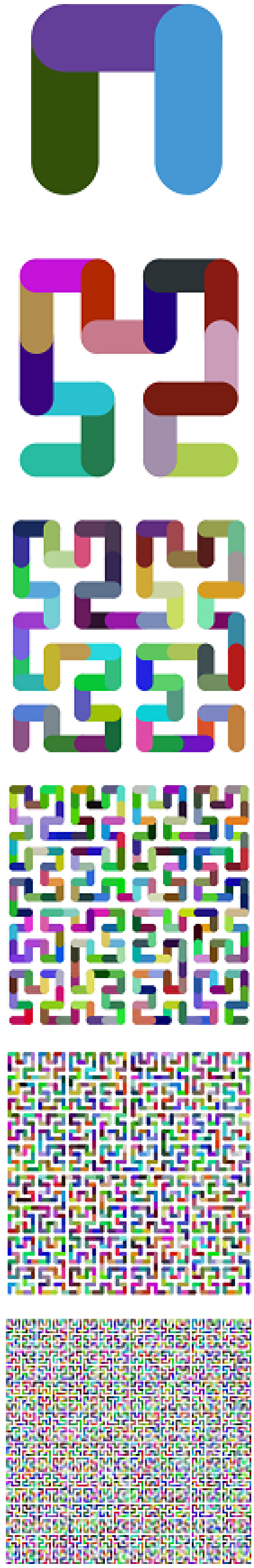
SELECT

# SQL

FROM

WHERE



# ORDER BY

remie.woudt@gmail.com
a.salimian@broklede.nl

**De afbeeldingen op het voorblad zijn van Xufan Zhang**

My "artistic" creation is Hilbert curve, a curve which is able to cover the whole plane, or well known as a "2D curve". I define 4 types of "U" shapes which together form the curve. The key point is to draw the bridges across all curves. (Actually, it is the bridges that form the whole curve.) To add some taste, I incorporated random colors and "case-dependent" line thickness.

Bron: http://www.cs.princeton.edu/courses/archive/fall10/cos126/art/index.php

# Inhoud

# 1

# SQL

# 1.1 Wat is SQL?

Als we op grote schaal gegevens op willen slaan gebruiken we daar meestal een database voor. Er zijn verschillende soorten databases die je ook op verschillende manieren moet benaderen om iets met die gegevens te gaan doen.

Een veel gebruikte type database is de **relationele database**. Deze wordt zo genoemd omdat de gegevens zijn ingedeeld in groepen en binnen die groepen hebben die gegevens een relatie met elkaar.
Zo zal er, als je de gegevens van een persoon erin opslaat, een relatie bestaan tussen de voornaam en de achternaam.

De meest gebruikte taal om een relationele database te benaderen om de gegevens in te voeren, op te vragen, te wijzigen of te verwijderen is SQL.
**SQL** is een afkorting voor **S**tructured **Q**uery **L**anguage.

SQL kun je onderverdelen in drie soorten:

- SQL-DDL, de SQL **D**ata **D**efinition **L**anguage
- SQL-DML, de SQL **D**ata **M**anipulation **L**anguage
- SQL-DCL, de SQL **D**ata **C**ontrol **L**anguage.

De belangrijkste voor ons is de **SQL-DML**. De SQL-DLC zal hier niet worden behandeld.

# 1.2 Hoe gaan we te werk?

Om met SQL te gaan werken hebben we tenminste het volgende nodig:

- Een databaseprogramma
- Gegevens van de tabellen waarmee we willen gaan werken

Een eenvoudig maar veel gebruikt databaseprogramma is **SQLite**. SQLite kan zowel onder Windows, MacOSX en Linux draaien. Ook wordt SQLite veel gebruikt in websites en in mobiele telefoons.
En het handige is dat we SQLite met wat kleine hulpprogramma's mooi op een usb-stick kunnen installeren.

Aangezien we later SQLite samen met PHP gaan gebruiken hebben we daarvoor een webserver nodig dus die gaan we maar meteen mee installeren.
Het zit allemaal in WebServer.

***Opdracht 1***

- Als je WebServer nog niet geïnstalleerd hebt volg je de handleiding **hier**.

Je start de webserver met het scriptje `Start Web Server.vbs`
en je stopt hem weer met `Stop Web Server.vbs`.

Tijdens het starten kun je de melding krijgen of je Nginx toegang wilt geven om door de firewall te mogen komen.
Dat heb je niet nodig dus annuleer die melding.

