorg dat je tabellen altijd met **InnoDB** aanmaakt. Je hoeft dit meestal niet meer manueel in te stellen, want InnoDB is standaard actief.

Windows - wampserver

Belangrijke info

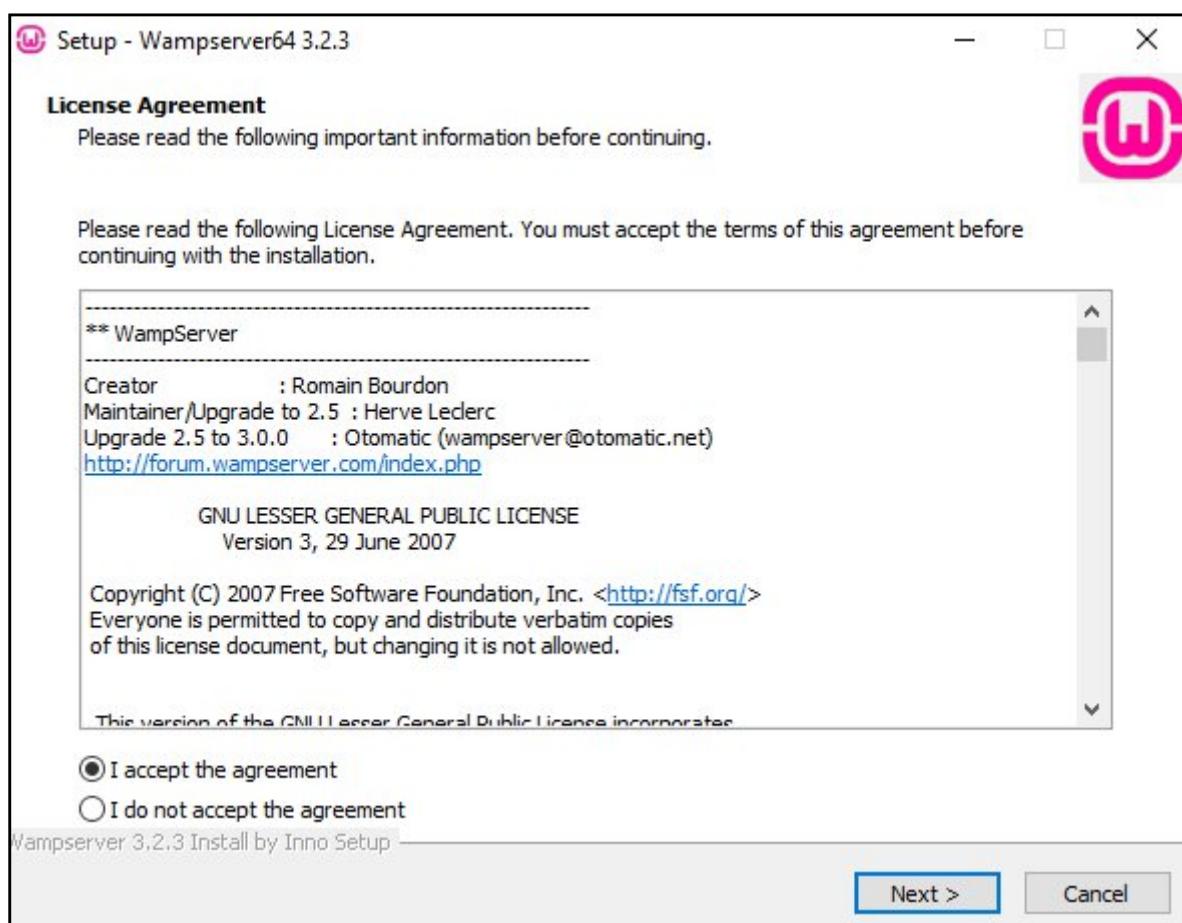
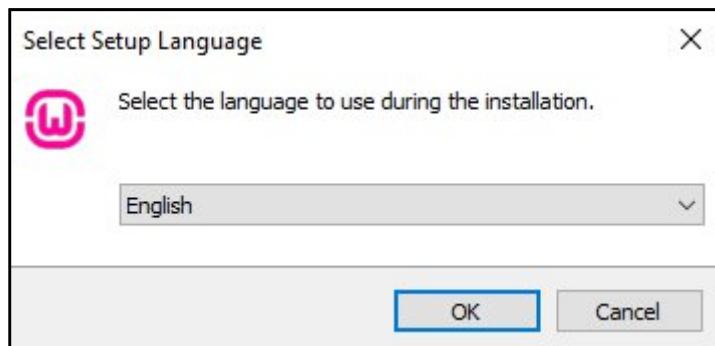
Om de apache webserver en de MySQL omgeving te installeren surf je naar de volgende link:
<http://wampserver.aviaTechno.net>

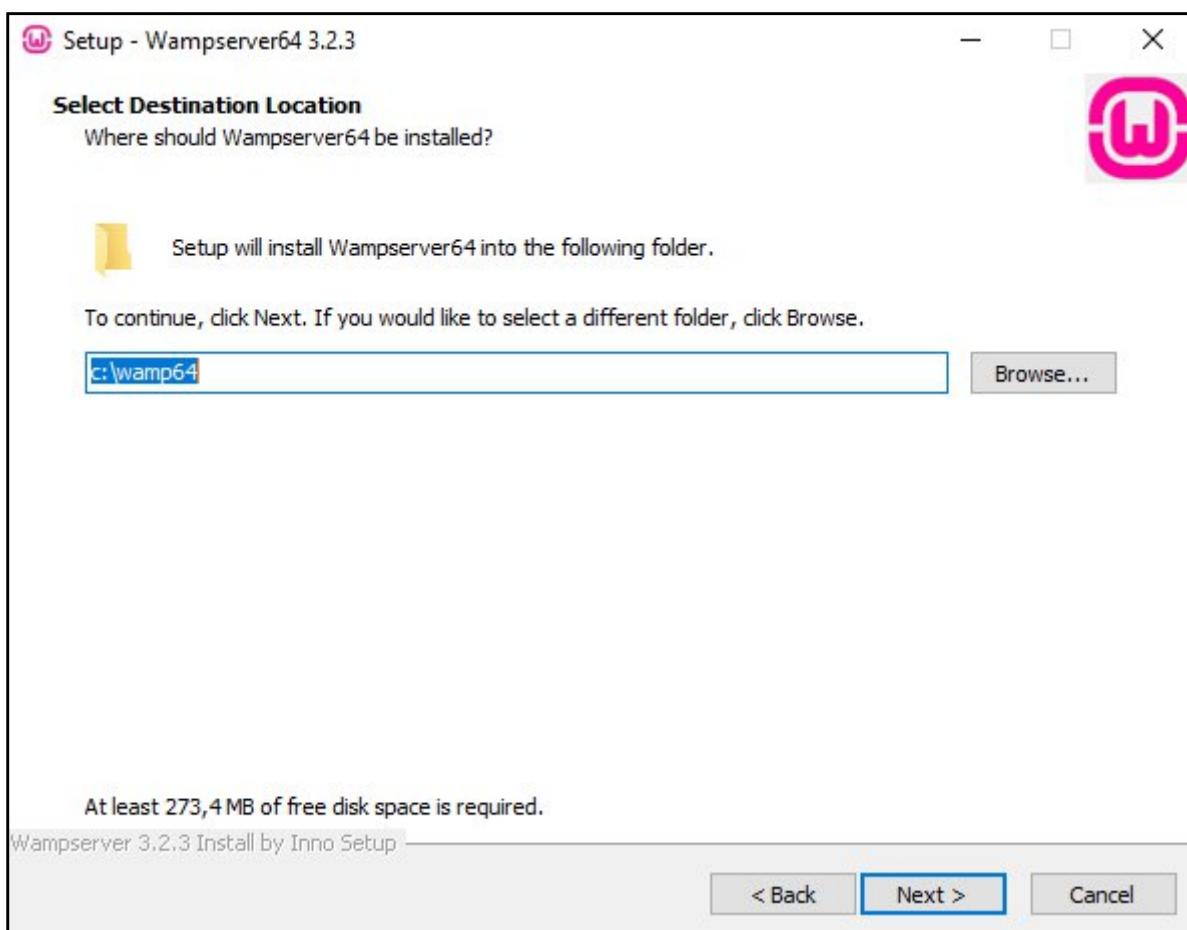
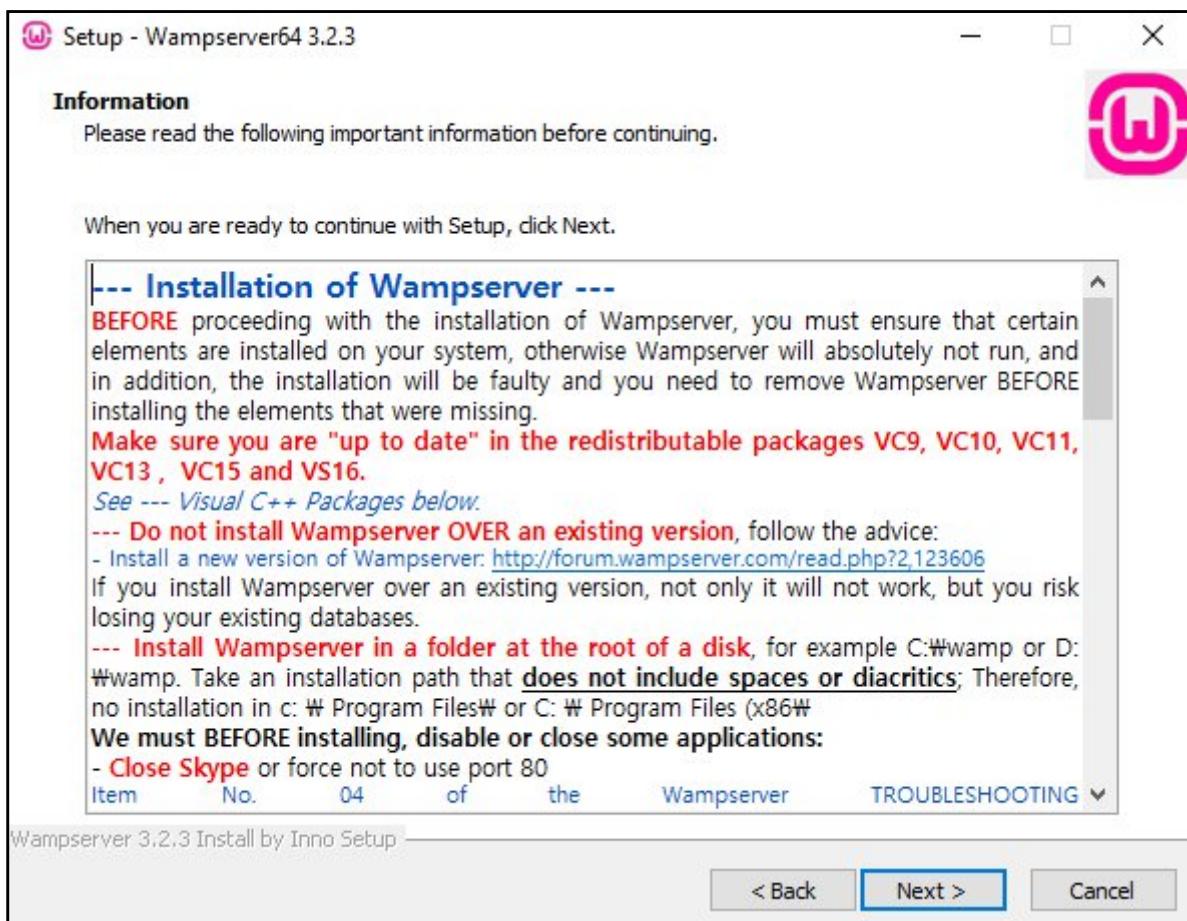
Download de gepaste versie (voor de meeste pc's zal dit de x64 versie zijn).

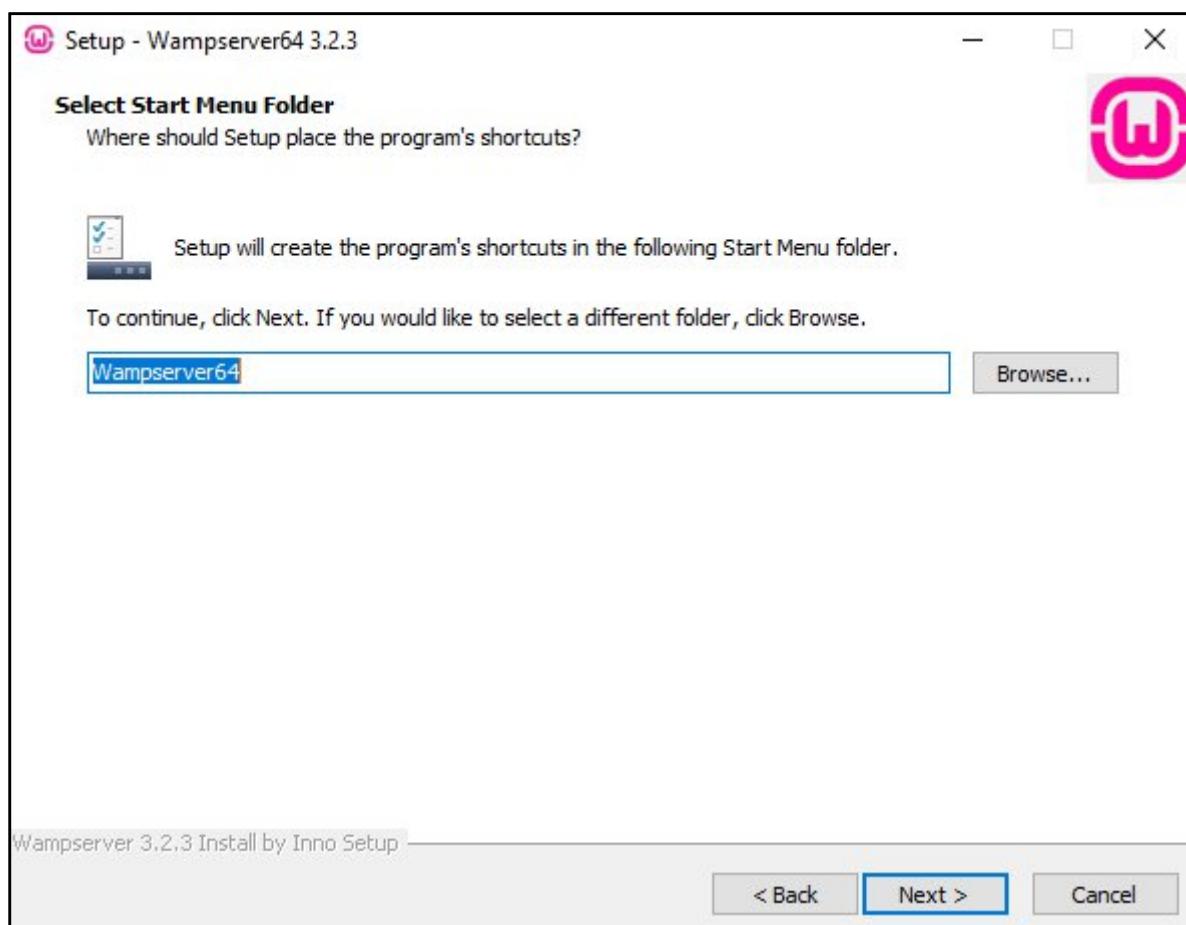
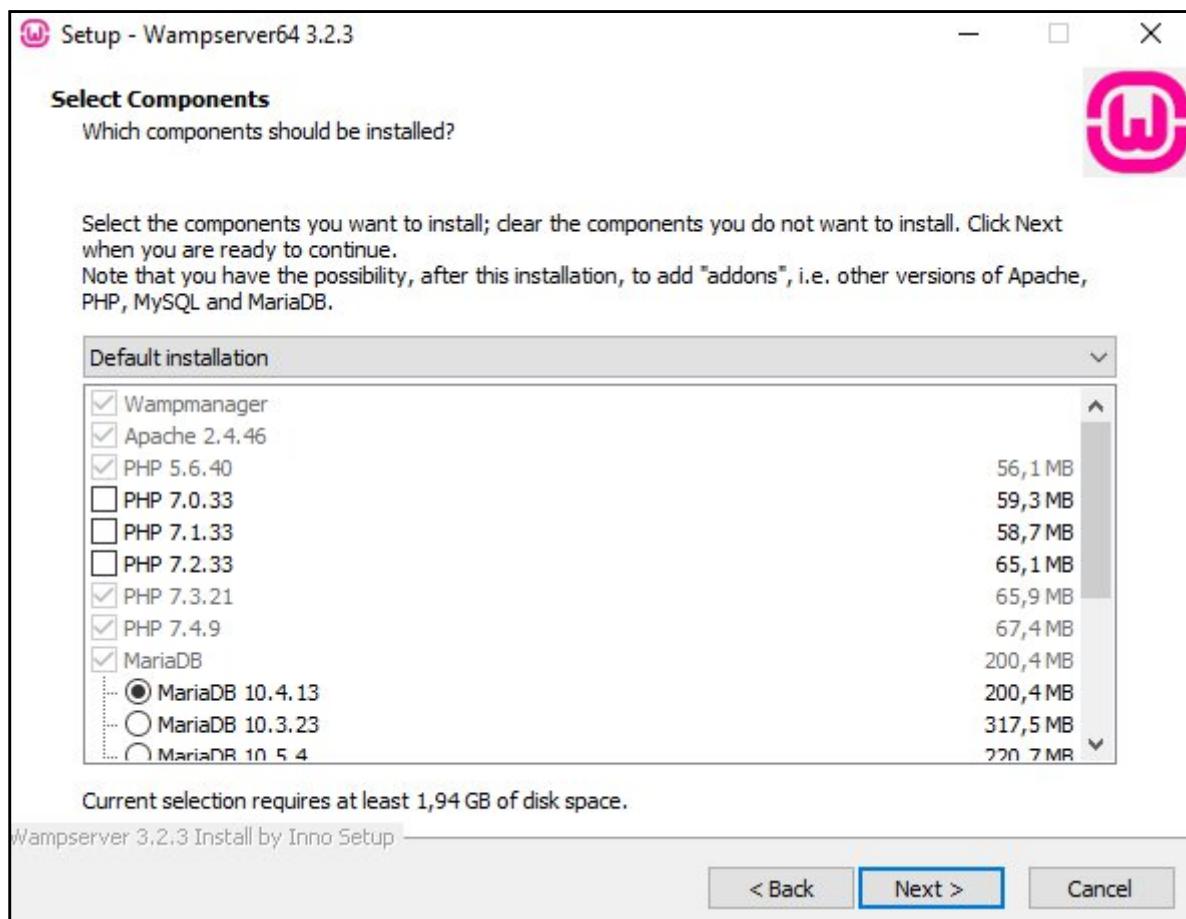


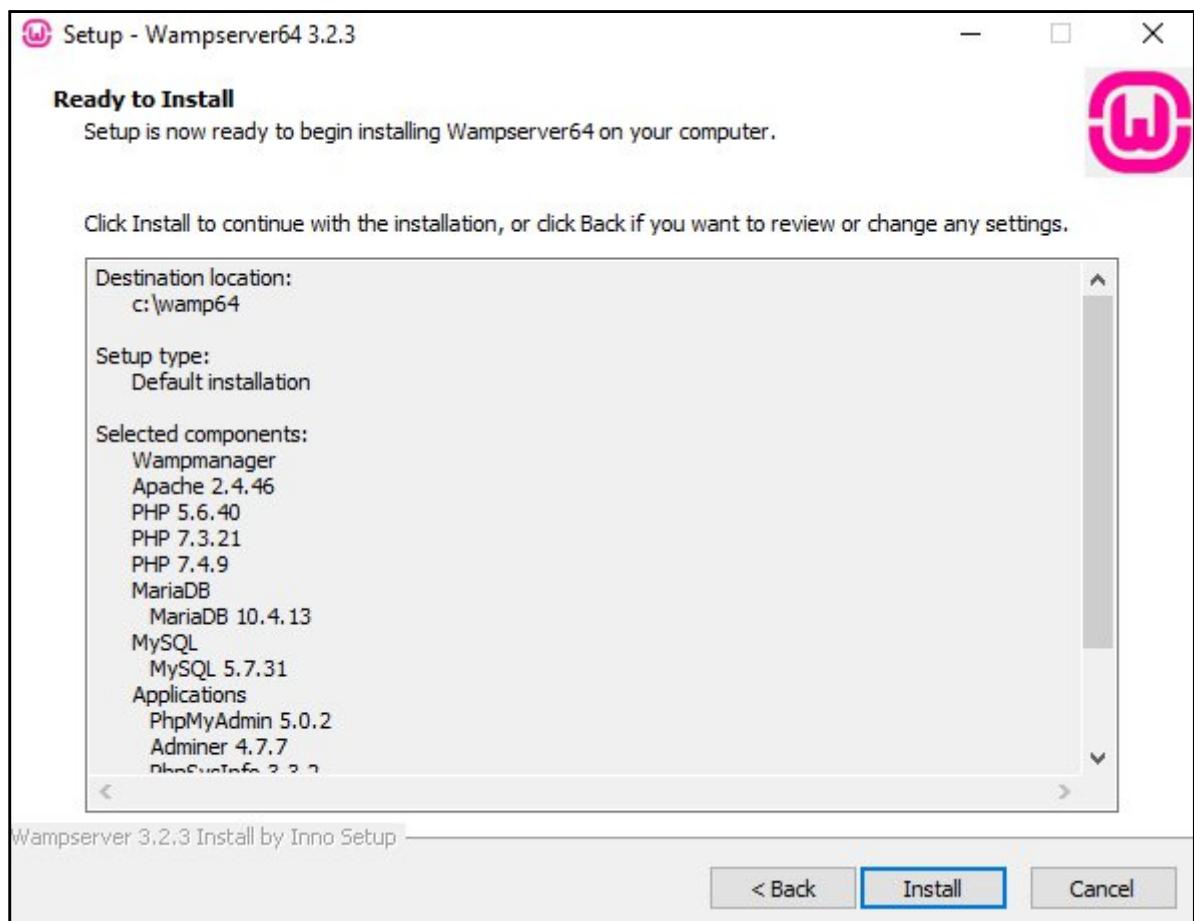
The screenshot shows the 'Repository of Wampserver files' page for WampServer version 3.2.3. It features a logo for WampServer, a news banner for xDebug 2.9.7, and links for 'Installers Wampserver full version'. Below these are download links for 'Wampserver 3.2.3 32 bit x86' and 'Wampserver 3.2.3 64 bit x64', both with Apache 2.4.46 and PHP 5.6.40/7.3.21/7.4.9. The MySQL version is 5.7.31. MariaDB versions 10.3.23, 10.4.13, and 10.5.4 are also listed. A note at the bottom states: 'Also includes PhpMyAdmin 5.0.2 - Adminer 4.7.7 - PhpSysInfo 3.3.2. The default database manager is MySQL, see wamp(64)/mariadb_mysql.txt for details. - As an option: PHP 7.0.33/7.1.33/7.2.33'.

In de downloadmap dubbelklik je op het uitvoeringsbestand en start je de installatie: volgt de schermafbeeldingen hieronder.









PHPmyadmin

De database omgeving waarin wij zullen werken is PHPmyadmin. We zullen dus de MySQL taal rechtstreeks op verschillende databases uitvoeren. Bij manipulatie van een database kan je dus met deze taal acties uitvoeren die dikwijls niet omkeerbaar zijn. Bijvoorbeeld het verwijderen van een tabel in een database of het wijzigen van gebruikers, Heden ten dage draait alles om data die zeer waardevol is voor verschillende mogelijke partijen, waaronder personen met minder goede bedoelingen zoals hackers. Wanneer je bijvoorbeeld later een carrière als ethisch hacker wil uitoefenen, dan dien je dus de SQL-taal al zeer goed te kennen en de omgevingen waarin deze taal wordt uitgevoerd.

Openen



Na de installatie van de wampserver zal je dit icoontje terugvinden op je bureaublad.

Dubbelklik op dit icoontje op zowel de apache webserver te starten alsook MySQL. Je pc fungeert nu eigenlijk als een lokale webserver die NIET geconnecteerd is met het web. Een reëele webserver is WEL geconnecteerd met het web.

Webservers voorzien dus enerzijds hosting om je website of webapplicatie op te laden alsook de PHPmyadmin databaseomgeving anderzijds.

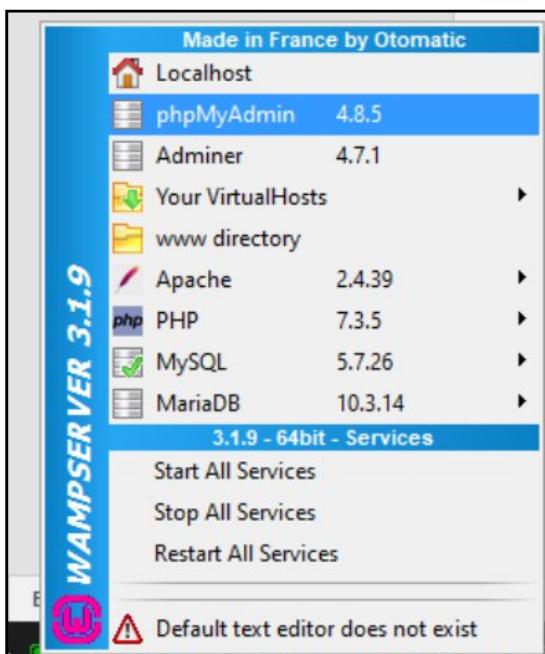
Als developer zullen we dus onze webapplicaties eerst lokaal maken alvorens deze op te laden naar een actieve hosting.



Wanneer je alles correct geïnstalleerd hebt en alles opgestart is, zul je rechts onderaan in je taakbalk hetzelfde icoontje zien staan die GROEN dient te zijn. Bij een groen icoontje werd alles correct gestart.

Bij een ORANJE icoontje werd ofwel de apache webserver of MySQL niet goed opgestart.

Bij een ROOD icoontje werd niets opgestart.



Klik nu met je linker muisknop op dit GROENE icoontje en klik vervolgens op PHPmyadmin om de database omgeving te openen. Deze omgeving zal jullie speelplaats worden tijdens deze cursus. Naargelang de evolutie van alle programma's kunnen de versies die in de schermafbeeldingen te zien zijn verschillen van deze cursus.

CURSUS MySQL

In het volgende scherm zal je nu kunnen inloggen. In de lokale ontwikkelomgeving dienen we geen speciale username of password te gebruiken. Standaard loggen we op PHPmyadmin in met de user **root** en laten we password **blanco** staan. Klik vervolgens op **Go**.

Vanzelfsprekend wanneer we onze afgewerkte database zouden opladen naar een reëele webserver zal dit aangepast worden. Het zou bijzonder leuk zijn voor personen met slechte bedoelingen moest dit ook online op deze manier toegankelijk zijn.

The screenshot shows the initial login screen of phpMyAdmin. At the top is the logo 'phpMyAdmin' with a sailboat icon. Below it is the heading 'Welcome to phpMyAdmin'. On the left, there's a 'Language' dropdown set to 'English'. The main area is titled 'Log in' with a question mark icon. It has three input fields: 'Username' containing 'root', 'Password' which is empty, and 'Server Choice' set to 'MySQL'. At the bottom right of this area is a 'Go' button.

Na het inloggen kom je terecht in de standaard omgeving van PHPmyadmin. Hier kan je verschillende databases aanmaken en/of bevragen. Het belangrijkste in onderstaand scherm is de positionering nl. **Server:MySQL:3306**. d.w.z. dat je nu op het hoogste niveau van de MySQL Server bevindt.

The screenshot shows the dashboard of phpMyAdmin after logging in. The title bar displays 'Server: MySQL_3306'. Below the title bar is a horizontal menu with icons and text for 'Databases', 'SQL', 'Status', 'User accounts', 'Export', and 'Import'. The main content area is titled 'General settings'. A blue arrow points downwards from the 'MySQL_3306' text in the title bar towards the 'General settings' section.

CURSUS MySQL

Er dient ook een woordje uitleg te worden gegeven over het onderstaande scherm. Hier zie je dat de **Web Server** die momenteel draaiende is dankzij wamp, de **Apache webserver** is.

De **Database Server** die we momenteel via PHPmyadmin bekijken is **MySQL**.

PHPmyadmin is de huidige omgeving waarin **MySQL** momenteel draait.

Dit kan dus evengoed een andere omgeving zijn zoals mongoDB, mariaDB die eveneens MySQL draaien, maar worden hier niet besproken.

Database server

- Server: MySQL (127.0.0.1 via TCP/IP)
- Server type: MySQL
- Server connection: SSL is not being used ⓘ
- Server version: 5.7.26 - MySQL Community Server (GPL)
- Protocol version: 10
- User: root@localhost
- Server charset: UTF-8 Unicode (utf8)

Web server

- Apache/2.4.39 (Win64) PHP/7.3.5
- Database client version: libmysql - mysqlnd 5.0.12-dev - 20150407 - \$Id: 7cc7cc96e675f6d72e5cf0f267f48e167c2abb23 \$
- PHP extension: mysqli ⓘ curl ⓘ mbstring ⓘ
- PHP version: 7.3.5

phpMyAdmin

- Version information: 4.8.5, latest stable version: 4.9.5
- Documentation
- Official Homepage
- Contribute
- Get support
- List of changes
- License

Databases

Nieuw

Een nieuwe database creëren via PHPmyadmin is eenvoudig. Aan de linkerkant zie je een menu die alle databases zal bevatten. Je kan dus oneindig veel databases op PHPmyadmin draaien voor verschillende webapplicaties. Klik op **New** om een database aan te maken.

Databases

Zoals je hierboven kan zien zie je de characterset latin_swedish_ci. Dit is de standaard. Deze characterset bestaat uit 8 bits in maximale grootte. Standaard is dit meestal meer dan genoeg om tekens (letters, cijfers, speciale tekens) te bewaren.

In deze characterset zitten natuurlijk alle tekens die ook in de ASCII tabel zitten en meer.

Waarom wordt dan de ASCII tabel niet gebruikt? ASCII telde initieel 127 symbolen die konden bewaard worden in een 7-bits karakterset. In een 8-bits karakterset kunnen er dus meer symbolen worden gebruikt.

Voor websites zitten we tegenwoordig aan 16-bits in grootte. De characterset die daar zal worden gebruikt is de huidige standaard UTF-8.

In de meeste gevallen heb je echter genoeg met de swedisch karakterset. Wil je zeker zijn dan kan je de characterset hier ook aanpassen naar UTF-8.

Kies nu **utf8_general_ci** als characterset en druk **create**.

De database werd nu aangemaakt. De engine die wordt gebruikt om de database te bevragen in PHPmyadmin is standaard MyISAM. Deze heeft als nadeel dat we geen relaties kunnen leggen tussen tabellen in deze omgeving. We dienen deze om te zetten naar de andere engine van PHPmyadmin, nl. InnoDB

Engine InnoDB

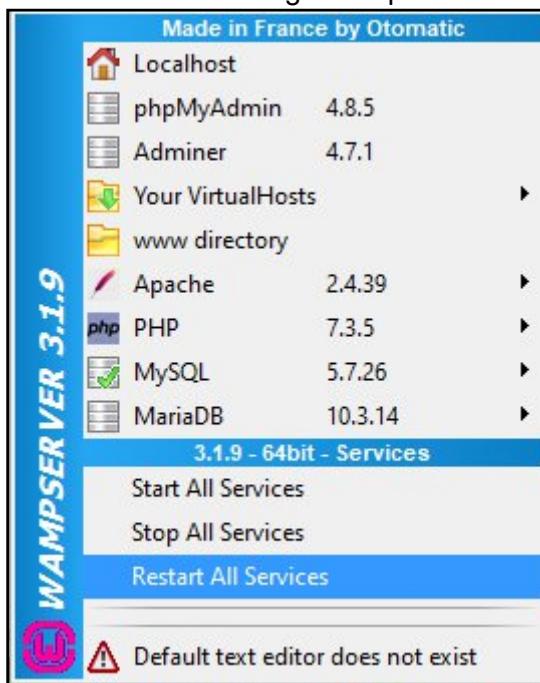
Om de engine te wijzigen dienen we het bestand my.ini te openen op onderstaande locatie (zie schermafbeelding).

Name	Date modified	Type	Size
bin	9/28/2019 2:49 PM	File folder	
data	9/23/2020 9:03 PM	File folder	
docs	9/28/2019 2:49 PM	File folder	
include	9/28/2019 2:49 PM	File folder	
lib	9/28/2019 2:49 PM	File folder	
share	9/28/2019 2:49 PM	File folder	
COPYING	4/13/2019 3:32 PM	File	18 KB
my.ini	9/23/2020 9:03 PM	Configuration sett...	7 KB
README	4/13/2019 3:32 PM	File	3 KB
wampserver.conf	12/11/2015 11:55 AM	CONF File	1 KB

In my.ini zoek je achter de myISAM engine en wijzig je deze naar InnoDB.

```
; The default storage engine that will be used when create new tables
default-storage-engine=InnoDB
; New for MySQL 5.6 default_tmp_storage_engine if skip-innodb enable
; default_tmp_storage_engine=MYISAM
```

Herstart nu de volledige wampserver als volgt om de engine in te laden:



Verwijderen

Databases kunnen we ook verwijderen uit PHPmyadmin.

- klik bovenaan op **Server:MySQL:3306**
- klik vervolgens op het tabblad **Databases**
 - wagens
 - Selecteer de te verwijderen database
 - Klik op **Drop** en druk **OK**

Sample Databases

Voor MySQL werden er enkel databases aangemaakt speciaal voor studenten die de taal wensen onder de knie te krijgen.

Link:<https://dev.MySQL.com/doc/index-other.html>

De databases die o.a. worden gebruikt zijn employee, world, sakila en menagerie.

Wij zullen de **sakila** database gebruiken, maar eerst installeren. **Download** hier het **Zip** bestand van de database. We pakken deze uit op de c:\ schijf.

Example Databases			
Title	Download DB	HTML Setup Guide	PDF Setup Guide
employee data (large dataset, includes data and test/verification suite)	GitHub	View	US Ltr A4
world database	Gzip Zip	View	US Ltr A4
world_x database	TGZ Zip	View	US Ltr A4
sakila database	TGZ Zip	View	US Ltr A4
menagerie database	TGZ Zip		

Installatie SAKILA database

Hiervoor zullen we de **HTML Setup Guide** gebruiken van de sakila database. Druk dan op **Installation**.

Sakila Sample Database

Table of Contents

- 1 Preface and Legal Notices
- 2 Introduction
- 3 History
- 4 Installation
- 5 Structure
- 6 Usage Examples
- 7 Known Issues
- 8 Acknowledgments
- 9 License for the Sakila Sample Database
- 10 Note for Authors
- 11 Sakila Change History

CURSUS MySQL

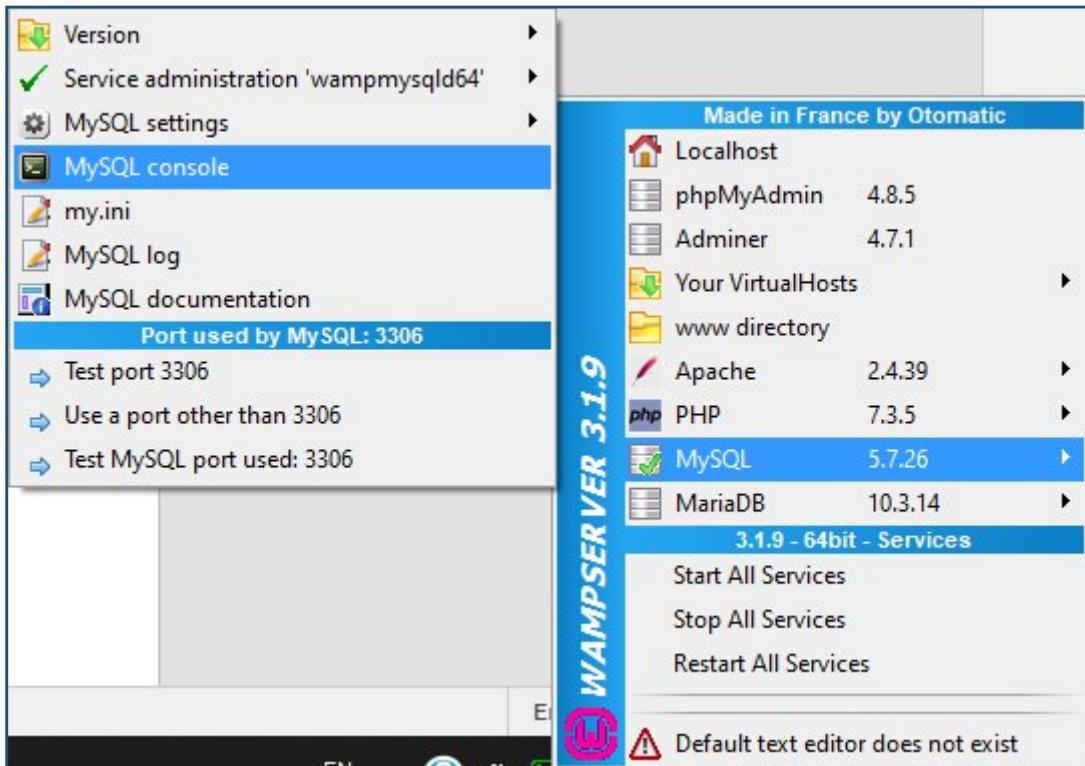
In de volgende pagina zie je nu enkele commando's die we in een console dienen in te geven:

To install the Sakila sample database, follow these steps:

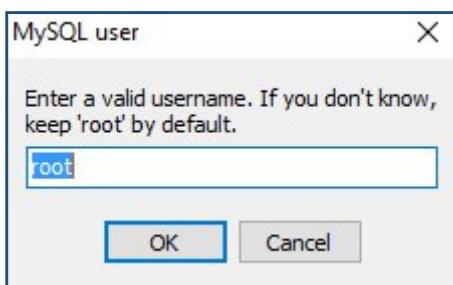
1. Extract the installation archive to a temporary location such as c:\temp\ or /tmp/. When you unpack the `sakila-schema.sql` and `sakila-data.sql` files.
2. Connect to the MySQL server using the `mysql` command-line client with the following command:

```
1 shell> mysql -u root -p
```

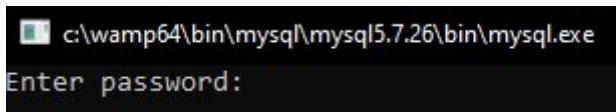
De console die we hiervoor gaan gebruiken is de MySQL console die je in de schermafbeelding hieronder ziet staan.