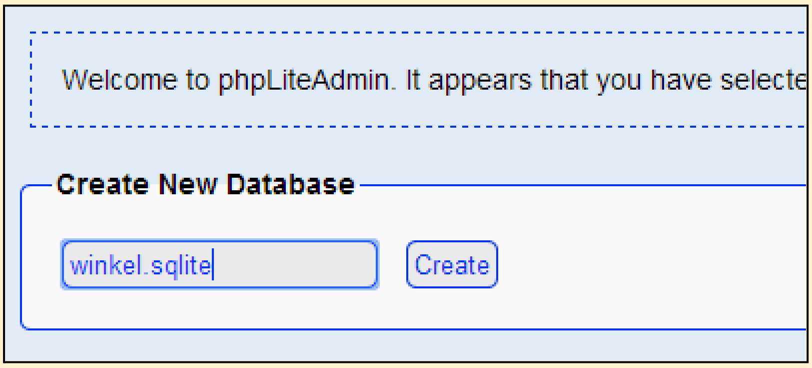
## 1.3 phpLiteAdmin gebruiken

Een database bestaat meestal uit een aantal tabellen die bij elkaar horen. Zo zal een winkeldatabase vaak bestaan uit een tabel met de klanten. Maar ook een tabel met de artikelen. En misschien ook wel een tabel met de artikelen die de klanten besteld hebben.
Om met SQLite te kunnen werken hebben we dus eerst een database nodig en daarna gaan we daarbinnen de tabellen plaatsen.

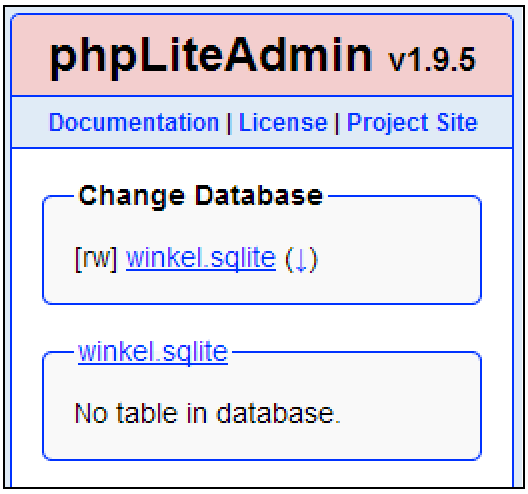
***Opdracht 2***

- Start phpLiteAdmin door WebServer te starten (als die nog niet gestart is) en dan in het browservenster in te typen:
  `http://localhost/phpliteadmin/phpliteadmin.php`

- Je ziet dan het volgende:




- Typ nu, net als hierboven, de naam van de database in: **winkel.sqlite** en klik op **Create**




*Afbeelding 1: De database is gemaakt*

Als het goed gedaan is zie je nu aan de linkerzijde de naam van de database staan (zie ook afbeelding 1).

We gaan nu de tabellen en de gegevens toevoegen.

## *Opdracht 3*

- Selecteer en kopieer de onderstaande coderegels
- Ga daarna naar phpLiteAdmin, klik op **SQL** en plak deze regels in het venster dat is verschenen (zie afbeelding 2)
- Klik dan op **Go** en de gegevens worden geïmporteerd

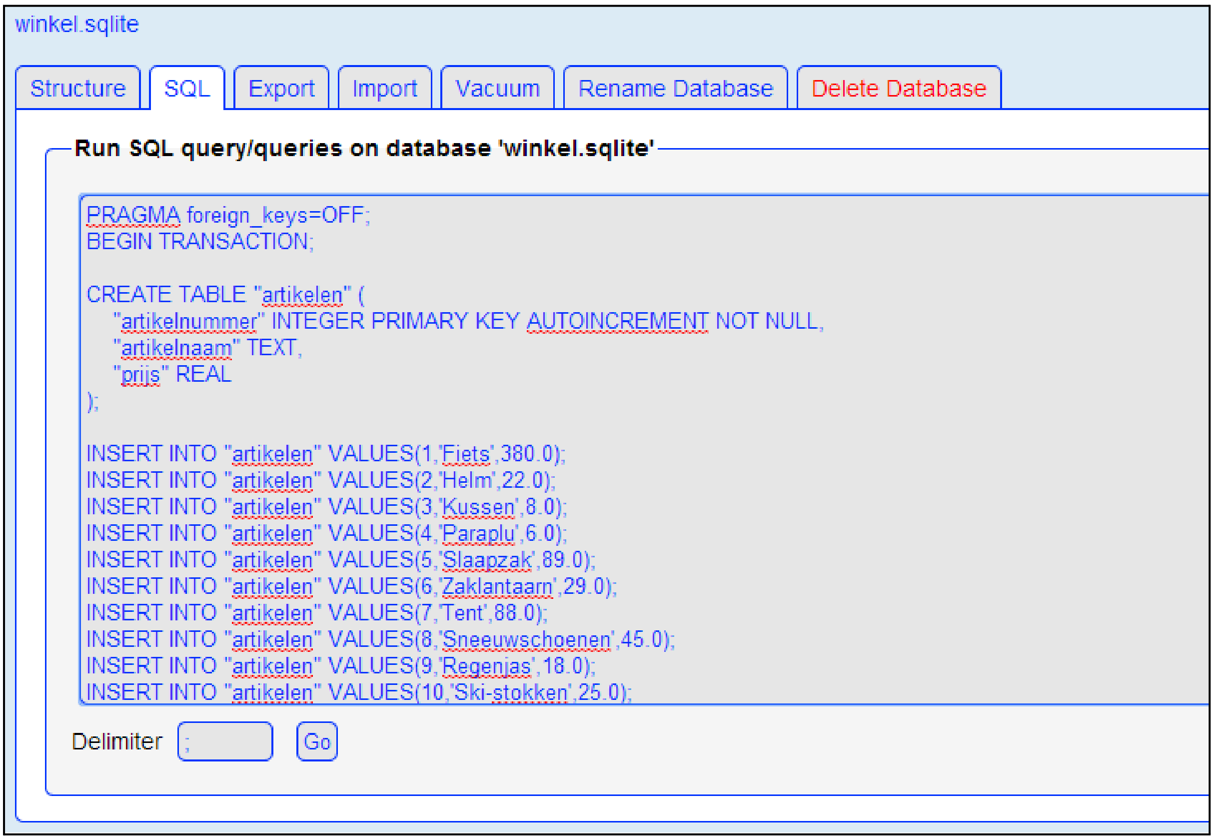
- Klik op **Structure** en je ziet dat er nu 3 tabellen in de database staan