De standaard username van PHPmyadmin zal worden gevraagd. Druk hier op **OK**.



Het paswoord die je dient in te vullen is **blanco**. Druk dus gewoon op **ENTER**.



CURSUS MySQL

Wanneer alles correct werd uitgevoerd zit je in de console van MySQL.

```
c:\wamp64\bin\mysql\mysql5.7.26\bin\mysql.exe
Enter password:
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 58
Server version: 5.7.26 MySQL Community Server (GPL)

Copyright (c) 2000, 2019, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

Nu dienen we het de sakila database die we uitgepakt hebben op de c schijf te installeren. Eerst wordt de structuur (tabellen) geïnstalleerd met het volgende commando. Let op dat de locatie juist is.

```
mysql> SOURCE C:/sakila-db/sakila-schema.sql
```

Vervolgens gaan we de data ook in de tabellen injecteren met het volgende commando.

```
mysql> SOURCE C:/sakila-db/sakila-data.sql
```

Opdracht

Probeer ook de world database te installeren.

Sluit nu de consoletoepassing af en open terug PHPmyadmin.

Resultaat:

Table	Action	Rows	Type	Collation	Size	Overhead
actor		200	InnoDB	utf8mb4_general_ci	32 Kib	-
actor_info		~0	View	---	-	-
address		603	InnoDB	utf8mb4_general_ci	112 Kib	-
category		16	InnoDB	utf8mb4_general_ci	16 Kib	-
city		600	InnoDB	utf8mb4_general_ci	64 Kib	-
country		109	InnoDB	utf8mb4_general_ci	16 Kib	-
customer		599	InnoDB	utf8mb4_general_ci	128 Kib	-
customer_list		~0	View	---	-	-
film		1,000	InnoDB	utf8mb4_general_ci	272 Kib	-
film_actor		5,462	InnoDB	utf8mb4_general_ci	272 Kib	-
film_category		1,000	InnoDB	utf8mb4_general_ci	80 Kib	-
film_list		~0	View	---	-	-
film_text		1,000	InnoDB	utf8mb4_general_ci	192 Kib	-
inventory		4,581	InnoDB	utf8mb4_general_ci	368 Kib	-
language		6	InnoDB	utf8mb4_general_ci	16 Kib	-
nicer_but_slower_film_list		~0	View	---	-	-
payment		16,049	InnoDB	utf8mb4_general_ci	2.1 MiB	-
rental		16,044	InnoDB	utf8mb4_general_ci	80 Kib	-
sales_by_film_category		~0	View	---	-	-
sales_by_store		~0	View	---	-	-
staff		2	InnoDB	utf8mb4_general_ci	48 Kib	-
staff_list		~0	View	---	-	-
store		2	InnoDB	utf8mb4_general_ci	48 Kib	-

Tabellen

Een database bestaat uit genormaliseerde tabellen. Normalisatie is een onderwerp die we later zullen bekijken. Het komt er op neer dat iedere tabel zijn eigen specifieke data heeft en gerelateerd kan zijn met andere tabellen in de database.

Voorbeeld:

De tabel met gebruikers kan gerelateerd zijn met de tabel van adressen.

Een gebruiker kan dus één of meerdere adressen hebben.

Datatypes

In een tabel bestaat elk **veld (kolom)** uit een bepaald **datatype**. Het datatype bepaalt welke soort gegevens in dat veld kunnen worden opgeslagen: getallen, tekst, datums, enz. Het correct kiezen van een datatype is essentieel voor **prestaties, opslagruimte en datakwaliteit**.

1. Numerieke datatypes

Type	Bereik (signed)	Bereik (unsigned)	Typische toepassing
TINYINT	-128 tot 127	0 tot 255	Leeftijd, kleine codes
SMALLINT	-32.768 tot 32.767	0 tot 65.535	Jaar, aantal stuks
MEDIUMINT	-8,3 miljoen tot 8,3 miljoen	0 tot 16,7 miljoen	Populatie, telling
INT	-2,1 miljard tot 2,1 miljard	0 tot 4,2 miljard	ID's, primaire sleutels
BIGINT	- 2^{63} tot $2^{63} - 1$	0 tot $2^{64} - 1$	Grote tellingen, financiële transacties

Gebruik UNSIGNED wanneer negatieve waarden onmogelijk zijn.

2. Decimale en floating-point getallen

Type	Precisie	Typische toepassing
FLOAT	7 cijfers (benadering, single precision)	Wetenschappelijke berekeningen
DOUBLE	15–16 cijfers (benadering, double precision)	Statistiek, meetwaarden
DECIMAL	Tot 65 cijfers voor de komma en 30 erna (exact)	Geldbedragen, prijzen

Belangrijk:

- FLOAT en DOUBLE zijn **benaderingen** → kans op afrondingsfouten.
- DECIMAL is exact → altijd gebruiken voor **geld en facturen**.

3. Tekst- en stringdatatypes

Type	Eigenschappen	Typische toepassing
CHAR(n)	Vaste lengte, max. 255 tekens	Postcode, landcode
VARCHAR(n)	Variabele lengte, max. 65.535 tekens (afhankelijk van charset en opslag)	Namen, e-mails
TEXT	Lange tekst (tot 4 GB)	Artikels, beschrijvingen
ENUM	Eén waarde uit een vaste lijst	Status: 'active', 'inactive'
BLOB	Binaire data, max. 4 GB	Afbeeldingen, audio, bestanden

Gebruik CHAR alleen voor velden met **altijd vaste lengte** (bv. landcode: "BE"). Voor alles wat variabel is, gebruik je VARCHAR.

4. Datum- en tijdsdatatype

Type	Formaat	Voorbeeld	Typische toepassing
DATE	YYYY-MM-DD	2025-09-29	Geboortedatum
TIME	hh:mm:ss	14:35:00	Tijdstippen
DATETIME	YYYY-MM-DD hh:mm:ss	2025-09-29 14:35:00	Geboortedatum + tijd
TIMESTAMP	YYYY-MM-DD hh:mm:ss	2025-09-29 14:35:00	Automatische logtijden
YEAR	YYYY	2025	Kalenderjaar

Verschil **DATETIME** vs **TIMESTAMP**:

- DATETIME → onafhankelijk van tijdzone.
- TIMESTAMP → wordt automatisch aangepast aan de ingestelde tijdzone en kan zichzelf updaten bij wijzigingen.

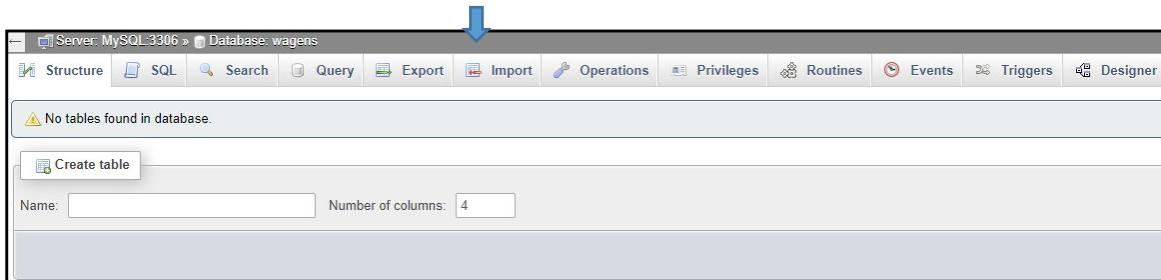
5. Praktische keuzes

In de praktijk gebruik je meestal:

- **INT UNSIGNED** → primaire sleutels (id).
- **DECIMAL(10,2)** → geld (10 cijfers totaal, 2 na de komma).
- **VARCHAR(255)** → tekst zoals namen, e-mails.
- **TEXT** → langere beschrijvingen.
- **DATETIME** of **TIMESTAMP** → wanneer je ook tijd nodig hebt.

Nieuw

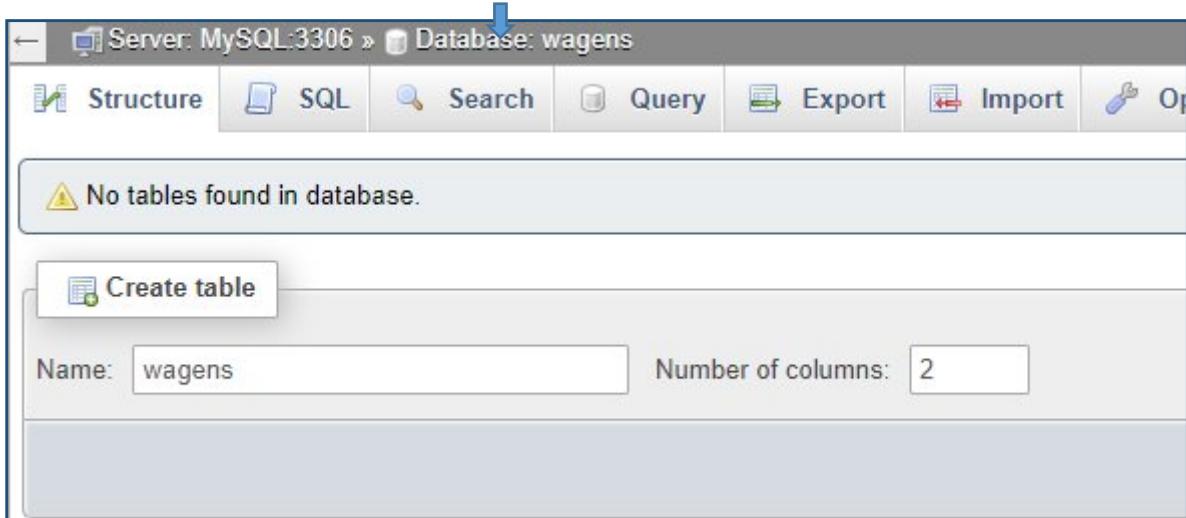
Een nieuwe tabel aanmaken is eenvoudig. Je dient eerst je database te selecteren. Je kan steeds controleren of je in de correcte database zit bovenaan PHPmyadmin. Net zoals in windows verkenner is dit een folder structuur die vertrekt van het server niveau.



In ons voorbeeld hebben we het over wagens. Wagens bestaan uit merken en modellen. Hiervoor creëren we dus 2 tabellen.

De eerste tabel wagens zal 2 velden bevatten: id, merk.

Vul de **naam** van de tabel in en duid hiervoor **2** kolommen aan voor het aantal velden. Klik dan op **Go**.



Conventie bij het schrijven van veldnamen: ALLEMAAL KLEINE LETTERS! GEEN CAMELCASE zoals variabelen in PROGRAMMEERTALEN.

Vul in zoals onderstaande afbeelding en klik op **SAVE**. Je hebt nu een lege tabel zonder data aangemaakt in de database van wagens.

Table name: wagens									
	Add	1	column(s)	Go	Structure	Attributes	Null Index	A_	Comments
Name	Type	Length/Values	Default	Collation	Attributes	Null Index	A_	Comments	Virtuality
id	INT	11	None		UNSIGNED	<input type="checkbox"/>	PRIMARY	PRIMARY	
merk	VARCHAR	255	None			<input type="checkbox"/>			
Table comments:				Storage Engine:					
				InnoDB					
PARTITION definition:									
Partition by:		(Expression or column list)							
Partitions:									
<input type="button" value="Preview SQL"/> <input type="button" value="Save"/>									

Veld instellingen

Hier zullen we kort alle mogelijke instellingen bespreken tijdens het maken van velden in een tabel.

Name

Hier worden de kolomkoppen of veldnamen ingegeven. De eerste **veldnaam** die wij bijna altijd zullen invullen is het **id** van de tabel. Het **id** is ook meestal een oplopende nummering die start van 1 met een **auto_increment**. Auto_increment zorgt dus m.a.w. automatisch dat een volgend **record** automatisch verder wordt genummerd. Dit id is **UNIEK** en wordt de **PRIMARY KEY** van een tabel genoemd.

Een **record** is één enkele data-lijn in de database met ingevulde data per veld.

Type

Het type per veld is het datatype die we dienen te bepalen. Aangezien een id een oplopende nummering is zal dit van het type integer zijn. Hier kiezen we **INT**.

Een VARCHAR kan bijvoorbeeld gebruikt worden voor een string, DATETIME voor een datum, ...

Length/values

Aangezien het datatype bijvoorbeeld INT is kunnen we tot ca 2 biljoen in cijfers laten oplopen.

Dankzij lengte beperken we het aantal karakters toch. Veelal wordt er bij een id 11 karakters aan lengte toegevoegd die voor de meeste database meer dan voldoende is.

Het laatste id zou dus in principe 99999999999 kunnen zijn.

Default values

Hiermee kan je een standaard waarde meegeven wanneer een veld niet zou worden ingevuld door een gebruiker.

Er zijn 4 opties:

none = is default en blijft dus leeg

NULL = hier wordt een nullable in het veld geschreven. Je zal dan het woord NULL zien staan in het veld.

As defined: hier verschijnt een extra veld waar je zelf een default waarde kan schrijven. Deze default waarde wordt in het veld van de database weggeschreven wanneer de gebruiker niets zou hebben geschreven.

Current_time_stamp = Zal de datumtijdstempel wegschrijven van de pc.

Collation

Voor ieder veld kan je een aparte character set toevoegen indien je dit nodig zou hebben. In de meeste gevallen is de initiële character set van de database voldoende en blijft dit veld leeg.

Attributes

BINARY

De opslag van een ingetikte waarde zal binair gebeuren (1 en 0)

UNSIGNED

Een unsigned value kan enkel positieve getallen opslaan in de database.

Zie tabel datatypes (voorgaand).

UNSIGNED ZERO FILL

Zero fill zorgt ervoor dat alle ontbrekende characters met een 0 worden opgevuld.

Bijvoorbeeld: id(11)

id(11) kan een lengte van 11 getallen opslaan. Wanneer we nu het getal 1 opslaan dan wordt dit met zero fill: 00000000001

on update CURRENT_TIMESTAMP

Wanneer er een veld van het datatype datetime wordt gebruikt dan kunnen we hier ervoor zorgen dat automatisch de huidige datumtijdstempel van de pc wordt gebruikt wanneer een gebruiker een wijziging aanbrengt op dit record.

Null

Het veld mag leeg zijn wanneer aangevinkt.

Index

PRIMARY

Met deze optie kan je een veld de primaire sleutel maken van je tabel (PK = Primary KEY)

In combinatie met A-I (auto increment), maak je een primaire sleutel dus aan die daarnaast ook uniek is

UNIQUE

Wanneer je een veld UNIQUE zet, dan zal je dubbele waarden vermijden in een database. Dit veld kan wel een NULL value bevatten.

INDEX

MySQL gebruikt indexen om snel rijen met specifieke kolomwaarden te vinden. Zonder index moet MySQL de hele tabel scannen om de relevante rijen te lokaliseren. Hoe groter de tafel, hoe langzamer hij zoekt.

A_I

Wanneer aangevinkt zorgt A_I voor een oplopende nummering van een veld met een datatype integer.

COMMENTS

Hier kan je als developer een beschrijving geven van de bedoeling van een veld.

Code: CREATE TABLE

Een tabel kan je ook met code aanmaken. We maken dus hier gebruik van de MySQL-taal.

Oefening

Voorbeeld:

```
CREATE TABLE Klanten (
    id INT(6) UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    voornaam VARCHAR(30) NOT NULL,
    familienaam VARCHAR(30) NOT NULL,
    email VARCHAR(50),
    reg_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
)
```

Na het gegevenstype kunt u voor elke kolom andere optionele attributen specificeren:

- NOT NULL - Elke rij moet een waarde voor die kolom bevatten, null-waarden zijn niet toegestaan
- DEFAULT-waarde - Stel een standaardwaarde in die wordt toegevoegd als er geen andere waarde wordt doorgegeven
- UNSIGNED - Gebruikt voor number datatypes , beperkt de opgeslagen gegevens tot positieve getallen en nul
- AUTO INCREMENT - MySQL verhoogt automatisch de waarde van het veld met 1 telkens wanneer een nieuw record wordt toegevoegd
- PRIMARY KEY - Wordt gebruikt om de rijen in een tabel uniek te identificeren. De kolom met PRIMARY KEY-instelling is vaak een ID-nummer en wordt vaak gebruikt met AUTO_INCREMENT

Opdracht

Maak een nieuwe tabel **modellen** aan in de tabel wagens. Zorg dat er 3 velden aanwezig zijn. Het eerste veld **id** is een integer die positieve primaire sleutel is met autonummering en een grootte van 11 heeft. Het tweede veld is **merkid** en is een positieve integer met een grootte van 11 en kan geen null bevatten. Het derde veld is naam en zal alle modelnamen bevatten. Dit is een string met een grootte van 255 en kan geen null bevatten.

Oplossing:

```
CREATE TABLE merken (
    id INT(11) UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    merkid INT(11) UNSIGNED NOT NULL,
    naam VARCHAR(255) NOT NULL
)
```

#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra	Action
1	id	int(11)		UNSIGNED	No	None		AUTO_INCREMENT	Change Drop ▾ More
2	merkid	int(11)		UNSIGNED	No	None			Change Drop ▾ More
3	naam	varchar(255)	utf8_general_ci		No	None			Change Drop ▾ More

Momenteel hebben we dus een database met de naam wagens. In deze database hebben we 2 tabellen, nl. merken en modellen. Wat we nog NIET hebben is een relatie tussen deze 2 tabellen.

Relaties

We sommen eerst de meest voorkomende relaties op die mogelijk zijn in een database tussen 2 of meerdere tabellen.

- één-op-één
- één-op-veel
- veel-op-veel

Relatie: één-op-veel

In ons voorbeeld hebben we bijvoorbeeld een model: AUDI
Audi heeft meerdere modellen in zijn gamma: A3, A4, A5,...

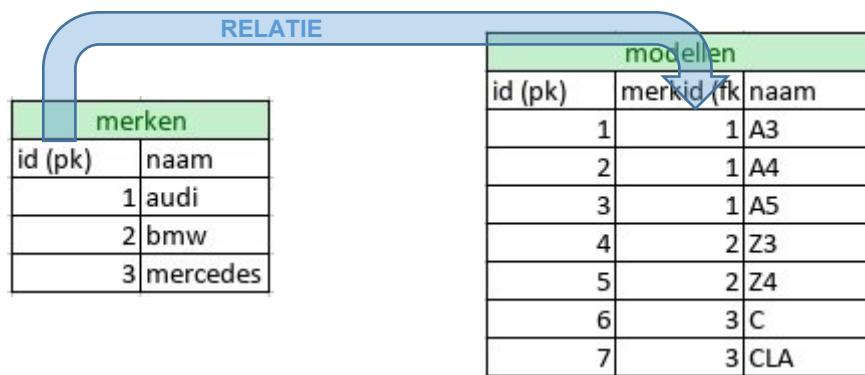
Dit noemen we een één-op-veel-relatie, want het merk audi (één) heeft meerdere modellen (veel).

Een relatie dienen we dus ook te definiëren. Momenteel hebben we reeds alle velden in de tabellen gedefinieerd, maar nog niet de relatie.

Wanneer we de tabellen even schematisch bekijken in de designer tab op het niveau van de database dan kan je dit zien. Er is duidelijk geen link dus deze 2 tabellen. Deze lijken te ‘zweven’ naast elkaar

wagens merken	wagens modellen
id : int(11) unsigned	id : int(11) unsigned
wagens : varchar(255)	merkid : int(11) unsigned

Hieronder zie je wat de bedoeling zou moeten zijn in een **één-op-veel** relatie datagewijs.



Zoals je kan zien hebben beide tabellen een unieke primary key. Het merkid echter is een sleutel die vreemd is aan de tabel modellen en die werd geërfd van de tabel merken. Deze sleutel noemen we een FOREIGN KEY.

Om nu de relatie te bewerkstelligen in PHPmyadmin dienen we de **TWEEDE** tabel modellen te selecteren en het tabblad **STRUCTURE** aan te klikken. Klik vervolgens op **RELATION VIEW**.

The screenshot shows the 'Relation view' tab selected in the 'modellen' table's structure view. It displays a 'Foreign key constraints' section where a constraint named 'model_merken' is being configured. The configuration shows 'ON DELETE RESTRICT' and 'ON UPDATE RESTRICT'. The 'Column' dropdown for the foreign key is set to 'merkid', the 'Database' to 'wagens', the 'Table' to 'merken', and the 'Column' to 'id'.

De opties die mogelijk zijn bij “on delete en on update” zijn:

RESTRICT

Wanneer je data die verbonden is in 2 tabellen zou willen wissen, dan zal dit niet lukken.

Wanneer je bijvoorbeeld audi in de tabel merken zou willen zal de database een restriction error geven omdat je modellen van audi in de tabel modellen zitten hebt.

CASCADE

Cascade zal wel wissen. Wanneer je audi zou wissen in de tabel merken dan zullen al zijn modellen in de tabel modellen ook gewist worden.

SET NULL

Wanneer je audi zou wissen in de tabel merken dan zullen de rijen in de tabel modellen niet gewist worden maar zal de inhoud van de rij op NULL worden geplaatst.

Wanneer we nu kijken naar de tabel modellen dan zie je een grijze sleutel staan die de foreign key weergeeft. Een gouden sleutel is de primary key van de tabel zelf.

#	Name	Type	Collation	Attributes	Null	Def
1	id	int(11)		UNSIGNED	No	
2	merkid	int(11)		UNSIGNED	No	
3	naam	varchar(255)	utf8_general_ci		No	

Op het database niveau kan je terug naar het designer tabblad gaan en zie je het uiteindelijke resultaat van een één-op-veel relatie.



Code: FOREIGN KEY CONSTRAINTS

Wanneer je alle bovenstaande acties in code zou willen uitvoeren dan kan dit natuurlijk ook. WIS de tabel modellen uit de database.

Open nu het SQL tabblad op database niveau en voeg onderstaande code toe:

```

CREATE TABLE modellen (
    id INT(11) UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    merkid INT(11) UNSIGNED NOT NULL,
    naam VARCHAR(255) NOT NULL,
    FOREIGN KEY (merkid) REFERENCES merken(id)
        ON UPDATE RESTRICT ON DELETE RESTRICT
)
    
```

Bovenstaande geeft hetzelfde resultaat terug en maakt de tabel modellen aan met de foreign key en zijn restriction.

CRUD

CRUD = CREATE READ UPDATE DELETE

Wanneer we over CRUD spreken, spreken we over alle mogelijk manipulaties die we op één of meerdere tabellen van een database kunnen uitvoeren.

De SQL syntax is de taal die we nodig zullen hebben om deze handelingen/manipulaties uit te voeren.

SELECT STATEMENT

We starten eerst met het READ gedeelte van de CRUD. Deze wordt in SQL uitgevoerd door het SELECT STATEMENT

Hieronder zie je de voorstelling van het volledige select statement die gebruikt KAN worden.

```
SELECT field_references
      [ALL | DISTINCT | DISTINCTROW ]
      [HIGH_PRIORITY]
      [STRAIGHT_JOIN]
      [SQL_SMALL_RESULT] [SQL_BIG_RESULT] [SQL_BUFFER_RESULT]
      [SQL_CACHE | SQL_NO_CACHE] [SQL_CALC_FOUND_ROWS]
      select_expr [, select_expr] ...
      [into_option]
      [FROM table_references
          [PARTITION partition_list]]
      [WHERE where_condition]
      [GROUP BY {col_name | expr | position}
          [ASC | DESC], ... [WITH ROLLUP]]
      [HAVING where_condition]
      [ORDER BY {col_name | expr | position}
          [ASC | DESC], ...]
      [LIMIT {[offset,] row_count | row_count OFFSET offset}]
```

We hebben de SAKILA database samen geïnstalleerd. Deze gaan we nu gebruiken om het select statement in detail aan te leren.

OEFENINGEN

BASIS SELECT EN/OF ORDER BY



ORDER BY: zorgt voor het alfabetisch sorteren van velden. Opties zijn ASC en DESC
DISTINCT: zorgt dat een VOLLEDIG record (rij) UNIEK is in het resultaat.

- Selecteer alle velden uit de tabel actor

```
SELECT *
```

```
FROM actor
```

- Selecteer enkel familienaam en voornaam uit de tabel actor

```
SELECT actor.last_name, actor.first_name
```

```
FROM actor
```

- Selecteer enkel familienaam en voornaam uit de tabel actor en sorteert deze eerst volgens familienaam (A-Z) en vervolgens volgens voornaam (A-Z)

```
SELECT actor.last_name, actor.first_name
```

```
FROM actor
```

```
ORDER BY actor.last_name ASC, actor.first_name DESC
```

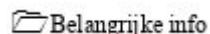
- Selecteer enkel de unieke familienaam en voornaam uit de tabel actor en sorteert Z-A

```
SELECT DISTINCT actor.last_name, actor.first_name
```

```
FROM actor
```

```
ORDER BY actor.last_name DESC, actor.first_name DESC
```

WHERE CLAUSE



LOGISCHE OPERATOREN: and, or, IN, BETWEEN

- Selecteer alle velden uit de tabel actor waar de familienaam gelijk is aan 'ZELLWEGER'

```
SELECT *
```

```
FROM actor
```

```
WHERE actor.last_name = 'ZELLWEGER'
```

- Selecteer alle velden uit de tabel actor waar de familienaam gelijk is aan 'ZELLWEGER' of 'GUINNESS'

oplossing 1:

```
SELECT *
```

```
FROM actor
```

```
WHERE actor.last_name = 'ZELLWEGER' or actor.last_name = 'GUINNESS'
```

oplossing 2:

```
SELECT *
```

```
FROM actor
```

```
WHERE actor.last_name IN ('ZELLWEGER', 'GUINNESS')
```

- Selecteer alle velden uit de tabel actor waar de familienaam begint met Z of met D

```
SELECT *
FROM actor
WHERE actor.last_name LIKE ('Z%') OR actor.last_name LIKE ('D%')
```

- Selecteer alle velden uit de tabel actor waar de familienaam DAVIS, DUKAKIS of ZELLWEGER is.

```
SELECT *
FROM actor
WHERE actor.last_name IN ('DAVIS', 'DUKAKIS', 'ZELLWEGER')
```

- Selecteer alle acteurs waar het id groter of gelijk is dan 5 en kleiner of gelijk is aan 10.

oplossing 1:

```
SELECT *
FROM actor
WHERE actor.actor_id >= 5 and actor.actor_id <= 10
```

oplossing 2:

```
SELECT *
FROM actor
WHERE actor.actor_id BETWEEN 5 and 10
```

FUNCTIONS

Belangrijke info

AS: het keyword AS wordt gebruikt om een veld een anderen naam te geven in een weergave of een nieuw samengesteld veld een naam te geven. AS is dus wat noemt een **alias** van een bestaand of nieuw veld.

In MySQL kunnen we daarnaast ook functies toepassen. Functies worden toegepast op bestaande velden in. Functies worden gekenmerkt door de naamgeving en ronde haakjes. Bijvoorbeeld: de functie MIN() haalt de laagste waarde uit een kolom.

Hieronder zie je een lijst van veel gebruikte functies

COUNT	De MySQL COUNT-aggregatiefunctie wordt gebruikt om het aantal rijen in een databasetabel te tellen.
MAX	Met de MySQL MAX-aggregatiefunctie kunnen we de hoogste (maximale) waarde voor een bepaalde kolom selecteren.
MIN	Met de MySQL MIN-aggregatiefunctie kunnen we de laagste (minimum) waarde voor een bepaalde kolom selecteren.
AVG	De MySQL AVG-aggregatiefunctie selecteert de gemiddelde waarde voor een bepaalde tabelkolom.
SUM	Met de MySQL SUM-aggregatiefunctie kunt u het totaal voor een numerieke kolom selecteren.
SQRT	Dit wordt gebruikt om een vierkantwortel van een bepaald getal te genereren.
RAND	Dit wordt gebruikt om een willekeurig getal te genereren met behulp van de MySQL-opdracht.
CONCAT	Dit wordt gebruikt om elke tekenreeks binnen een MySQL-opdracht samen te voegen .
LENGTH	Geeft de lengte van een string terug
CURDATE	Retourneert de huidige datum
CURTIME	Retourneert de huidige tijd
ADDDATE	Telt dagen bij een datum op
ADDTIME	Telt tijd bij tijd op
DATEFORMAT	Hier kan je de weergave van een datum mee wijzigen
DATE_SUB	Hier kan je 2 data van elkaar aftrekken
DATE	Geeft de datum van vandaag terug
WEEKDAY	Retourneert het weekdag nummer (index)
WEEK	Retourneert het weeknummer (index)

MySQL telt dus heel wat BUILT-IN functions. Er zijn er een paar honderd. Alle functies kun je hieronder terugvinden via de volgende link:

[MySQL :: MySQL 8.4 Reference Manual :: 14.1 Built-In Function and Operator Reference](#)

- Selecteer het hoogste bedrag uit de tabel payment
`SELECT max(amount)
 from payment`
- Hoeveel rijen telt de tabel payment
`SELECT max(amount)
 from payment`

- Selecteer de gemiddelde prijs van de bedragen uit de tabel payment
- ```
SELECT avg(amount)
FROM payment
```

## KЛАSSИКАLE OEFENINGEN

### 1. Film Titels in Hoofdletters

Schrijf een query die de titels van alle films in hoofdletters retourneert.

**Query:**

```
SELECT UPPER(title) AS film_titel
FROM film;
```

### 2. Lengte van de Film Titel

Haal de titels van alle films op en geef ook de lengte van elke titel weer.

**Query:**

```
SELECT title, LENGTH(title) AS titel_lengte
FROM film;
```

### 3. Film Titels zonder Spaties aan de Randen

Geef de film titels weer waarbij eventuele leidende en afsluitende spaties worden verwijderd.

**Query:**

```
SELECT TRIM(title) AS titel_bijgesneden
FROM film;
```

### 4. Films die een Bepaalde Tekst bevatten

Haal de titels van alle films op die het woord 'love' in hun titel hebben (ongeacht hoofdletters).

**Query:**

```
SELECT title
FROM film
WHERE LOWER(title) LIKE '%love%';
```

## 5. Eerste en Laatste Teken van Acteursnamen

Toon de voornaam van elke acteur, samen met het eerste en laatste teken van hun voornaam.

**Query:**

```
SELECT first_name, LEFT(first_name, 1) AS eerste_karakter, RIGHT(first_name, 1) AS laatste_karakter
FROM actor;
```

```
SELECT count(distinct last_name) as aantal
```

```
FROM actor
```

**GROUP BY**

 Belangrijke info

Een group by groepeert rijen/velden die identiek zijn aan elkaar tot één rij.

Let op: bij een GROUP BY dien je altijd ALLE VELDEN over te nemen die in het SELECT gedeelte staan ZONDER de velden die van een functie zouden voorzien zijn.

Voorbeeld van een group by :

ADAM JONES

ADAM JONES

BRIDGET SLOAN

DENNIS HARPER



ADAM JONES

BRIDGET SLOAN

DENNIS HARPER

- Selecteer alle familienamen uit de tabel actor en voeg daarna een tweede kolom toe die het aantal per naam weergeeft. Sorteer dan volgens het aantal van Z-A.

```
SELECT last_name, count(last_name) as aantal
FROM actor
GROUP BY last_name
ORDER BY aantal DESC
```

- Selecteer alle familienamen en voornamen uit de tabel actor en voeg daarna een tweede kolom toe die het aantal per naam weergeeft. Sorteer dan volgens het aantal van Z-A.

```
SELECT last_name, first_name, count(last_name) as aantal
FROM actor
GROUP BY last_name, first_name
ORDER BY aantal DESC
```

**HAVING**

 Belangrijke info

Een having clause wordt gebruikt wanneer we uit het resultaat van een group by dienen te FILTEREN. Een having is eigenlijk de where voor een GROUP BY

- Selecteer alle familienamen uit de actor tabel. Geef dan de lijst weer met familienamen die meer dan 1 x voorkomen!

**CURSUS MySQL**

```
select last_name
from actor
group by last_name
having count(*) > 1
```

## Oefening 1

**Vraag:** Selecteer alle verschillende rental\_rate waarden uit de film tabel.

```
SELECT DISTINCT rental_rate
FROM film;
```

## Oefening 2

**Vraag:** Toon alle actors met de achternaam SMITH of JONES.

```
SELECT *
FROM actor
WHERE last_name = 'SMITH' OR last_name = 'JONES';
```

## Oefening 3

**Vraag:** Selecteer de film\_id en title van alle films met een rental\_rate van meer dan 2.99, gesorteerd op title.

```
SELECT film_id, title
FROM film
WHERE rental_rate > 2.99
ORDER BY title;
```

## Oefening 4

**Vraag:** Tel het aantal films per rating en toon enkel diegenen met meer dan 100 films.

```
SELECT rating, COUNT(*) AS film_count
FROM film
GROUP BY rating
HAVING COUNT(*) > 100;
```

## Oefening 5

**Vraag:** Selecteer alle customer\_id waar de email NULL is.

```
SELECT customer_id
FROM customer
WHERE email IS NULL;
```

## Oefening 6

**Vraag:** Selecteer alle films (title) met een rental\_duration van 5 dagen en een replacement\_cost tussen 15 en 25.

```
SELECT title
FROM film
WHERE rental_duration = 5 AND replacement_cost BETWEEN 15 AND 25;
```

## Oefening 7

**Vraag:** Toon alle inventory\_id's waarvan de store\_id niet gelijk is aan 2.

```
SELECT inventory_id
FROM inventory
WHERE NOT store_id = 2;
```

## Oefening 8

**Vraag:** Selecteer de verschillende store\_id waarden uit de staff tabel.

```
SELECT DISTINCT store_id
FROM staff;
```

## Oefening 9

**Vraag:** Selecteer het aantal klanten (customer\_id) per store\_id.

```
SELECT store_id, COUNT(customer_id) AS customer_count
FROM customer
GROUP BY store_id;
```

## Oefening 10

**Vraag:** Toon alle film titels waarvan de replacement\_cost groter is dan 20, gesorteerd op aflopende replacement\_cost.

```
SELECT title
FROM film
WHERE replacement_cost > 20
ORDER BY replacement_cost DESC;
```

## Oefening 11

**Vraag:** Selecteer alle category\_id's van de category tabel waarvan de name niet Children is.

```
SELECT category_id
FROM category
WHERE NOT name = 'Children';
```

## Oefening 12

**Vraag:** Tel het aantal inventory\_id per film\_id en toon enkel resultaten met meer dan 20 exemplaren.

```
SELECT film_id, COUNT(inventory_id) AS inventory_count
FROM inventory
GROUP BY film_id
HAVING COUNT(inventory_id) > 20;
```

## Oefening 13

**Vraag:** Selecteer de actor\_id en first\_name voor acteurs met de achternaam die begint met K.

```
SELECT actor_id, first_name
FROM actor
WHERE last_name LIKE 'K%';
```

## Oefening 14

**Vraag:** Toon het totaal aantal films per rental\_duration en toon alleen de gevallen met een rental\_duration van meer dan 3 dagen.

```
SELECT rental_duration, COUNT(*) AS film_count
FROM film
GROUP BY rental_duration
HAVING rental_duration > 3;
```

## Oefening 15

**Vraag:** Selecteer alle rental\_id waarbij de return\_date nog niet is ingevuld (NULL).

```
SELECT rental_id
FROM rental
WHERE return_date IS NULL;
```

## Oefening 16

**Vraag:** Selecteer alle verschillende ratings uit de film tabel.

```
SELECT DISTINCT rating
FROM film;
```

## Oefening 17

**Vraag:** Tel het aantal klanten per store\_id waarbij active gelijk is aan 1.

```
SELECT store_id, COUNT(customer_id) AS active_customer_count
FROM customer
WHERE active = 1
GROUP BY store_id;
```

## Oefening 18

**Vraag:** Toon alle actor\_id's waarvan de last\_name niet gelijk is aan DAVIS of JOHNSON.

```
SELECT actor_id
FROM actor
WHERE last_name NOT IN ('DAVIS', 'JOHNSON');
```

## Oefening 19

**Vraag:** Selecteer alle payment\_id's waarbij de amount groter is dan 5 en de payment\_date is in 2005.

```
SELECT payment_id
FROM payment
WHERE amount > 5 AND payment_date LIKE '2005%';
```

## Oefening 20

**Vraag:** Toon de city\_id en tel het aantal steden in elke country\_id met meer dan 5 steden.

```
SELECT country_id, COUNT(city_id) AS city_count
FROM city
GROUP BY country_id
HAVING COUNT(city_id) > 5;
```

## Oefening 21

**Vraag:** Selecteer alle film\_id waarbij de description niet NULL is.

```
SELECT film_id
FROM film
WHERE description IS NOT NULL;
```

## Oefening 22

**Vraag:** Toon de store\_id en tel hoeveel inventory\_id's elke store heeft.

```
SELECT store_id, COUNT(inventory_id) AS inventory_count
FROM inventory
GROUP BY store_id;
```

## Oefening 23

**Vraag:** Selecteer alle actor\_id en last\_name waar de first\_name gelijk is aan NICK.

```
SELECT actor_id, last_name
FROM actor
WHERE first_name = 'NICK';
```

## Oefening 24

**Vraag:** Toon alle customer\_id waarbij create\_date gelijk is aan 2006-02-14.

```
SELECT customer_id
FROM customer
WHERE create_date = '2006-02-14';
```

## Oefening 25

**Vraag:** Tel het aantal verschillende film\_id's per store\_id in de inventory tabel.

```
SELECT store_id, COUNT(DISTINCT film_id) AS unique_film_count
FROM inventory
GROUP BY store_id;
```

## Oefening 26

**Vraag:** Selecteer de rental\_id en rental\_date voor alle verhuurtransacties in februari 2005.

```
SELECT rental_id, rental_date
FROM rental
WHERE rental_date LIKE '2005-02%';
```

## Oefening 27

**Vraag:** Tel het aantal address\_id per city\_id en toon enkel steden met meer dan 3 adressen.

```
SELECT city_id, COUNT(address_id) AS address_count
FROM address
GROUP BY city_id
HAVING COUNT(address_id) > 3;
```

## Oefening 28

**Vraag:** Selecteer alle customer\_id waar de klant niet actief is (active = 0).