- Door op **Browse** te klikken of links op een tabel zie je welke gegevens in die tabel staan.

```
PRAGMA foreign_keys=OFF;
BEGIN TRANSACTION;

CREATE TABLE "artikelen" (
    "artikelnummer" INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
    "artikelnaam" TEXT,
    "prijs" REAL
);

INSERT INTO "artikelen" VALUES(1,'Fiets',380.0);
INSERT INTO "artikelen" VALUES(2,'Helm',22.0);
INSERT INTO "artikelen" VALUES(3,'Kussen',8.0);
INSERT INTO "artikelen" VALUES(4,'Paraplu',6.0);
INSERT INTO "artikelen" VALUES(5,'Slaapzak',89.0);
INSERT INTO "artikelen" VALUES(6,'Zaklantaarn',29.0);
INSERT INTO "artikelen" VALUES(7,'Tent',88.0);
INSERT INTO "artikelen" VALUES(8,'Sneeuwschoenen',45.0);
INSERT INTO "artikelen" VALUES(9,'Regenjas',18.0);
INSERT INTO "artikelen" VALUES(10,'Ski-stokken',25.0);
INSERT INTO "artikelen" VALUES(11,'Eenwieler',180.0);
INSERT INTO "artikelen" VALUES(12,'Paraplu',4.0);
INSERT INTO "artikelen" VALUES(13,'Parachute',1250.0);
INSERT INTO "artikelen" VALUES(14,'Reddingsvest',125.0);
INSERT INTO "artikelen" VALUES(15,'Skateboard',33.0);
INSERT INTO "artikelen" VALUES(16,'Vlot',58.0);
INSERT INTO "artikelen" VALUES(17,'Pogo stick',28.0);
INSERT INTO "artikelen" VALUES(18,'Kano peddel',40.0);
INSERT INTO "artikelen" VALUES(19,'Oorwarmers',12.5);
INSERT INTO "artikelen" VALUES(20,'Sneeuwschuiver',16.75);
INSERT INTO "artikelen" VALUES(21,'Kano',280.0);
INSERT INTO "artikelen" VALUES(22,'Hoola-hoop',14.0);
INSERT INTO "artikelen" VALUES(23,'Zaklantaarn',28.0);
INSERT INTO "artikelen" VALUES(24,'Zaklantaarn',16.0);
INSERT INTO "artikelen" VALUES(25,'Luchtbed',38.0);
INSERT INTO "artikelen" VALUES(26,'Tent',79.0);
INSERT INTO "artikelen" VALUES(27,'Tuinstoel',32.0);
INSERT INTO "artikelen" VALUES(28,'Eenwieler',192.0);
INSERT INTO "artikelen" VALUES(29,'Kompas',8.0);
INSERT INTO "artikelen" VALUES(30,'Zaklantaarn',4.0);
INSERT INTO "artikelen" VALUES(31,'Slaapzak',88.0);
```

```sql
INSERT INTO "artikelen" VALUES(32,'Zakmes',22.0);
INSERT INTO "artikelen" VALUES(33,'Kano peddel',40.0);
INSERT INTO "artikelen" VALUES(34,'Oorwarmers',12.0);
INSERT INTO "artikelen" VALUES(35,'Sneeuwschuiver',18.15);

CREATE TABLE "bestellingen" (
    "bestellingnummer" INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
    "klantnummer" INTEGER,
    "artikelnummer" INTEGER,
    "order_datum" TEXT,
    "hoeveelheid" INTEGER
);

INSERT INTO "bestellingen" VALUES(1,1,12,'1999-12-15',1);
INSERT INTO "bestellingen" VALUES(2,2,4,'1999-12-01',1);
INSERT INTO "bestellingen" VALUES(3,3,8,'1999-11-02',1);
INSERT INTO "bestellingen" VALUES(4,3,25,'1999-11-01',1);
INSERT INTO "bestellingen" VALUES(5,15,6,'1999-10-28',1);
INSERT INTO "bestellingen" VALUES(6,8,32,'1999-09-19',2);
INSERT INTO "bestellingen" VALUES(7,1,1,'1999-09-18',1);
INSERT INTO "bestellingen" VALUES(8,14,12,'1999-09-01',1);
INSERT INTO "bestellingen" VALUES(9,12,5,'1999-08-18',1);
INSERT INTO "bestellingen" VALUES(10,17,14,'1999-08-14',2);
INSERT INTO "bestellingen" VALUES(11,16,28,'1999-08-13',1);
INSERT INTO "bestellingen" VALUES(12,8,27,'1999-07-27',1);
INSERT INTO "bestellingen" VALUES(13,7,25,'1999-07-06',1);
INSERT INTO "bestellingen" VALUES(14,8,4,'1999-07-01',4);
INSERT INTO "bestellingen" VALUES(15,10,1,'1999-07-01',1);
INSERT INTO "bestellingen" VALUES(16,17,32,'1999-06-30',1);
INSERT INTO "bestellingen" VALUES(17,1,13,'1999-06-30',1);
INSERT INTO "bestellingen" VALUES(18,1,14,'0000-00-00',2);
INSERT INTO "bestellingen" VALUES(19,2,15,'0000-00-00',1);
INSERT INTO "bestellingen" VALUES(20,7,8,'0000-00-00',1);
INSERT INTO "bestellingen" VALUES(21,8,16,'1999-12-22',1);
INSERT INTO "bestellingen" VALUES(22,2,17,'1999-12-30',3);
INSERT INTO "bestellingen" VALUES(23,15,18,'2000-01-01',4);
INSERT INTO "bestellingen" VALUES(24,17,19,'2000-01-02',1);
INSERT INTO "bestellingen" VALUES(25,2,21,'2000-01-18',1);
INSERT INTO "bestellingen" VALUES(26,4,22,'2000-01-18',1);
INSERT INTO "bestellingen" VALUES(27,5,24,'2000-01-19',4);
INSERT INTO "bestellingen" VALUES(28,1,6,'2000-01-30',1);
INSERT INTO "bestellingen" VALUES(29,15,34,'2000-02-02',1);
INSERT INTO "bestellingen" VALUES(30,12,20,'2000-02-29',1);
INSERT INTO "bestellingen" VALUES(31,10,22,'2000-03-08',2);
INSERT INTO "bestellingen" VALUES(32,7,1,'2000-03-18',1);
INSERT INTO "bestellingen" VALUES(33,9,9,'2000-03-19',2);
```

```sql
INSERT INTO "bestellingen" VALUES(34,3,29,'2000-04-01',1);
INSERT INTO "bestellingen" VALUES(35,2,30,'2000-04-19',1);

CREATE TABLE "klanten" (
    "klantnummer" INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
    "voornaam" TEXT,
    "achternaam" TEXT,
    "woonplaats" TEXT,
    "provincie" TEXT
);

INSERT INTO "klanten" VALUES(1,'Jan','Grijs','Sneek','Friesland');
INSERT INTO "klanten" VALUES(2,'Harry','Bruin','Baarn','Utrecht');
INSERT INTO "klanten"
VALUES(3,'Eddy','Kelder','Snakkeburen','Friesland');
INSERT INTO "klanten"
VALUES(4,'Lisa','Jansen','Purmerend','Noord-Holland');
INSERT INTO "klanten" VALUES(5,'Ginger','Bergsma','Utrecht','Utrecht');
INSERT INTO "klanten" VALUES(6,'Kelly','Witte','Almelo','Overijssel');
INSERT INTO "klanten" VALUES(7,'Herbert','Dalton','Almere','Flevoland');
INSERT INTO "klanten"
VALUES(8,'Michael','Veenstra','Maarssen','Utrecht');
INSERT INTO "klanten" VALUES(9,'Anton','Vergo','Woerden','Zuid-Holland');
INSERT INTO "klanten"
VALUES(10,'Peter','Elegast','Rotterdam','Zuid-Holland');
INSERT INTO "klanten"
VALUES(11,'Marianne','Spaarzaam','Middelburg','Zeeland');
INSERT INTO "klanten"
VALUES(12,'Donald','Jongsma','Aalten','Gelderland');
INSERT INTO "klanten" VALUES(13,'Linda','Salamon','Vinkeveen','Utrecht');
INSERT INTO "klanten"
VALUES(14,'Sarah','Graham','Bredevoort','Gelderland');
INSERT INTO "klanten"
VALUES(15,'Kevin','Smit','Delfstrahuizen','Friesland');
INSERT INTO "klanten"
VALUES(16,'Conrad','Klaren','Leiden','Zuid-Holland');
INSERT INTO "klanten"
VALUES(17,'Isabelle','Zandman','Utrecht','Utrecht');

DELETE FROM sqlite_sequence;
INSERT INTO "sqlite_sequence" VALUES('artikelen',35);
INSERT INTO "sqlite_sequence" VALUES('bestellingen',35);
INSERT INTO "sqlite_sequence" VALUES('klanten',17);
COMMIT;
```




*Afbeelding 2: Gegevens importeren in phpLiteAdmin*

## 1.4 Data Definition Language

In de code hiervoor zie je o.a. de volgende SQL opdracht:

```
CREATE TABLE "artikelen" (
    "artikelnummer" INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
    "artikelnaam" TEXT,
    "prijs" REAL
);
```

**CREATE TABLE** is één van de SQL-DDL opdrachten. Met de DDL opdrachten kun je databases en tabellen laten aanmaken. Maar je kunt ze ook wijzigen en verwijderen. De belangrijkste SQL-DDL opdrachten zijn: **CREATE TABLE**, **ALTER TABLE** en **DROP TABLE**. (Er zijn overigens meerdere Data Definition Languages, bijvoorbeeld voor XML, vandaar de toevoeging SQL ervoor.)

Wij hoeven echter niets van die SQL-DDL opdrachten te weten want we kunnen ze lekker door een programma als phpLiteAdmin laten uitvoeren. Dat is veel gebruiksvriendelijker.
Maar anders is het met de SQL-DML.

## 1.5 Data Manipulation Language

SQL-DML bestaat uit de opdrachten **`SELECT`**, **`INSERT`**, **`UPDATE`** en **`DELETE`**.
Met **`INSERT`** voeg je gegevens toe, met **`UPDATE`** wijzig je gegevens en met **`DELETE`** verwijder je gegevens. Maar **`SELECT`** is de belangrijkste. Daarmee kunnen we gegevens uit de tabellen opvragen.

In opdracht 5 hebben we de tabellen aan laten maken. Dat betekent dat we nu de beschikking hebben over een bestand om SQL mee te oefenen. Maar voor we aan de slag gaan is het goed nog even te kijken naar de structuur van de tabellen die we nu in de database **winkel.sqlite** hebben. Want om goed te kunnen oefenen moet je de velden kennen.

In het bestand **winkel.sqlite** staan nu de volgende 3 tabellen:

| **Artikelen** | **Bestellingen** | **Klanten** |
|---|---|---|
| <u>artikelnummer</u> | <u>bestellingnummer</u> | <u>klantnummer</u> |
| artikelnaam | klantnummer | voornaam |
| prijs | artikelnummer | achternaam |
| | order_datum | woonplaats |
| | hoeveelheid | provincie |

Die tabellen horen bij elkaar. Zo zie je bepaalde benamingen zoals <u>klantnummer</u> en <u>artikelnummer</u> in meerdere tabellen voorkomen. Zo wordt met klantnummer in de tabel bestellingen aangegeven voor welke klant die bestelling is. En met artikelnummer in de tabel bestellingen wordt aangegeven om welk artikel het gaat.

In de **`SELECT`** opdrachten zullen we gebruik maken van de namen van de velden en wanneer we gegevens tegelijkertijd uit meerdere tabellen willen halen moeten we rekening houden met die koppelingen tussen de tabellen.

# 1.6 SELECT

De `SELECT` opdracht wordt gebruikt om **zoekopdrachten** aan de database te geven en het **resultaat** van die zoekopdracht te laten zien.