```
SELECT customer_id
FROM customer
WHERE active = 0;
```

## Oefening 29

**Vraag:** Selecteer de title en film\_id voor films met een length tussen 90 en 120 minuten.

```
SELECT title, film_id
FROM film
WHERE length BETWEEN 90 AND 120;
```

## Oefening 30

**Vraag:** Toon alle verschillende store\_id waarden uit de customer tabel.

```
SELECT DISTINCT store_id
FROM customer;
```

## JOINS

### **Wat is een Join?**

Een **join** in MySQL stelt ons in staat om data uit meerdere tabellen te combineren. Dit is vooral handig als de tabellen op een bepaalde manier aan elkaar gerelateerd zijn via een primaire en vreemde sleutel (foreign key). Voor de wagens-database in jouw voorbeeld hebben we twee tabellen: `wagens merken` en `wagens modellen`.

- **wagens merken:** Bevat informatie over verschillende automerken zoals Audi, BMW, enz.
- **wagens modellen:** Bevat informatie over verschillende modellen die bij elk merk horen, met een `merkid` die aangeeft bij welk merk het model hoort.

De sleutelrelatie tussen de tabellen stelt ons in staat gegevens uit beide tabellen te combineren, zoals welk model bij welk merk hoort. Laten we de verschillende soorten joins bekijken die in MySQL beschikbaar zijn.

### **Types van Joins**

MySQL ondersteunt vier hoofdtypen van joins: 1. **INNER JOIN** 2. **LEFT JOIN** 3. **RIGHT JOIN** 4. **CROSS JOIN**

We zullen elk type join stap voor stap behandelen, met uitleg en voorbeelden van de query's en de resulterende output.

We vullen de 2 tabellen als volgt:

merken:

|                                                                 |  |  | <b>id</b> | <b>naam</b> |
|-----------------------------------------------------------------|--|--|-----------|-------------|
| <i>Click the drop-down arrow to toggle column's visibility.</i> |  |  |           |             |
| <input type="checkbox"/>                                        |  |  | 1         | audi        |
| <input type="checkbox"/>                                        |  |  | 2         | bmw         |
| <input type="checkbox"/>                                        |  |  | 3         | volkswagen  |

modellen:

|                          |  |  | <b>id</b> | <b>merkid</b> | <b>naam</b> |
|--------------------------|--|--|-----------|---------------|-------------|
| <input type="checkbox"/> |  |  | 1         | 1             | A3          |
| <input type="checkbox"/> |  |  | 2         | 1             | A4          |

## 1. INNER JOIN

Een **INNER JOIN** geeft alleen de rijen weer die in beide tabellen een overeenkomst hebben. Dit betekent dat alleen de merken die ook modellen hebben, worden weergegeven.

### Voorbeeld:

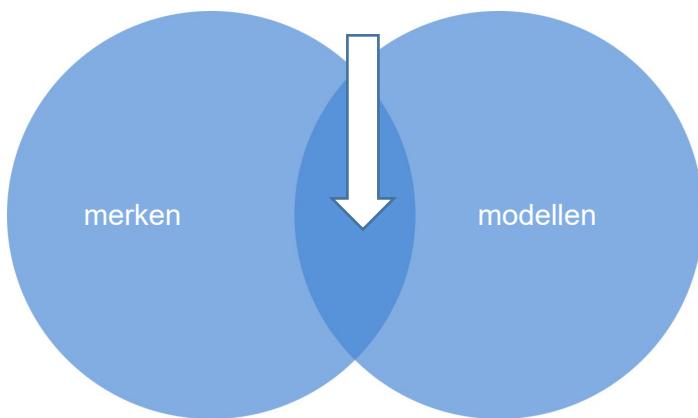
```
SELECT merken.naam, modellen.naam
FROM merken
INNER JOIN modellen ON merken.id = modellen.merkid;
```

- Hier verbinden we de tabel merken met de tabel modellen op basis van de `id` van merken en de `merkid` van modellen.
- Het resultaat laat alleen merken en hun bijbehorende modellen zien, zoals Audi met A3 en A4.

| naam | naam |
|------|------|
| audi | A3   |
| audi | A4   |

### Grafisch:

Een Venn-diagram voor een **INNER JOIN** toont alleen de overlappende gedeelten tussen merken en modellen.



## KЛАSSИКАЛЬНЫЕ ОФОРМЛЕНИЯ

### SAKILA DATABASE

1. **Toon alle films met de acteurs en de datum waarop elke film beschikbaar werd gesteld**

```
SELECT film.title, actor.first_name, actor.last_name, film.release_year
FROM film
INNER JOIN film_actor ON film.film_id = film_actor.film_id
INNER JOIN actor ON film_actor.actor_id = actor.actor_id;
```

- Hier wordt de release-informatie van de film gecombineerd met de acteurs die in de film spelen.

2. **Vind alle medewerkers met de huurtransacties die zij persoonlijk hebben afgehandeld**

```
SELECT staff.first_name, staff.last_name, rental.rental_date,
rental.return_date
FROM staff
INNER JOIN rental ON staff.staff_id = rental.staff_id;
```

- Deze query toont alle medewerkers met hun respectieve huurtransacties, wat helpt om te begrijpen wie welke klanten heeft geholpen.

## 2. LEFT JOIN

Een **LEFT JOIN** toont alle rijen uit de linkertabel (`merken`), ongeacht of er een overeenkomst is in de rechtertabel (`modellen`). Als er geen bijpassend model is, wordt er `NULL` weergegeven.

### Voorbeeld:

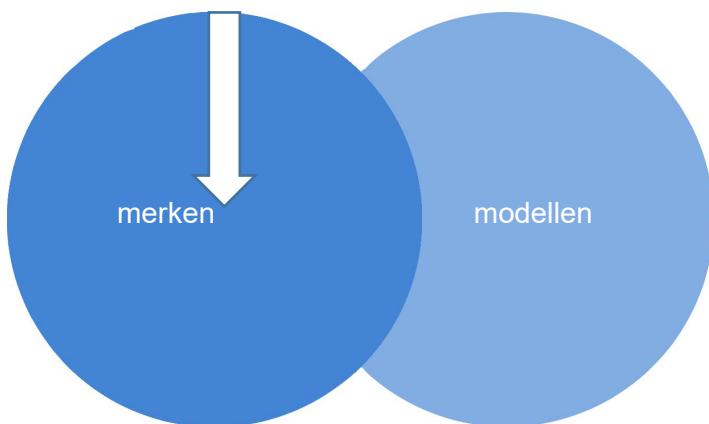
```
SELECT merken.naam, modellen.naam
FROM merken
LEFT JOIN modellen ON merken.id = modellen.merkid;
```

- Deze query selecteert alle merken, zelfs als ze geen modellen hebben. Bijvoorbeeld, als Volkswagen geen model heeft, wordt `NULL` weergegeven bij het model.

### Resultaat:

| naam       | naam |
|------------|------|
| audi       | A3   |
| audi       | A4   |
| bmw        | NULL |
| volkswagen | NULL |

### Grafisch:



Het Venn-diagram voor een **LEFT JOIN** laat de volledige cirkel van `merken` zien, plus de gedeelten die overlappen met `modellen`.

Voorbeeld: Stel je hebt een lijst van klanten en een lijst van bestellingen. Je wilt een overzicht van alle klanten, ook degenen die nog geen bestelling hebben geplaatst.

## KЛАSSИКАЛЬНЫЕ ОФОРМЛЕНИЯ

### SAKILA DATABASE

3. **Geef een lijst van alle klanten en hun laatst bekeken film, inclusief klanten die nog geen film hebben bekeken**

```
SELECT customer.first_name, customer.last_name, film.title
FROM customer
LEFT JOIN rental ON customer.customer_id = rental.customer_id
LEFT JOIN inventory ON rental.inventory_id = inventory.inventory_id
LEFT JOIN film ON inventory.film_id = film.film_id
ORDER BY rental.rental_date DESC;
```

- Deze query toont alle klanten en, indien beschikbaar, de laatst gehuurde film per klant.

4. **Toon alle categorieën van films en de hoeveelheid films die eraan zijn gekoppeld**

```
SELECT category.name, COUNT(film_category.film_id) AS aantal_films
FROM category
LEFT JOIN film_category ON category.category_id =
film_category.category_id
GROUP BY category.category_id;
```

- Deze query geeft een lijst van alle categorieën met het aantal films dat eraan gekoppeld is, inclusief categorieën zonder films.

### 3. RIGHT JOIN

Een **RIGHT JOIN** is vergelijkbaar met een LEFT JOIN, maar nu worden alle rijen uit de rechtertabel (modellen) weergegeven, ongeacht of er een overeenkomst is in de linkertabel (merken). Dit is handig als je zeker wilt weten dat je alle modellen toont, ook al is er geen bijbehorend merk.

#### Voorbeeld:

```
SELECT merken.naam, modellen.naam
FROM merken
RIGHT JOIN modellen ON merken.id = modellen.merkid;
```

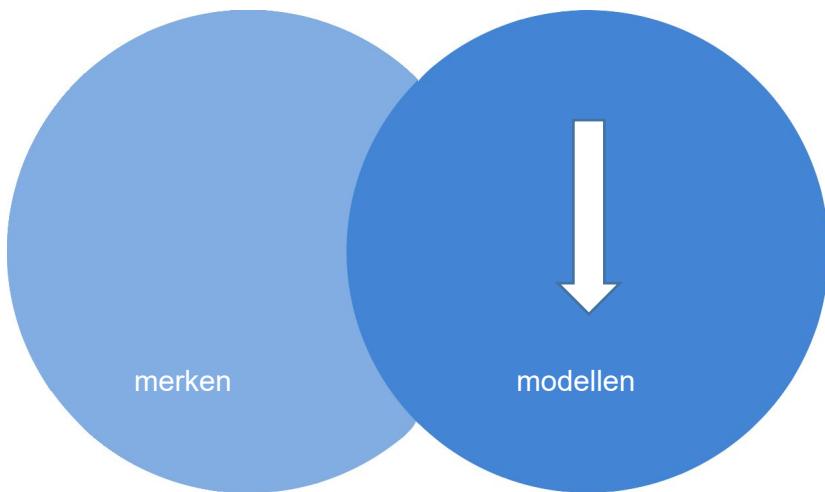
- In dit voorbeeld zouden alle modellen worden weergegeven, zelfs als er geen overeenkomstig merk is.

#### Resultaat:

Resultaat:

| naam | naam |
|------|------|
| audi | A3   |
| audi | A4   |
| NULL | A3   |
| NULL | A4   |

#### Grafisch:



Het Venn-diagram voor een **RIGHT JOIN** laat de volledige cirkel van modellen zien, plus de gedeelten die overlappen met merken.

Voorbeeld: Stel je hebt een lijst van bestellingen en een lijst van producten. Je wilt een overzicht van alle producten, ook als ze nooit zijn besteld.

## KЛАSSИКАЛЬНЫЕ ОФОРМЛЕНИЯ

### SAKILA DATABASE

5. **Toon een lijst van alle inventarisitems met de bijbehorende film, zelfs als er geen inventaris is gekoppeld aan een film**

```
SELECT inventory.inventory_id, film.title
FROM film
RIGHT JOIN inventory ON film.film_id = inventory.film_id;
```

- Dit toont alle inventarisitems en de gekoppelde film, zelfs als een film momenteel niet in de inventaris zit.

6. **Vind alle steden met winkels en de namen van die winkels, inclusief steden zonder winkels**

```
SELECT city.city, store.store_id
FROM city
RIGHT JOIN address ON city.city_id = address.city_id
RIGHT JOIN store ON address.address_id = store.address_id;
```

- Hiermee wordt een lijst gegenereerd van alle steden en, indien aanwezig, de winkels in die stad. Dit kan nuttig zijn voor inzicht in regio's zonder winkels.

## 4. CROSS JOIN

Een **CROSS JOIN** verbindt elke rij van de ene tabel met elke rij van de andere tabel. Het resultaat is een Cartesian product, wat betekent dat het aantal rijen het product is van het aantal rijen in beide tabellen. Dit type join is vaak inefficiënt voor grote datasets, maar kan nuttig zijn voor specifieke analyses.

### Voorbeeld:

```
SELECT *
FROM merken
CROSS JOIN modellen;
```

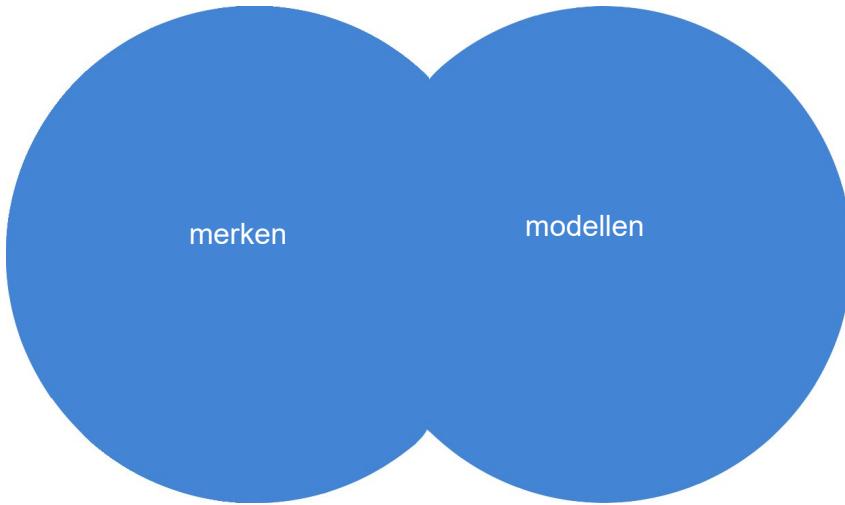
- Deze query combineert elke rij van de tabel `merken` met elke rij van de tabel `modellen`.

### Resultaat:

Resultaat:

| <b>id</b> | <b>naam</b> | <b>id</b> | <b>merkid</b> | <b>naam</b> |
|-----------|-------------|-----------|---------------|-------------|
| 1         | audi        | 1         | 1             | A3          |
| 2         | bmw         | 1         | 1             | A3          |
| 3         | volkswagen  | 1         | 1             | A3          |
| 1         | audi        | 2         | 1             | A4          |
| 2         | bmw         | 2         | 1             | A4          |
| 3         | volkswagen  | 2         | 1             | A4          |

### Grafisch:



Het Venn-diagram voor een **CROSS JOIN** toont de volledige overlapping tussen de twee tabellen, wat betekent dat elke rij in `merken` wordt gecombineerd met elke rij in `modellen`.

Voorbeeld: Stel je hebt een winkel met een lijst van klanten en een lijst van promotieproducten. Je wilt alle mogelijke combinaties van klanten en producten bekijken om te bepalen welke klanten je een promotie-aanbieding wilt sturen. In dit geval kun je een CROSS JOIN gebruiken om elk product met elke klant te combineren.

Klanten tabel:

| klant_id | naam  |
|----------|-------|
| 1        | Jan   |
| 2        | Maria |
| 3        | Peter |

producten tabel:

| product_id | naam      |
|------------|-----------|
| 1          | Koffiemok |
| 2          | T-shirt   |

Resultaat:

| klant | product   |
|-------|-----------|
| Jan   | Koffiemok |
| Jan   | T-shirt   |
| Maria | Koffiemok |
| Maria | T-shirt   |
| Peter | Koffiemok |
| Peter | T-shirt   |

## KЛАССИКАЛЬНЫЕ ОФОРМЛЕНИЯ

SAKILA DATABASE

### 7. Geef een combinatie van alle klanten en alle medewerkers

```
SELECT customer.first_name AS klant, staff.first_name AS medewerker
FROM customer
CROSS JOIN staff;
```

- Deze query toont alle mogelijke klant-medewerkercombinaties, wat nuttig kan zijn voor het evalueren van de mogelijke klant-ondersteuning relaties.

### 8. Lijst van alle films en alle acteurs om mogelijke cast-opties te onderzoeken

```
SELECT film.title, actor.first_name, actor.last_name
FROM film
CROSS JOIN actor;
```

- Deze query toont een volledige lijst van elke mogelijke combinatie van films en acteurs, wat handig kan zijn bij het simuleren van potentiële castingkeuzes.

## Samenvatting

- **INNER JOIN:** Toont alleen de rijen die overeenkomsten hebben in beide tabellen.
- **LEFT JOIN:** Toont alle rijen uit de linkertabel, zelfs als er geen overeenkomst is in de rechtertabel.
- **RIGHT JOIN:** Toont alle rijen uit de rechtertabel, zelfs als er geen overeenkomst is in de linkertabel.
- **CROSS JOIN:** Verbindt elke rij van de ene tabel met elke rij van de andere tabel.

Door deze join types goed te begrijpen, kun je effectief data combineren en relaties tussen tabellen inzichtelijk maken. Dit is een essentieel onderdeel van databasebeheer en helpt bij het opstellen van rijke datasets voor analyses en rapportages.

## 5. COMPLEXE OEFENINGEN

### KЛАССИКАЛЬНЫЕ ОУЧЕНИЯ

#### SAKILA DATABASE

9. Geef een lijst van films die zijn gehuurd door klanten, met de naam van de medewerker die de verhuur heeft uitgevoerd en de naam van de stad waarin de klant woont

```
SELECT film.title, customer.first_name AS klant, staff.first_name AS medewerker, city.city AS woonplaats
FROM film
INNER JOIN inventory ON film.film_id = inventory.film_id
INNER JOIN rental ON inventory.inventory_id = rental.inventory_id
INNER JOIN customer ON rental.customer_id = customer.customer_id
INNER JOIN address ON customer.address_id = address.address_id
INNER JOIN city ON address.city_id = city.city_id
INNER JOIN staff ON rental.staff_id = staff.staff_id;
```

- Deze complexe join combineert meerdere tabellen om te tonen welke films door klanten zijn gehuurd, met de medewerkers die de transactie hebben afgehandeld en de stad waarin de klant woont.

10. Toon het totaalbedrag dat klanten hebben betaald, gegroepeerd per stad

```
SELECT city.city, SUM(payment.amount) AS totale_omzet
FROM payment
INNER JOIN rental ON payment.rental_id = rental.rental_id
INNER JOIN customer ON rental.customer_id = customer.customer_id
INNER JOIN address ON customer.address_id = address.address_id
INNER JOIN city ON address.city_id = city.city_id
GROUP BY city.city;
```

- Hier wordt het totale bedrag dat klanten hebben betaald per stad berekend, wat handig kan zijn voor marketinganalyse.

## 6. GEAVANCEERDE OEFENINGEN

### KЛАSSИКАЛЬНЫЕ ОУЧЕНИЯ

#### SAKILA DATABASE

11. **Lijst van klanten die meer dan vijf films in dezelfde categorie hebben gehuurd**

```
WITH KlantenHuur AS (
 SELECT customer.customer_id, category.name AS categorie,
 COUNT(rental.rental_id) AS aantal_huur
 FROM rental
 INNER JOIN inventory ON rental.inventory_id = inventory.inventory_id
 INNER JOIN film ON inventory.film_id = film.film_id
 INNER JOIN film_category ON film.film_id = film_category.film_id
 INNER JOIN category ON film_category.category_id = category.category_id
 INNER JOIN customer ON rental.customer_id = customer.customer_id
 GROUP BY customer.customer_id, category.name
)
SELECT customer.first_name, customer.last_name, categorie
FROM KlantenHuur
INNER JOIN customer ON KlantenHuur.customer_id = customer.customer_id
WHERE aantal_huur > 5;
```

- In deze query wordt een CTE (common table expression) gebruikt om het aantal gehuurde films per klant per categorie te berekenen, en daarna klanten te tonen die meer dan vijf films in dezelfde categorie hebben gehuurd.

12. **Vind alle films die zijn uitgeleend door de minst actieve medewerkers (die de minste verhuurtransacties hebben gedaan)**

```
WITH MedewerkerVerhuur AS (
 SELECT staff.staff_id, COUNT(rental.rental_id) AS totaal_verhuur
 FROM rental
 INNER JOIN staff ON rental.staff_id = staff.staff_id
 GROUP BY staff.staff_id
)
SELECT film.title, staff.first_name, staff.last_name
FROM film
INNER JOIN inventory ON film.film_id = inventory.film_id
INNER JOIN rental ON inventory.inventory_id = rental.inventory_id
INNER JOIN staff ON rental.staff_id = staff.staff_id
WHERE staff.staff_id IN (
 SELECT staff_id FROM MedewerkerVerhuur WHERE totaal_verhuur =
 (SELECT MIN(totaal_verhuur) FROM MedewerkerVerhuur
)
);

```

- Deze query maakt gebruik van een **CTE** om de medewerkers te identificeren met het laagste aantal verhuurtransacties, en toont vervolgens alle films die door deze medewerkers zijn uitgeleend.

## 1. INNER JOIN Oefeningen

1. Selecteer alle films met hun acteurs.

```
SELECT film.title, actor.first_name, actor.last_name
FROM film
INNER JOIN film_actor ON film.film_id = film_actor.film_id
INNER JOIN actor ON film_actor.actor_id = actor.actor_id;
```

2. Vind alle klanten en de stad waar ze wonen.

```
SELECT customer.first_name, customer.last_name, city.city
FROM customer
INNER JOIN address ON customer.address_id = address.address_id
INNER JOIN city ON address.city_id = city.city_id;
```

3. Toon alle films en de categorie waartoe ze behoren.

```
SELECT film.title, category.name
FROM film
INNER JOIN film_category ON film.film_id = film_category.film_id
INNER JOIN category ON film_category.category_id =
category.category_id;
```

4. Geef alle huurtransacties weer met de klantnamen en de film die ze huurden.

```
SELECT rental.rental_id, customer.first_name, customer.last_name,
film.title
FROM rental
INNER JOIN customer ON rental.customer_id = customer.customer_id
INNER JOIN inventory ON rental.inventory_id = inventory.inventory_id
INNER JOIN film ON inventory.film_id = film.film_id;
```

5. Selecteer alle medewerkers en hun bijbehorende winkel.

```
SELECT staff.first_name, staff.last_name, store.store_id
FROM staff
INNER JOIN store ON staff.store_id = store.store_id;
```

## 2. LEFT JOIN Oefeningen

6. Geef een lijst van alle klanten en, indien aanwezig, hun huurtransacties.

```
SELECT customer.first_name, customer.last_name, rental.rental_id
FROM customer
LEFT JOIN rental ON customer.customer_id = rental.customer_id;
```

7. Toon alle films en de eventuele verhuurdata, inclusief films die niet zijn verhuisd.

```
SELECT film.title, rental.rental_date
FROM film
LEFT JOIN inventory ON film.film_id = inventory.film_id
LEFT JOIN rental ON inventory.inventory_id = rental.inventory_id;
```

8. Selecteer alle adressen en geef indien beschikbaar de bijbehorende klanten weer.

```
SELECT address.address, customer.first_name, customer.last_name
FROM address
LEFT JOIN customer ON address.address_id = customer.address_id;
```

9. Toon alle medewerkers en, indien aanwezig, de huurtransacties die ze hebben verwerkt.

```
SELECT staff.first_name, staff.last_name, rental.rental_id
FROM staff
LEFT JOIN rental ON staff.staff_id = rental.staff_id;
```

10. Vind alle categorieën en toon alle films die eronder vallen, inclusief categorieën zonder films. sql      **SELECT** category.name, film.title      **FROM** category
 **LEFT JOIN** film\_category **ON** category.category\_id =
 film\_category.category\_id      **LEFT JOIN** film **ON** film\_category.film\_id =
 film.film\_id;

## RIGHT JOIN Oefeningen

11. Toon alle films en geef, indien beschikbaar, de acteurs in die films weer.

```
SELECT film.title, actor.first_name, actor.last_name FROM actor
RIGHT JOIN film_actor ON actor.actor_id = film_actor.actor_id RIGHT
JOIN film ON film_actor.film_id = film.film_id;
```

12. Geef alle inventarisitems weer en de bijbehorende films, zelfs als er geen bijpassende film is.

```
SELECT inventory.inventory_id, film.title FROM film RIGHT JOIN
inventory ON film.film_id = inventory.film_id;
```

13. Toon alle adressen en de klanten die op dat adres wonen, inclusief adressen zonder klanten.

```
SELECT address.address, customer.first_name, customer.last_name
FROM customer RIGHT JOIN address ON customer.address_id =
address.address_id;
```

14. Geef een lijst van alle winkels en de medewerkers die daar werken, zelfs als er geen medewerkers zijn.

```
SELECT store.store_id, staff.first_name, staff.last_name FROM staff
RIGHT JOIN store ON staff.store_id = store.store_id;
```

15. Toon alle categorieën en de films die erbij horen, zelfs als er geen film aan een categorie is gekoppeld.

```
SELECT category.name, film.title FROM film RIGHT JOIN
film_category ON film.film_id = film_category.film_id RIGHT JOIN
category ON film_category.category_id = category.category_id;
```

## 4. CROSS JOIN Oefeningen

16. Toon een combinatie van alle klanten met alle films.

```
SELECT customer.first_name, customer.last_name, film.title FROM
customer CROSS JOIN film;
```

17. Maak een combinatie van alle medewerkers en alle winkels.

```
SELECT staff.first_name, staff.last_name, store.store_id FROM staff
CROSS JOIN store;
```

18. Selecteer alle categorieën met alle mogelijke acteurs.

```
SELECT category.name, actor.first_name, actor.last_name FROM
category CROSS JOIN actor;
```

19. Geef een combinatie van alle films en alle mogelijke huurdatums.

```
SELECT film.title, rental.rental_date FROM film CROSS JOIN
rental;
```

20. Maak een combinatie van alle mogelijke inventarisitems en alle klanten.

```
SELECT inventory.inventory_id, customer.first_name, customer.last_name
FROM inventory CROSS JOIN customer;
```

## 5. Complexe Join Oefeningen

21. Toon alle films met hun categorieën en acteurs.

```
SELECT film.title, category.name, actor.first_name, actor.last_name
FROM film INNER JOIN film_category ON film.film_id =
film_category.film_id INNER JOIN category ON
film_category.category_id = category.category_id INNER JOIN
film_actor ON film.film_id = film_actor.film_id INNER JOIN actor ON
film_actor.actor_id = actor.actor_id;
```

22. Vind alle klanten die een film hebben gehuurd, inclusief de medewerker die de transactie heeft afgehandeld.

```
SELECT customer.first_name, customer.last_name, film.title,
staff.first_name AS medewerker FROM rental INNER JOIN customer
ON rental.customer_id = customer.customer_id INNER JOIN inventory
ON rental.inventory_id = inventory.inventory_id INNER JOIN film ON
inventory.film_id = film.film_id INNER JOIN staff ON
rental.staff_id = staff.staff_id;
```

**23. Toon de totale hoeveelheid huurtransacties per film.**

```
SELECT film.title, COUNT(rental.rental_id) AS aantal_huur FROM film
INNER JOIN inventory ON film.film_id = inventory.film_id INNER JOIN
rental ON inventory.inventory_id = rental.inventory_id GROUP BY
film.title;
```

**24. Selecteer alle winkels en de klanten die daar hebben gehuurd.**

```
SELECT store.store_id, customer.first_name, customer.last_name FROM
store INNER JOIN staff ON store.store_id = staff.store_id INNER
JOIN rental ON staff.staff_id = rental.staff_id INNER JOIN customer
ON rental.customer_id = customer.customer_id;
```

**25. Geef een lijst van alle films met hun gemiddelde huurprijs per categorie.**

```
SELECT category.name, film.title, AVG(film.rental_rate) AS
gemiddelde_prijs FROM film INNER JOIN film_category ON
film.film_id = film_category.film_id INNER JOIN category ON
film_category.category_id = category.category_id GROUP BY
category.name, film.title;
```

## TABEL ALIASES

Een **tabel alias** is een alternatieve naam die je aan een tabel toewijst in een SQL-query. Het gebruik van een tabel alias maakt je query vaak **korter, leesbaarder**, en **makkelijker te schrijven**, vooral wanneer je met meerdere tabellen werkt, joins gebruikt of dezelfde tabel meerdere keren in een query nodig hebt. Een alias is slechts een tijdelijke naam die alleen geldt binnen de context van die specifieke query.

### Hoe werkt een Tabel Alias?

Je kunt een alias toekennen met behulp van het sleutelwoord AS, maar dit sleutelwoord is optioneel. Het alias komt direct na de tabelnaam.

**Basisvoorbeeld:**

```
SELECT *
FROM klanten AS k;
```

In dit voorbeeld wordt de tabel klanten hernoemd naar k voor gebruik in de query.

### Voordelen van Tabel Aliassen

1. **Leesbaarheid:** Vooral als je lange tabellamen hebt, maakt een alias de query korter en eenvoudiger te begrijpen.
2. **Duplicatie van Tabellen:** Wanneer je **self joins** gebruikt (joins van een tabel met zichzelf), zijn aliassen essentieel om verschillende versies van de tabel te onderscheiden.
3. **Gebruiksgemak:** Het vereenvoudigt het schrijven van complexe queries waar dezelfde tabelnaam vaak wordt gebruikt.

### Voorbeeldgebruik van Tabel Aliassen

Stel dat je werkt met de **Sakila**-database, die onder andere de tabellen rental, inventory, en film bevat. Zonder tabelaliassen kunnen je joins er log en moeilijk leesbaar uitzien.

**Zonder Tabelaliassen:**

```
SELECT film.title, rental.rental_date
FROM rental
INNER JOIN inventory ON rental.inventory_id = inventory.inventory_id
INNER JOIN film ON inventory.film_id = film.film_id;
```

In deze query worden de volledige tabelnamen telkens herhaald, wat niet alleen langdradig is maar ook lastig te volgen als je meerdere joins hebt.

**Met Tabelaliassen:**

```
SELECT f.title, r.rental_date
FROM rental AS r
INNER JOIN inventory AS i ON r.inventory_id = i.inventory_id
INNER JOIN film AS f ON i.film_id = f.film_id;
```

- **r**: Alias voor rental
- **i**: Alias voor inventory
- **f**: Alias voor film

Door aliassen (**r**, **i**, en **f**) te gebruiken, kun je de query veel eenvoudiger schrijven en wordt deze overzichtelijker. Dit helpt ook als de tabelnamen lang zijn, zoals `film_actor`, die je kunt verkorten tot `fa`.

## GEAVANCEERDE OEFENINGEN.

### **1. Films met de hoogste gemiddelde beoordeling per categorie**

Uitleg: Toon elke film met zijn gemiddelde beoordeling, gegroepeerd per categorie.

```
SELECT category.name AS categorie, film.title AS film, AVG(rating.rating) AS
gemiddelde_beoordeling FROM film INNER JOIN film_category ON
film.film_id = film_category.film_id INNER JOIN category ON
film_category.category_id = category.category_id INNER JOIN rating ON
film.film_id = rating.film_id GROUP BY category.name, film.title ORDER
BY categorie, gemiddelde_beoordeling DESC;
```

### **2. Vind alle klanten die meer dan drie films hebben gehuurd in dezelfde categorie**

Uitleg: Toon de klanten die minimaal drie films uit dezelfde categorie hebben gehuurd.

```
SELECT customer.first_name, customer.last_name, category.name,
COUNT(rental.rental_id) AS aantal_huur FROM customer INNER JOIN rental
ON customer.customer_id = rental.customer_id INNER JOIN inventory ON
rental.inventory_id = inventory.inventory_id INNER JOIN film ON
inventory.film_id = film.film_id INNER JOIN film_category ON film.film_id =
film_category.film_id INNER JOIN category ON film_category.category_id =
category.category_id GROUP BY customer.customer_id, category.category_id
HAVING aantal_huur > 3;
```

### **3. Filmcategorieën met omzet per winkel**

Uitleg: Toon voor elke winkel de totale omzet per categorie.

```
SELECT store.store_id, category.name AS categorie, SUM(payment.amount) AS
totale_omzet FROM store INNER JOIN staff ON store.store_id =
staff.store_id INNER JOIN payment ON staff.staff_id = payment.staff_id
INNER JOIN rental ON payment.rental_id = rental.rental_id INNER JOIN
inventory ON rental.inventory_id = inventory.inventory_id INNER JOIN film
ON inventory.film_id = film.film_id INNER JOIN film_category ON
film.film_id = film_category.film_id INNER JOIN category ON
film_category.category_id = category.category_id GROUP BY store.store_id,
category.name;
```

## **4. Klanten zonder huur in de afgelopen zes maanden**

Uitleg:Vind alle klanten die geen film hebben gehuurd in de laatste zes maanden.

```
SELECT customer.first_name, customer.last_name FROM customer LEFT JOIN
rental ON customer.customer_id = rental.customer_id WHERE
rental.rental_date IS NULL OR rental.rental_date < DATE_SUB(CURDATE(),
INTERVAL 6 MONTH);
```

## **5. Vind medewerkers met de meeste transacties per winkel**

Uitleg:Toon de medewerkers met de meeste uitgevoerde huurtransacties per winkel.

```
SELECT store.store_id, staff.first_name, staff.last_name,
COUNT(rental.rental_id) AS aantal_transacties FROM store INNER JOIN
staff ON store.store_id = staff.store_id INNER JOIN rental ON
staff.staff_id = rental.staff_id GROUP BY store.store_id, staff.staff_id
ORDER BY store.store_id, aantal_transacties DESC;
```

## **6. Klanten die films van dezelfde categorie hebben gehuurd als hun vorige huur**

Uitleg:Toon klanten die twee keer achter elkaar films uit dezelfde categorie hebben gehuurd.

```
SELECT c.first_name, c.last_name, cat.name AS categorie FROM rental r1
INNER JOIN rental r2 ON r1.customer_id = r2.customer_id AND r2.rental_id >
r1.rental_id INNER JOIN inventory i1 ON r1.inventory_id = i1.inventory_id
INNER JOIN film_category fc1 ON i1.film_id = fc1.film_id INNER JOIN
inventory i2 ON r2.inventory_id = i2.inventory_id INNER JOIN film_category
fc2 ON i2.film_id = fc2.film_id INNER JOIN category cat ON fc1.category_id
= cat.category_id INNER JOIN customer c ON r1.customer_id = c.customer_id
WHERE fc1.category_id = fc2.category_id ORDER BY c.customer_id;
```

## **7. Films met geen enkele huur in een specifieke winkel**

Uitleg:Toon de films die nog nooit zijn verhuurd in winkel 1.

```
SELECT film.title FROM film LEFT JOIN inventory ON film.film_id =
inventory.film_id AND inventory.store_id = 1 LEFT JOIN rental ON
inventory.inventory_id = rental.inventory_id WHERE rental.rental_id IS
NULL;
```

## **8. Grootste klanten op basis van huuruitgaven per maand**

Uitleg: Toon de klanten met de hoogste huuruitgaven voor elke maand.

```
SELECT YEAR(payment.payment_date) AS jaar, MONTH(payment.payment_date) AS maand, customer.first_name, customer.last_name, SUM(payment.amount) AS totale_uitzgaben FROM payment INNER JOIN customer ON payment.customer_id = customer.customer_id GROUP BY jaar, maand, customer.customer_id ORDER BY jaar, maand, totale_uitzgaben DESC;
```

## **9. Lijst van alle medewerkers en hun winkel met het aantal klanten dat ze hebben geholpen**

Uitleg: Toon alle medewerkers en het aantal klanten dat ze hebben geholpen.

```
SELECT staff.first_name, staff.last_name, store.store_id, COUNT(DISTINCT rental.customer_id) AS aantal_klanten FROM staff LEFT JOIN store ON staff.store_id = store.store_id LEFT JOIN rental ON staff.staff_id = rental.staff_id GROUP BY staff.staff_id, store.store_id;
```

## **10. Lijst van alle films die alleen in één winkel beschikbaar zijn**

Uitleg: Vind films die alleen in één specifieke winkel aanwezig zijn.

```
SELECT film.title, COUNT(DISTINCT inventory.store_id) AS aantal_winkels FROM film INNER JOIN inventory ON film.film_id = inventory.film_id GROUP BY film.film_id HAVING aantal_winkels = 1;
```

## **11. Klanten die meerdere films op dezelfde dag hebben gehuurd**

Uitleg: Toon klanten die op dezelfde dag meer dan één film hebben gehuurd.

```
SELECT customer.first_name, customer.last_name, rental.rental_date, COUNT(rental.rental_id) AS aantal_films FROM customer INNER JOIN rental ON customer.customer_id = rental.customer_id GROUP BY customer.customer_id, rental.rental_date HAVING aantal_films > 1;
```

## 12. Films met zowel de hoogste als de laagste beoordelingen per categorie

Uitleg: Toon de films met de hoogste en de laagste beoordelingen voor elke categorie.

```
WITH Beoordelingen AS (SELECT category.name AS categorie, film.title,
AVG(rating.rating) AS gemiddelde FROM film INNER JOIN
film_category ON film.film_id = film_category.film_id INNER JOIN
category ON film_category.category_id = category.category_id INNER
JOIN rating ON film.film_id = rating.film_id GROUP BY category.name,
film.title)

SELECT * FROM Beoordelingen WHERE gemiddelde = (SELECT MAX(gemiddelde)
FROM Beoordelingen WHERE categorie = Beoordelingen.categorie) OR
gemiddelde = (SELECT MIN(gemiddelde) FROM Beoordelingen WHERE categorie =
Beoordelingen.categorie);
```

De **WITH-clausule**, ook wel een **Common Table Expression (CTE)** genoemd, wordt gebruikt om een tijdelijke naam te geven aan een subquery waarvan je later in je SQL-query gebruik kunt maken. Het belangrijkste doel van een **CTE** is om **leesbaarheid te verbeteren en complexe query's eenvoudiger** te maken, vooral wanneer dezelfde subquery meerdere keren wordt gebruikt. Hierdoor wordt de SQL-code modulairder en makkelijker te onderhouden.

### Hoe werkt de **WITH-clausule**?

De **WITH-clausule** laat je een subquery definiëren die je vervolgens kunt gebruiken in de hoofdsom van je query. Deze subquery wordt als een soort tijdelijke tabel gedefinieerd die alleen beschikbaar is binnen de context van de query waarin hij wordt aangeroepen.

## 13. Films die alleen zijn verhuurd door specifieke medewerkers

Uitleg: Toon alle films die uitsluitend door medewerker 'Mike' zijn verhuurd.