De `SELECT` opdracht heeft vijf bijbehorende deelopdrachten die gebruikt kunnen worden bij het zoeken, maar alleen de `FROM` deelopdracht is verplicht. Ieder van deze vijf kan weer opties, parameters enz. hebben.

Hier een overzicht van de `SELECT` opdracht:

```
SELECT [ALL | DISTINCT] [ * | veld1[,veld2]
FROM tabel1[,tabel2]
[WHERE voorwaarden]
[GROUP BY veld-lijst]
[HAVING voorwaarden]
[ORDER BY veld-lijst [ASC | DESC] ];
```

Staat iets tussen rechte haken zoals `[,veld2]` dan is dat niet verplicht. Staat er een verticaal streepje tussen zoals bij `[ALL | DISTINCT]` dan moet je uit één van beide kiezen (als je die optie gebruikt). De diverse mogelijkheden van de `SELECT` opdracht zullen in de volgende paragrafen aan de orde komen.

*Voorbeeld:*
Veronderstel dat je uit de tabel artikelen die artikelen wilt zien waarvan de prijs lager dan € 10 is. De opdracht luidt dan als volgt:

```
SELECT *
FROM artikelen
WHERE prijs <10;
```

Met `SELECT *` wordt aangegeven dat je alle velden van de tabellen die bij `FROM` staan aangegeven wilt zien. Je kunt hier ook een deel van de velden aangeven, bijvoorbeeld `SELECT artikelnaam`.
Met `FROM bestellingen` geef je aan uit welke tabel de gegevens gehaald moeten worden.
Tenslotte zie je dat met `WHERE prijs <10;` de voorwaarde gegeven wordt waaraan de informatie moet voldoen.

***Opdracht 4***

Test dit als volgt met phpLiteAdmin uit:

- Typ in het **SQL** scherm:
  `SELECT * FROM artikelen WHERE prijs <10;`

- Druk op **Go**

Als het goed is zie je (als je hetzelfde bestand hebt gebruikt) de gegevens van 5 artikelen waarvan de prijs lager is dan 10.
De `;` aan het eind van de regel is eigenlijk verplicht. Maar als je hem in phpLiteAdmin vergeet doet hij het ook prima.

*Samengevat:*

- Achter **`SELECT`** komen dus de **velden** die je wilt opvragen.
- Achter **`FROM`** komen de **tabellen** waaruit de gegevens moeten komen.
- Achter **`WHERE`** komen de **voorwaarden** waaraan de gegevens moeten voldoen.

Bij een **`WHERE`** opdracht gebruik je vaak een vergelijkingsoperator. Hieronder een overzicht van de operatoren die je kunt gebruiken:

| Operator | Betekenis |
|---|---|
| `=` | is gelijk aan |
| `>` | is groter dan |
| `<` | is kleiner dan |
| `>=` | is groter dan of gelijk aan |
| `<=` | is kleiner dan of gelijk aan |
| `<> of !=` | is ongelijk aan |
| `LIKE` | vergelijkt (een deel van) de tekst |

De meeste van de operatoren in deze tabel zullen wel duidelijk zijn. **`LIKE`** wordt hieronder met een voorbeeld toegelicht.
Veronderstel weer de tabel artikelen.

```sql
SELECT *
FROM artikelen
WHERE artikelnaam LIKE 'Pa%';
```

Hier wordt een selectie gemaakt op artikel. Iedere artikel waarvan de naam begint met **Pa** komt in de selectie. Dus of het nu om een **Pa**raplu gaat of om een **Pa**rachute, het procentteken geeft aan dat wat na Pa komt alles mag zijn.

Het procentteken mag ook vóór de vergelijkingstekst komen. Bijvoorbeeld:

```sql
SELECT *
FROM artikelen
WHERE artikelnaam LIKE '%ch%';
```

In dit geval zal hij o.a. Sneeuws**ch**oenen maar ook Lu**ch**tbed selecteren omdat in beide het tekstgedeelte **ch** voorkomt.

**ALL** selecteert alle records terwijl * alle velden van de geselecteerde records laat zien.
**ALL** en **DISTINCT** worden gebruikt om alle of juist unieke records op te vragen. **ALL** is de standaard (dat noemt men ook wel default waarde). Als je hier niets vermeldt zal hij alle records laten zien die aan de selectiecriteria voldoen. Dus eigenlijk hoef je **ALL** niet te gebruiken. Gebruik je **DISTINCT** dan geeft je opdracht dat de informatie die je opvraagt uniek moet zijn.
Als je bijvoorbeeld wilt weten uit welke provincies de klanten komen dan kun je de volgende select opdracht laten uitvoeren:

```sql
SELECT DISTINCT provincie
FROM klanten;
```

Dit geeft een heel andere uitvoer dan:

```sql
SELECT provincie
FROM klanten;
```

***Opdracht 5***

- Test beide hierboven gegeven queries uit en bekijk het verschil in uitvoer.

### *Opdracht 6*

- Geef de **SELECT** opdracht waarmee je een lijst krijgt van alle bestellingen van de klant met klantnummer 17.
  Laat daarbij het klantnummer, het artikelnummer en de hoeveelheid zien.

### *Opdracht 7*

- Geef de **SELECT** opdracht waarmee je alle velden ziet van de artikelen met de artikelnaam Tent.

### *Opdracht 8*

- Geef de **SELECT** opdracht waarmee je de velden klantnummer, order_datum en artikelnummer ziet voor alle bestellingen die in 1999 geplaatst zijn.

### *Opdracht 9*

- Geef de **SELECT** opdracht waarmee je de unieke artikelnamen laat zien.

### *Opdracht 10*

- Geef de **SELECT** opdracht waarmee je een lijst krijgt van alle klanten die in een provincie wonen die begint met de letter “F”.

***Opdracht II***

- Geef de **SELECT** opdracht waarmee je het klantnummer of de klantnummers krijgt die als voornaam "Jan" hebben.

Soms is het lastig bij query's dat SQLite hoofdlettergevoelig is. Maar dat kun je op de volgende manier oplossen:

```
SELECT * FROM artikelen WHERE UPPER(artikelnaam) = 'TENT';
SELECT * FROM artikelen WHERE LOWER(artikelnaam) = 'tent';
```

of als het om een deel van de artikelnaam gaat:

```
SELECT * FROM artikelen WHERE UPPER(artikelnaam) LIKE '%EN%';
SELECT * FROM artikelen WHERE LOWER(artikelnaam) LIKE '%en%';
```

Maar het kan nog eenvoudiger:

```
SELECT * FROM artikelen WHERE artikelnaam LIKE 'tent';
```

geeft ook het gewenste resultaat.

# 1.7 Tabellen combineren met Join

Misschien wel de belangrijkste SQL opdracht, de **join**, alhoewel het helemaal geen opdracht is zoals we ze eerder gehad hebben.
Met een join kunnen we gegevens uit meerdere tabellen opvragen.

Tijdens het gebruik van de bestanden heb je al gezien dat er een klantentabel, een bestellingentabel en een artikeltabel is. In de bestellingen tabel zit het veld klantnummer waarmee de koppeling wordt aangegeven van de klant die die bestelling geplaatst heeft. Maar als je de bestelling wilt verzenden heb je er niet zoveel aan als je alleen het klantnummer hebt. Je wilt dan natuurlijk ook de adresgegevens weten. En die zul je dan op moeten zoeken in de klantentabel. Zoiets doe je met een join.

Er is sprake van een join als je meer dan één tabel achter **FROM** hebt staan. Probeer onderstaande opdracht maar eens:

```
SELECT voornaam, achternaam, woonplaats, artikelnummer
FROM klanten, bestellingen
WHERE klanten.klantnummer = bestellingen.klantnummer;
```

Als het goed gegaan is krijg je de voornaam, de achternaam, de woonplaats en het artikelnummer dat de betreffende klant besteld heeft. Heel belangrijk daarbij is dat je in de **WHERE** opdracht aangeeft hoe de beide tabellen met elkaar verbonden zijn. Kortom, SQL mag alleen een regel laten zien wanneer het klantnummer in de tabel klanten gelijk is aan het klantnummer in de bestellingen tabel. Want wat gebeurt er als je dat niet doet? Probeer deze maar eens:

```
SELECT voornaam, achternaam, woonplaats, artikelnummer
FROM klanten, bestellingen;
```

Je krijgt nu een veel langere lijst. SQL heeft iedere regel uit de klantentabel gekoppeld aan iedere regel uit de bestellingentabel. En dat is niet de bedoeling!

Nog één belangrijk punt. Als je probeert een veld op te vragen dat in beide tabellen voorkomt, moet je dat veld aanduiden met de tabelnaam, dan een punt en dan de veldnaam. Wil je met deze tabellen het klantnummer ook opvragen dan gaat het er dus als volgt uitzien:

```
SELECT klanten.klantnummer, voornaam, achternaam, woonplaats,
artikelnummer
FROM klanten, bestellingen
WHERE klanten.klantnummer = bestellingen.klantnummer;
```

I.p.v. **klanten.klantnummer** in de **SELECT** regel, had je hier ook **bestellingen.klantnummer** kunnen typen. Het resultaat is hetzelfde.