```
SELECT DISTINCT film.title FROM film INNER JOIN inventory ON
film.film_id = inventory.film_id INNER JOIN rental ON
inventory.inventory_id = rental.inventory_id INNER JOIN staff ON
rental.staff_id = staff.staff_id WHERE staff.first_name = 'Mike' EXCEPT
SELECT DISTINCT film.title FROM film INNER JOIN inventory ON
film.film_id = inventory.film_id INNER JOIN rental ON
inventory.inventory_id = rental.inventory_id INNER JOIN staff ON
rental.staff_id = staff.staff_id WHERE staff.first_name != 'Mike';
```

## INSERT STATEMENT

We gaan verder met het CREATE gedeelte van de CRUD. Deze wordt in SQL uitgevoerd door het INSERT STATEMENT. Met het insert statement voegen we één of meerdere records toe aan een database.

Schrijfwijze :

```
INSERT INTO `customer`(
`customer_id`,
`store_id`,
`first_name`,
`last_name`,
`email`,
`address_id`,
`active`,
`create_date`,
`last_update`)
VALUES ([value-1],[value-2],[value-3],[value-4],[value-5],[value-6],[value-7],[value-8],[value-9])
```

## UPDATE STATEMENT

We gaan verder met het UPDATE gedeelte van de CRUD. Deze wordt in SQL uitgevoerd door het UPDATE STATEMENT. Het update gedeelte zal een bestaand record wijzigen in de database.

```
UPDATE `customer` SET `customer_id`=[value-1],`store_id`=[value-2],`first_name`=[value-3],`last_name`=[value-4],`email`=[value-5],`address_id`=[value-6],`active`=[value-7],`create_date`=[value-8],`last_update`=[value-9] WHERE 1
```

## DELETE STATEMENT

We gaan verder met het DELETE gedeelte van de CRUD. Deze wordt in SQL uitgevoerd door het DELETE STATEMENT. Met het delete statement worden records compleet verwijderd uit de database. In realiteit gaan we dit in een relationele database nooit toepassen. Data wordt betere inactief gezet i.p.v. verwijderd.

```
DELETE FROM `customer` WHERE 0
```

## VIEWS

Een view wordt gemaakt wanneer je een resultaat van bijvoorbeeld een query wenst te behouden.

CREATE VIEW viewnaam

AS

Voorbeeld:

```
DROP VIEW Categoryaantal;
CREATE VIEW Categoryaantal
AS
SELECT category.name, count(film.film_id) as aantal
FROM category
LEFT JOIN film_category ON category.category_id =
film_category.category_id
LEFT JOIN film ON film.film_id = film_category.film_id
GROUP BY category.name
```

GEBRUIK

Select \* from Categoryaantal

Resultaat:

| + Options   |      |        |
|-------------|------|--------|
|             | name | aantal |
| Action      | 64   |        |
| Animation   | 66   |        |
| Children    | 60   |        |
| Classics    | 57   |        |
| Comedy      | 58   |        |
| Documentary | 68   |        |
| Drama       | 62   |        |
| Family      | 69   |        |
| Foreign     | 73   |        |
| Games       | 61   |        |
| Horror      | 56   |        |
| Music       | 51   |        |
| New         | 63   |        |
| Sci-Fi      | 61   |        |
| Sports      | 74   |        |
| Travel      | 57   |        |

## EIGEN FUNCTIES

Je kan natuurlijk ook je eigen functies zelf schrijven in MySQL.  
We spreken soms van deterministic of non-deterministic functions.

Verschil tussen DETERMINISTIC en NON -DETERMINISTIC ?

- Een functie wordt als *deterministisch* beschouwd als deze altijd dezelfde resultatenset retourneert wanneer deze met dezelfde set invoerwaarden wordt aangeroepen.
- Een functie wordt als *niet-deterministisch* beschouwd als deze niet dezelfde resultatenset retourneert wanneer deze met dezelfde set invoerwaarden wordt aangeroepen.

Dit klinkt misschien wat ingewikkeld, maar dat is het echt niet. Neem bijvoorbeeld de functies DATEDIFF en GETDATE . DATEDIFF is deterministic omdat het altijd dezelfde gegevens retourneert telkens wanneer het met dezelfde invoerparameters wordt uitgevoerd. GETDATE is niet-deterministisch omdat het nooit dezelfde datum zal retourneren elke keer dat het wordt uitgevoerd.

Op het niveau van de sakila database open je de SQL tab en voer je onderstaand voorbeeld uit. Codegewijs maken we op dergelijke manier een functie aan.

Het DELIMITER-commando wordt gebruikt om het standaardscheidingsteken van MySQL-commando's te wijzigen (;). Omdat de instructies binnen de routines (functies, opgeslagen procedures of triggers) eindigen met een puntkomma (;), gebruiken we DELIMITER om ze als een samengestelde instructie te behandelen.

De functie **retourneert één getal** van het type **decimal**. Er worden 2 parameters ontvangen via het programma.

Het programma begint met de key words **BEGIN** en eindigt met **END**.

Voorbeeld:

```
DELIMITER $$
CREATE FUNCTION voorbeeld_function (x1 decimal, y1 decimal)
RETURNS decimal
DETERMINISTIC
BEGIN
 DECLARE dist decimal;
 SET dist = SQRT(x1 - y1);
 RETURN dist;
END$$
DELIMITER ;
```

Wanneer je vervolgens op functions klikt in het menu aan de linkerkant van de sakila database dan krijg je het volgende scherm te zien.

| Name                       | Action                   | Type     | Returns       |
|----------------------------|--------------------------|----------|---------------|
| get_customer_balance       | Edit Execute Export Drop | FUNCTION | decimal(5,2)  |
| inventory_held_by_customer | Edit Execute Export Drop | FUNCTION | int(11)       |
| inventory_in_stock         | Edit Execute Export Drop | FUNCTION | tinyint(1)    |
| voorbeeld_function         | Edit Execute Export Drop | FUNCTION | decimal(10,0) |

Check all     With selected:

## EXECUTE

Wanneer je op execute klikt wordt de functie opgeroepen en dien je de 2 gevraagde parameters mee te geven.

| Name | Type    | Function | Value |
|------|---------|----------|-------|
| x1   | DECIMAL |          | 8     |
| y1   | DECIMAL |          | 5     |

Resultaat in dit voorbeeld:

Your SQL query has been executed successfully.

```
SET @p0='8'; SET @p1='5'; SELECT `voorbeeld_function`(@p0, @p1) AS `voorbeeld_function`;
```

Execution results of routine `voorbeeld\_function`

| voorbeeld_function |
|--------------------|
| 2                  |

## EXECUTE - CODE

Je kan ook vanaf de SQL TAB je functie aanroepen zoals we die gewoon zijn te gebruiken. Net zoals bij de built-in functions van MySQL is het dezelfde manier van uitvoeren. We gebruiken het select statement en vragen de functie op en geven zijn parameters mee.

Showing rows 0 - 0 (1 total, Query took 0.0003 seconds.)

```
select voorbeeld_function(8,5)
```

|                         | Show all | Number of rows: | 25 | Filter rows |
|-------------------------|----------|-----------------|----|-------------|
| + Options               |          |                 |    |             |
| voorbeeld_function(8,5) |          |                 |    |             |
| 2                       |          |                 |    |             |

SET

Met set ga je een variabele initialiseren die je dan meegeeft als parameter aan je functie. Na iedere set dien je een ; te gebruiken. Deze variabelen worden steeds voorafgegeven door een @.

Server: MySQL:3306 » Database: sakila

Structure SQL Search

Run SQL query/queries on database sakila:

```
1 SET @p1 = 8;
2 SET @p2 = 5;
3 SELECT voorbeeld_function(@p1,@p2)|
```

**Resultaat:**

Iedere lijn wordt door de query engine van PHPmyadmin apart uitgevoerd. De laatste query bevat uiteindelijk het resultaat.

The screenshot shows the PHPMyAdmin interface with the following details:

- Server:** MySQL:3306
- Database:** sakila
- Navigation Bar:** Structure, SQL, Search, Query, Export, Import
- Show query box:** A dropdown menu is open.
- Query Results:**
  - MySQL returned an empty result set (i.e. zero rows). (Query took 0.0002 seconds.)
  - SET @p1 = 8
  - MySQL returned an empty result set (i.e. zero rows). (Query took 0.0001 seconds.)
  - SET @p2 = 5
  - Showing rows 0 - 0 (1 total, Query took 0.0003 seconds.)
  - SELECT voorbeeld\_function(@p1,@p2)

## ***STORED PROCEDURE***

Waarom wordt een stored procedure gebruikt.

De belangrijkste factor is snelheid. Stored procedures maken gebruik van caching. Elk resultaat van een stored procedure wordt bewaard zodat een query niet telkens opnieuw dient te worden uitgevoerd. Daarnaast wordt de stored procedure op de server bewaard waardoor één enkele request (vraag) voldoende is.

## ***CREATE PROCEDURE***

DELIMITER \$\$

```
CREATE PROCEDURE myCustomers()
BEGIN
 SELECT
 last_name,
 first_name,
 email,
 active,
 FROM
 customer
 ORDER BY last_name,first_name ;
END$$
DELIMITER ;
```

### Schrijfwijze:

Een stored procedure kan uit meerdere blokken code bestaan die gescheiden worden door een puntkomma (;).

### Het gebruik van DELIMITER:

De volgende stap is om het standaardscheidingssteken van de MySQL-scriptparser te wijzigen van puntkomma (;) in dubbele dollartekken (\$\$). De reden dat u dit doet, is dat de puntkomma's na elke instructie in de hoofdtekst van de routine niet door de parser worden geïnterpreteerd als het einde van de CREATE PROCEDURE instructie. Dit komt omdat het hele CREATE PROCEDURE blok, van CREATE PROCEDURE tot END eigenlijk een enkele instructie is die op zichzelf moet worden uitgevoerd. Zonder de wijziging van het scheidingssteken zou het script breken, omdat elke instructie in BEGIN en END afzonderlijk zou worden uitgevoerd. Merk op dat u een aantal niet-gereserveerde tekens kunt gebruiken om uw eigen aangepaste scheidingssteken te maken.

Als resultaat ziet u in de database de toegevoegde stored procedure myCustomers.

### Uitvoeren stored procedure

Wanneer u een stored procedure selecteert dan hebt u de mogelijkheid om deze uit te voeren.

| Routines          |          |           |            |             |             |  |  |
|-------------------|----------|-----------|------------|-------------|-------------|--|--|
| Naam              | Actie    |           |            | Type        | Geeft terug |  |  |
| film_in_stock     | Wijzigen | Uitvoeren | Exporteren | Verwijderen | PROCEDURE   |  |  |
| film_not_in_stock | Wijzigen | Uitvoeren | Exporteren | Verwijderen | PROCEDURE   |  |  |
| myCustomers       | Wijzigen | Uitvoeren | Exporteren | Verwijderen | PROCEDURE   |  |  |
| rewards_report    | Wijzigen | Uitvoeren | Exporteren | Verwijderen | PROCEDURE   |  |  |

Een tweede mogelijkheid heeft u om in sql zelf de call functie te gebruiken en de stored procedure uit te roepen: **CALL myCustomers()**

### Variabelen in stored procedures

#### Declareren variabelen

Declareren van een variabele gebeurt met het keyword **DECLARE**

```
DECLARE x INT DEFAULT 0;
```

#### Toewijzen van variabelen

Het vullen of toewijzen van een variabele gebeurt met het keyword **SET**

```
DECLARE x INT DEFAULT 0;
SET x = 10;
```

### Parameters in stored procedures

#### IN, OUT en INOUT parameters.

In is de standaard. Wanneer we deze definiëren zal het aangeroepen programma een parameter doorgeven aan de stored procedure.

Wanneer we de bestaande stored procedure bekijken **film\_in\_stock**.

```
DROP PROCEDURE `film_in_stock`;
CREATE PROCEDURE `film_in_stock`(IN `p_film_id` INT, IN `p_store_id` INT, OUT `p_film_count` INT)
BEGIN
 SELECT inventory_id
 FROM inventory
 WHERE film_id = p_film_id
 AND store_id = p_store_id
 AND inventory_in_stock(inventory_id);

 SELECT COUNT(*)
 FROM inventory
 WHERE film_id = p_film_id
 AND store_id = p_store_id
 AND inventory_in_stock(inventory_id)
 INTO p_film_count;
END
```

# Database normalisatie

**D**atabase-normalisatie of gewoon normalisatie, is een proces dat wordt gebruikt voor datamodellering of het maken van databases, waarbij u uw gegevens en tabellen organiseert zodat deze efficiënt kunnen worden bijgewerkt.

Het kan op elke relationele database worden uitgevoerd. Dit betekent dat normalisatie in een DMBS (DATABASE MANAGEMENT SYSTEM) kan worden toegepast. Bijvoorbeeld: Oracle, MySQL, Microsoft SQL Server, PostgreSQL, ...

## Voordelen van database normalisatie

- voorkomt dat gegevens dubbel worden opgeslagen
- voorkomt dat gegevens niet worden verwijderd of data gegevens verloren raken
- vermindering van de opslagruimte
- zorgt voor een vlotte afhandeling van zoekopdrachten

Kortom, het maakt de volledige database efficiënter.

## Doel database normalisatie

Alle gegevensafwijkingen detecteren. Deze gegevensafwijkingen worden ook gegevensanomalieën genoemd.

## Voorbeeld

In ons voorbeeld gaan we van start met data die heel wat gegevensafwijkingen bevat. Onderstaande is dus geen genormaliseerde tabel of database.

| studentnaam     | prijs | cursusnaam        | cursus 1          | cursus 2    | cursus 3    |
|-----------------|-------|-------------------|-------------------|-------------|-------------|
| Tom Vanhoutte   | 100   | Full Stack        | PHP OOP           | Javascript  | Laravel     |
| Tim Vanhoutte   | 200   | Grafisch design   | Photoshop 1       | Photoshop 2 | Photoshop 3 |
| Nick Dewaele    | 250   | Wetenschappen     | Fysica 1          | Fysica 2    | Chemie      |
| Remco Evenepoel | 150   | Fietsenhersteller | Afstellen fietsen |             |             |

Zoals je kan zien worden hier bepaalde onderwerpen bijgehouden:

- studentnaam
- de prijs die de student heeft betaald
- de cursus en lessen die de student zal volgen of gevuld heeft

Er zijn ook zaken die niet aanwezig zijn zoals bijvoorbeeld:

- adres van de student
- geboortedatum van de student
- datum van betaling

- naam van de docent die de cursussen geeft

## Normalisatieproces

In ons voorbeeld zie je duidelijk dat er enkele problemen zich voordoen. Om dit op te lossen hebben we het normalisatieproces die we dienen te doorlopen. In het normalisatieproces gebruiken we de normaalvormen om dit op te lossen.

In totaal zijn er 6 normaalvormen, maar voor de meeste database zijn de eerste 3 normaalvormen reeds voldoende. Deze 3 zullen we in detail bekijken.

## 0DE NORMAALVORM

In ons voorbeeld zie je duidelijk dat er enkele problemen zich voordoen. Om dit op te lossen hebben we het normalisatieproces die we dienen te doorlopen. In het normalisatieproces gebruiken we de normaalvormen om dit op te lossen. De 0de normaalvorm is niet echt een normaalvorm, maar we benoemen deze zo omdat we hier alle gegevens eerst gaan **inventariseren**.

Het inventariseren is het opsommen van alle data die je wel en niet ziet staan. D.w.z. de zichtbare data en de eventuele tekorten die je als database developer kan vaststellen. De inventarisatie is dus een lijst van veldnamen.

student\_naam  
prijs  
cursus\_naam  
cursus\_1  
cursus\_2  
cursus\_3

Eventuele tekorten. Het is perfect normaal dat je op dit niveau nog niet alle tekorten onmiddellijk kan vaststellen. Al de tekorten die je hier al kan vaststellen zijn natuurlijk reeds een meerwaarde. Wanneer je er hier geen kan vinden is ook dit geen probleem, dit wordt later in de normaalvormen toch duidelijk.

student\_naam  
**student\_geboortedatum**  
prijs  
cursus\_naam  
cursus\_1  
**cursus\_1\_docent**  
cursus\_2  
**cursus\_2\_docent**  
cursus\_3  
**cursus\_3\_docent**

## 1STE NORMAALVORM

Om de eerste normaalvorm toe te passen zetten we eerst alle velden uit de inventarisatie in een tabel.

| studentnaam     | geboortedatum | prijs | cursusnaam      | cursus 1          | cursus 1 docent | cursus 2    | cursus 2 docent | cursus 3    | cursus 3 docent |
|-----------------|---------------|-------|-----------------|-------------------|-----------------|-------------|-----------------|-------------|-----------------|
| Tom Vanhoutte   | 8-11-1973     | 100   | Full Stack      | PHP OOP           | Jan De Nerd     | Javascript  | Luc Mertens     | Laravel     | Jan De Nerd     |
| Tim Vanhoutte   | 10-2-1980     | 200   | Grafisch design | Photoshop 1       | Lucky Luke      | Photoshop 2 | An Debon        | Photoshop 3 | An Debon        |
| Nick Dewaele    | 5-1-1998      | 250   | Wetenschappen   | Fysica 1          | Irina Desmurf   | Fysica 2    | Jakke Boem      | Chemie      | Irina Desmurf   |
| Remco Evenepoel | 5-1-2000      | 150   | Fietshersteller | Afstellen fietsen | Eddy Planckaert |             |                 |             |                 |

We dienen ons nu de volgende vragen te stellen:

- Is de **COMBINATIE** van alle velden **TELKENS** een **UNIEKE RIJ**?
  - Hier is het antwoord duidelijk **NEEN**
- Welk **VELD** kan gebruikt worden om een **UNIEKE RIJ** te identificeren?
  - We hebben momenteel **GEEN** veld die een **UNIEKE RIJ** kan identificeren.
- **Verwijder alle rekenkundige velden.**

Wanneer we **GEEN VELD** hebben die een **UNIEKE RIJ** kan identificeren dan dienen we deze zelf aan te maken als **PRIMARY KEY**. Deze primary key zullen we **student\_id** noemen.

Daarnaast geven we ook een naam aan onze tabel, nl. **Student**

De schrijfwijze vanaf nu zullen we vereenvoudigen. We sommen de velden op i.p.v. ze weer te geven in een tabel. De primary sleutel van iedere tabel wordt **onderstreept** weergegeven.

**Student**(**student\_id**,studentnaam, geboortedatum, prijs, cursusnaam, cursus1, cursus 1 docent, cursus 2, cursus 2 docent, cursus3, cursus 3 docent)

## 2DE NORMAALVORM

Om de eerste normaalvorm toe te passen dien je de volgende regels te volgen:

- **Is de eerste normaalvorm voldaan?**
  - Hier is het antwoord duidelijk JA
- **Elk niet-sleutelattribuut moet functioneel afhankelijk zijn van de primaire sleutel.**  
Hier stel je jezelf de vraag: **“Zijn al de kolommen afhankelijk van en specifiek gerelateerd aan de primaire sleutel”?**

We hernemen onze tabel en bekijken deze van nabij:

**Student**(**student\_id**, studentnaam, geboortedatum, prijs, cursusnaam, cursus1, cursus 1 docent, cursus 2, cursus 2 docent, cursus3, cursus 3 docent)

Per kolom controleren we nu de functionele afhankelijkheid t.o.v. de primaire sleutel.

**studentnaam:**

is **studentnaam** functioneel afhankelijk van **student\_id**? **JA**, want een andere **student\_id** betekent een andere studentnaam.

**geboortedatum:**

is **geboortedatum** functioneel afhankelijk van **student\_id**? **JA**, want het is specifiek voor deze student.

**prijs:**

is **prijs** functioneel afhankelijk van **student\_id**? **NEEN**, want de prijs hangt af van de cursus.

**cursusnaam:**

is **cursusnaam** functioneel afhankelijk van **student\_id**? **NEEN**, de cursusnaam is niet afhankelijk.

**cursus 1:**

is **cursus 1** functioneel afhankelijk van **student\_id**? **NEEN**, voor een cursus kan meer dan één student worden ingeschreven.

**cursus 1 docent:**

is **cursus 1 docent** functioneel afhankelijk van **student\_id**? **NEEN**, want een docent geeft les aan meer dan één cursist.

**cursus 2:**

is **cursus 2** functioneel afhankelijk van **student\_id**? **NEEN**, voor een cursus kan meer dan één student worden ingeschreven.

**cursus 2 docent:**

is **cursus 2 docent** functioneel afhankelijk van **student\_id**? **NEEN**, want een docent geeft les aan meer dan één cursist.

**cursus 3:**

is **cursus 3** functioneel afhankelijk van **student\_id**? **NEEN**, voor een cursus kan meer dan één student worden ingeschreven.

**cursus 3 docent:**

is **cursus 3 docent** functioneel afhankelijk van **student\_id**? **NEEN**, want een docent geeft les aan meer dan één cursist.

## CURSUS MySQL

We VERWIJDEREN nu alle neen velden af van de tabel.

**Student(student\_id,studentnaam, geboortedatum)**

---

Nu bekijken we de **overgebleven velden**

cursusnaam, cursus1, cursus 1 docent, cursus 2, cursus 2 docent, cursus3, cursus 3 docent

Deze overgebleven namen stellen terug een nieuwe tabel voor. We dienen dus in de tweede normaalvorm hier opnieuw te beginnen en ons dezelfde vragen te stellen zoals voorheen totdat we niet meer verder kunnen opsplitsen.

Hier hebben we de tabel **cursus**, met als primary key **cursus\_id**

**Cursus(cursus\_id,cursusnaam, cursus1, cursus 1 docent, cursus 2, cursus 2 docent, cursus3, cursus 3 docent)**

Terug dienen we ons dezelfde vragen te stellen: “**is een kolom functioneel afhankelijk van de primary key cursus\_id ?**”

De tabel wordt dus:

**Cursus(cursus\_id, cursusnaam, prijs)**

---

Nu bekijken we de **overgebleven velden**

cursus1, cursus 1 docent, cursus 2, cursus 2 docent, cursus3, cursus 3 docent

Aangezien we reeds een tabel cursus hebben, kunnen we hier geen tweede tabel maken met dezelfde cursus, we geven dit dus een andere noemer, nl. vak. (cursus1, cursus 2 en cursus 3 worden dus hernoemd naar vak)

**Vak(vak\_id, vak)**

---

Nu bekijken we de **overgebleven velden**

cursus 1 docent docent, cursus 2 docent, cursus 3 docent

**Docent(docent\_id, naam)**

## CURSUS MySQL

Het voorlopig resultaat van tabellen momenteel is dus:

**Student**(student\_id, studentnaam, geboortedatum)

**Cursus**(cursus\_id, cursusnaam, prijs)

**Vak**(vak\_id, vak)

**Docent**(docent\_id, naam)

### RELATIES

We hebben reeds 3 relaties gezien:

- één op één
- één op veel
- veel op veel

Deze tabellen zijn nog niet gerelateerd met elkaar. Hiervoor gebruiken we onze vreemde sleutels.

Hier dien je je zelf vragen te stellen. Wat zijn de onderlinge relaties tussen de tabellen?

---

Student en Cursus.

Een student kan ingeschreven zijn in verschillende cursussen, maar een cursus kan ook verschillende vakken bevatten. Hier hebben we een veel op veel relatie.

**Student**(student\_id, studentnaam, geboortedatum)

**Student\_Cursus**(student\_id, cursus\_id)

**Cursus**(cursus\_id, cursusnaam, prijs)

---

Cursus en Vak

Een cursus kan bestaan uit meerdere vakken. Maar een vak kan ook in meerdere cursussen terechtkomen. Het benoemen van een tussentabel is een samentrekking van de tabelnamen. Een tussentabel bestaat ENKEL uit de FOREIGN KEYS van de buitenste tabellen. De samenstelling van deze FOREIGN KEYS vormt de PRIMARY KEY van de tussentabel.

**Cursus**(cursus\_id, cursusnaam, prijs)

**Cursus\_Vak**(cursus\_id, vak\_id)

**Vak**(vak\_id, vak)

---

Vak en docent

Een vak kan meerdere docenten hebben en een docent kan meerdere vakken geven.

Ook hier hebben we een veel op veel relatie.

**Vak**(vak\_id, vak)

**Vak\_Docent**(vak\_id, docent\_id)

**Docent**(docent\_id, naam)

## 3DE NORMAALVORM

De derde normaalvorm is in veel gevallen de laatste fase van het normalisatieproces. De regels hiervoor zijn:

- **Is de tweede normaalvorm voldaan?**
  - Hier is het antwoord duidelijk JA
- **Transitieve afhankelijkheid.** Iedere kolom moet ENKEL EN ALLEEN afhankelijk zijn van de primaire sleutel.

In ons voorbeeld is dit reeds het geval en is de derde normaalvorm overbodig. Maar we kunnen wel een voorbeeld geven.

Een adresveld wordt als volgt geschreven:

Steenstraat 1, 8820 Torhout

Dit zou resulteren in een aparte tabel:

Adres(adres\_id, adresnaam).

Maar adresnaam voldoet hier niet omdat er meerdere velden aanwezig zijn binnen adresnaam, nl. straat, nr, postcode en gemeente. Voor deze velden gebruiken we de 3de normaalvorm.

Hier kunnen we dus ook 3 tabellen maken:

Adres, Postcode en Gemeente.

Vervolgens dienen dan ook terug de relaties gelegd te worden net zoals in de tweede normaalvorm.

---

Hieronder kun je de SQL kopiëren om de database te simuleren:

```

SET SQL_MODE = "NO_AUTO_VALUE_ON_ZERO";
SET AUTOCOMMIT = 0;
START TRANSACTION;
SET time_zone = "+00:00";

/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
/*!40101 SET @OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESULTS */;
/*!40101 SET @OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTION */;
/*!40101 SET NAMES utf8mb4 */;

--
-- Database: `dbtest`
--

--
-- Table structure for table `cursus`
--

DROP TABLE IF EXISTS `cursus`;
```

**CURSUS MySQL**

```
CREATE TABLE IF NOT EXISTS `cursus` (
 `cursus_id` int(11) UNSIGNED NOT NULL AUTO_INCREMENT,
 `cursusnaam` varchar(255) NOT NULL,
 `prijs` decimal(10,2) NOT NULL,
 PRIMARY KEY (`cursus_id`)
) ENGINE=InnoDB AUTO_INCREMENT=5 DEFAULT CHARSET=latin1;

--
-- Dumping data for table `cursus`
--

INSERT INTO `cursus` (`cursus_id`, `cursusnaam`, `prijs`) VALUES
(1, 'Full Stack', '100.00'),
(2, 'Grafisch Design', '200.00'),
(3, 'Wetenschappen', '250.00'),
(4, 'Fietshersteller', '150.00');

--
-- Table structure for table `cursus_vak`
--

DROP TABLE IF EXISTS `cursus_vak`;
CREATE TABLE IF NOT EXISTS `cursus_vak` (
 `cursus_id` int(11) UNSIGNED DEFAULT NULL,
 `vak_id` int(11) UNSIGNED DEFAULT NULL,
 KEY `Cursusvak` (`cursus_id`),
 KEY `Vakcursus` (`vak_id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

--
-- Dumping data for table `cursus_vak`
--

INSERT INTO `cursus_vak` (`cursus_id`, `vak_id`) VALUES
(1, 1),
(1, 2),
(1, 3),
(2, 4),
(2, 5),
(2, 6),
(3, 7),
(3, 8),
(3, 9),
(4, 10);

--
-- Table structure for table `docent`
--
```

```

--

DROP TABLE IF EXISTS `docent`;
CREATE TABLE IF NOT EXISTS `docent` (
 `docent_id` int(11) UNSIGNED NOT NULL AUTO_INCREMENT,
 `naam` varchar(255) NOT NULL,
 PRIMARY KEY (`docent_id`)
) ENGINE=InnoDB AUTO_INCREMENT=8 DEFAULT CHARSET=latin1;

--
-- Dumping data for table `docent`
--

INSERT INTO `docent` (`docent_id`, `naam`) VALUES
(1, 'Jan De Nerd'),
(2, 'Lucky Luke'),
(3, 'Irina Desmurf'),
(4, 'Eddy Planckaert'),
(5, 'Luc Mertens'),
(6, 'An Debon'),
(7, 'Jakke Boem');

--
-- Table structure for table `student`
--

DROP TABLE IF EXISTS `student`;
CREATE TABLE IF NOT EXISTS `student` (
 `student_id` int(11) UNSIGNED NOT NULL AUTO_INCREMENT,
 `naam` varchar(255) DEFAULT NULL,
 `geboortedatum` date DEFAULT NULL,
 PRIMARY KEY (`student_id`)
) ENGINE=InnoDB AUTO_INCREMENT=5 DEFAULT CHARSET=latin1;

--
-- Dumping data for table `student`
--

INSERT INTO `student` (`student_id`, `naam`, `geboortedatum`) VALUES
(1, 'Tom Vanhoutte', '1973-08-11'),
(2, 'Tim Vanhoutte', '1980-02-10'),
(3, 'Nick Dewaele', '1998-01-05'),
(4, 'Remco Evenepoel', '2000-01-05');

--
-- Table structure for table `student_cursus`
--
```

```

DROP TABLE IF EXISTS `student_cursus`;
CREATE TABLE IF NOT EXISTS `student_cursus` (
 `student_id` int(11) UNSIGNED DEFAULT NULL,
 `cursus_id` int(11) UNSIGNED DEFAULT NULL,
 KEY `studentcursus` (`student_id`),
 KEY `cursusstudent` (`cursus_id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

--
-- Dumping data for table `student_cursus`
--

INSERT INTO `student_cursus` (`student_id`, `cursus_id`) VALUES
(1, 1),
(2, 2),
(3, 3),
(4, 4);

--
-- Table structure for table `vak`
--

DROP TABLE IF EXISTS `vak`;
CREATE TABLE IF NOT EXISTS `vak` (
 `vak_id` int(11) UNSIGNED NOT NULL AUTO_INCREMENT,
 `vak` varchar(255) NOT NULL,
 PRIMARY KEY (`vak_id`)
) ENGINE=InnoDB AUTO_INCREMENT=11 DEFAULT CHARSET=latin1;

--
-- Dumping data for table `vak`
--

INSERT INTO `vak` (`vak_id`, `vak`) VALUES
(1, 'PHP OOP'),
(2, 'Javascript'),
(3, 'Laravel'),
(4, 'Photoshop 1'),
(5, 'Photoshop 2'),
(6, 'Photoshop 3'),
(7, 'Fysica 1'),
(8, 'Fysica 2'),
(9, 'Chemie'),
(10, 'Afstellen fietsen');

--
```

**CURSUS MySQL**

```
-- Table structure for table `vak_docent`
--

DROP TABLE IF EXISTS `vak_docent`;
CREATE TABLE IF NOT EXISTS `vak_docent` (
 `vak_id` int(11) UNSIGNED DEFAULT NULL,
 `docent_id` int(11) UNSIGNED DEFAULT NULL,
 KEY `Vakdocent` (`vak_id`),
 KEY `Docentvak` (`docent_id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

--
-- Dumping data for table `vak_docent`
--

INSERT INTO `vak_docent` (`vak_id`, `docent_id`) VALUES
(1, 1),
(2, 5),
(3, 1),
(4, 2),
(5, 6),
(6, 6),
(7, 3),
(8, 7),
(9, 3),
(10, 4);

--
-- Constraints for dumped tables
--

--
-- Constraints for table `cursus_vak`
--

ALTER TABLE `cursus_vak`
 ADD CONSTRAINT `Cursusvak` FOREIGN KEY (`cursus_id`) REFERENCES
 `cursus`(`cursus_id`),
 ADD CONSTRAINT `Vakcursus` FOREIGN KEY (`vak_id`) REFERENCES `vak`
 (`vak_id`);

--
-- Constraints for table `student_cursus`
--

ALTER TABLE `student_cursus`
 ADD CONSTRAINT `cursusstudent` FOREIGN KEY (`cursus_id`) REFERENCES
 `cursus`(`cursus_id`),
 ADD CONSTRAINT `studentcursus` FOREIGN KEY (`student_id`) REFERENCES
 `student`(`student_id`);

--
-- Constraints for table `vak_docent`
```

**CURSUS MySQL**

```
--
ALTER TABLE `vak_docent`
 ADD CONSTRAINT `Docentvak` FOREIGN KEY (`docent_id`) REFERENCES
 `docent`(`docent_id`),
 ADD CONSTRAINT `Vakdocent` FOREIGN KEY (`vak_id`) REFERENCES `vak`
 (`vak_id`);
COMMIT;

/*!40101 SET CHARACTER_SET_CLIENT=@OLD_CHARACTER_SET_CLIENT */;
/*!40101 SET CHARACTER_SET_RESULTS=@OLD_CHARACTER_SET_RESULTS */;
/*!40101 SET COLLATION_CONNECTION=@OLD_COLLATION_CONNECTION */;
```

In de context van databases, en specifiek MySQL, verwijst het toepassen van security vaak naar het implementeren van maatregelen die ervoor zorgen dat de data veilig is en dat er geen ongeautoriseerde toegang plaatsvindt. Dit omvat onder andere:

1. **SQL Injection voorkomen:** Dit is een veelvoorkomende kwetsbaarheid waarbij aanvallers kwaadaardige SQL-code invoeren in inputvelden. De cursus lijkt te vragen dat studenten begrijpen hoe ze hun queries kunnen beschermen tegen SQL Injection door bijvoorbeeld prepared statements en parameterized queries te gebruiken.
2. **Correct opslaan van wachtwoorden:** Dit verwijst naar het gebruik van hash-algoritmes (zoals bcrypt of SHA-256) om wachtwoorden veilig op te slaan, in plaats van ze in platte tekst in de database te bewaren.

Een **SQL injection** ontstaat wanneer een webapplicatie gebruikersinvoer direct in een SQL-query verwerkt zonder deze invoer eerst te valideren of te ontsmetten. Kwaadaardige gebruikers kunnen speciaal geformatteerde invoer geven die de query manipuleert om ongewenste acties uit te voeren, zoals toegang krijgen tot gevoelige gegevens of zelfs de hele database te compromitteren.

Hier is hoe het proces werkt en hoe een SQL injection uiteindelijk in een database terecht kan komen:

## 1. Invoer vanuit een gebruikersinterface

Een webapplicatie biedt vaak invoervelden aan, zoals een login-formulier, zoekbalk, of URL-parameters, waarin een gebruiker gegevens kan invoeren. Als deze invoer direct in een SQL-query wordt opgenomen zonder te worden gecontroleerd, kan de gebruiker kwaadaardige SQL-code invoeren.

**Voorbeeld:** Stel dat je een zoekformulier hebt waarin gebruikers een naam kunnen invoeren om in een database te zoeken:

```
<form action="/search" method="POST">
 <input type="text" name="name" />
 <input type="submit" value="Search" />
</form>
```

## 2. Kwetsbare SQL-query in de applicatie

Wanneer de invoer van de gebruiker direct wordt opgenomen in een SQL-query zonder enige validatie of ontsmetting, kan een aanvaller deze kans gebruiken om de SQL-code te injecteren.

Hier is een slecht voorbeeld van een SQL-query waarin gebruikersinvoer direct in de query wordt gebruikt:

```
$name = $_POST['name'];
$query = "SELECT * FROM users WHERE name = '$name'";
```

In dit geval wordt de waarde van \$name (de invoer van de gebruiker) rechtstreeks in de query geplaatst, zonder enige vorm van ontsmetting of voorbereiding.

## 3. Kwaadaardige invoer van de gebruiker

Een aanvaller kan een speciaal geformatteerde invoer geven, bijvoorbeeld:

```
' OR '1'='1
```

Dit zou betekenen dat de uiteindelijke SQL-query er als volgt uitziet:

```
SELECT * FROM users WHERE name = '' OR '1'='1';
```

In plaats van alleen de gebruiker met een specifieke naam op te zoeken, retourneert deze query alle records, omdat de voorwaarde '`'1'='1'`' altijd waar is. Dit geeft de aanvaller toegang tot alle gebruikersgegevens, wat niet de bedoeling is.

## 4. Manipuleren van data in de database

Een aanvaller kan verder gaan en invoer geven die niet alleen gegevens ophaalt, maar ook wijzigingen aanbrengt in de database. Een veelvoorkomend doel is om de database te laten denken dat de aanvaller een geautoriseerde gebruiker is, of om gegevens te wijzigen of te verwijderen.

**Kwaadaardige invoer voor data-manipulatie:**

```
'; DELETE FROM users; --
```

Hier wordt de SQL-query afgesloten met een apostrof ('), waarna de aanvaller een tweede opdracht toevoegt om alle gegevens uit de `users`-tabel te verwijderen. De uiteindelijke query zou er zo uitzien:

```
SELECT * FROM users WHERE name = ''; DELETE FROM users; --;
```

Het eerste deel van de query is een lege zoekopdracht, en het tweede deel verwijdert de gehele `users`-tabel. De -- markeert het begin van een SQL-opmerking, waardoor de rest van de oorspronkelijke query wordt genegeerd.

## 5. Hoe SQL Injection in de database komt

De SQL injection komt “terecht” in de database doordat de invoer van de gebruiker direct in de SQL-query wordt verwerkt, waardoor de aanvaller controle krijgt over hoe de query wordt uitgevoerd. Dit kan leiden tot:

- **Ongeoorloofde toegang** tot gegevens, bijvoorbeeld door alle gegevens uit een tabel op te vragen.
- **Wijzigen of verwijderen van gegevens**, door invoer te manipuleren om bewerkingen uit te voeren, zoals het verwijderen van records of het wijzigen van belangrijke gegevens.
- **Wachtwoordaanvallen**, waarbij de aanvaller via SQL injection probeert wachtwoorden of andere gevoelige gegevens te stelen.

## Hoe te voorkomen?

1. **Prepared Statements gebruiken:** Dit zorgt ervoor dat gebruikersinvoer als gegevens wordt behandeld en niet als SQL-code.
2. **Invoervalidatie:** Valideer en ontsmet alle invoer die van gebruikers komt.
3. **Minimale rechten:** Geef databasegebruikers alleen de noodzakelijke rechten (bijv. alleen leesrechten voor bepaalde acties).
4. **Stored Procedures:** Gebruik indien mogelijk stored procedures die minder vatbaar zijn voor SQL injection.

Door deze maatregelen te nemen, kun je voorkomen dat een SQL injection de database beschadigt of misbruikt.