***Opdracht 12***

- Geef de **SELECT** opdracht om de volgende velden te laten zien: klantnummer, voornaam, achternaam, order_datum, artikelnummer, hoeveelheid voor alles wat iedere klant heeft besteld.

***Opdracht 13***

- Geef de **SELECT** opdracht om de volgende velden te laten zien: alle bestelgegevens en de bijbehorende artikelnaam.

***Opdracht 14***

- Geef de **SELECT** opdracht om alle bestellingen van klanten uit Utrecht te laten zien.

***Opdracht 15***

- Geef de **SELECT** opdracht om alle gegevens uit de tabel klanten te laten zien van klanten die een zaklantaarn bestelden.
  (Let op, hier komen dus drie tabellen achter **FROM** en dus ook een dubbele join. Zet daarvoor tussen de beide joins het woordje **AND.** Zie ook 1.12).

## 1.8 De aggregaat functies

Aggregaat functies worden gebruikt om te kunnen rekenen aan de kolommen (onder een kolom verstaan we alle waarden van een bepaald veld) die met de **SELECT** opdracht worden opgehaald. Het gaat dan meestal om optellen, gemiddelden enz.

| Functie | Betekenis |
|---|---|
| **MIN** | geeft de **laagste** waarde van een kolom |

| | |
|---|---|
| `MAX` | geeft de **hoogste** waarde van een kolom |
| `SUM` | geeft het **totaal** van een kolom |
| `AVG` | geeft het **gemiddelde** van een kolom |
| `COUNT(`*`kolom`*`)` | geeft het **aantal geselecteerde items** (bevat een kolom de waarde null dan wordt die niet mee geteld) |
| `COUNT(*)` | geeft het **aantal geselecteerde rijen** |

*Voorbeelden:*
De gemiddelde prijs van de artikelen kun je opvragen met:

```
SELECT AVG(prijs)
FROM artikelen;
```

In het volgende voorbeeld wordt de gemiddelde prijs gevraagd van alle tenten:

```
SELECT AVG(prijs)
FROM artikelen
WHERE artikelnaam='Tent';
```

En op deze manier tel je het aantal artikelen:

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

### *Opdracht I6*

- Geef de **`SELECT`** opdracht waarmee je uit de tabel artikelen de hoogste prijs van de artikelen vindt.
(Het gaat hier dus om de prijs van het duurste artikel. Het gaat er niet om dat je erbij vermeldt welk artikel het betreft).

### *Opdracht I7*

- Geef de **`SELECT`** opdracht waarmee je de gemiddelde prijs krijgt van alle artikelen die in december zijn besteld.

***Opdracht 18***

- Geef de **SELECT** opdracht waarmee je het totaal aantal bestelde zaklantaarns krijgt.

***Opdracht 19***

- Geef de **SELECT** opdracht waarmee je van alle bestelde zaklantaarns de laagste prijs krijgt. Laat alleen de prijs zien!

***Opdracht 20***

- Geef de **SELECT** opdracht waarmee je het aantal bestellingen van januari krijgt.

***Opdracht 21***

- Geef de **SELECT** opdracht waarmee je het aantal klanten krijgt waarvan de achternaam met een ’G’ begint.

***Opdracht 22***

- Geef de **SELECT** opdracht die je de gemiddelde prijs van de zaklantaarns oplevert.

# 1.9 GROUP BY

De **GROUP BY** optie maakt steeds **één rij van identieke gegevens** van een kolom. **GROUP BY** is bij uitstek geschikt om te gebruiken samen met de aggregaat functies.

*Voorbeeld:*
Veronderstel dat je de artikelen tabel wilt groeperen op grond van de bestelde hoeveelheid en als je dan ook nog wilt weten wat de hoogste prijs is van zo'n groep, dan krijg je een dergelijke opdracht:

```
SELECT hoeveelheid, MAX(prijs)
FROM bestellingen, artikelen
WHERE bestellingen.artikelnummer = artikelen.artikelnummer
GROUP BY hoeveelheid;
```

***Opdracht 23***

- Voer deze opdracht in en controleer of het klopt.

***Opdracht 24***

- Geef de **SELECT** opdracht om te bepalen hoeveel klanten er in iedere provincie (uniek) wonen. Laat de provincie zien en het aantal mensen. (Tip: **COUNT** gebruik je om rijen te tellen, **SUM** werkt alleen op numerieke gegevens maar kan die totaliseren!)

***Opdracht 25***

- Geef de **SELECT** opdracht waarmee je ieder artikel en de maximum en de minimum prijs ervan krijgt.

***Opdracht 26***

- Geef de **SELECT** opdracht om het aantal bestellingen per klant en het totaal bestede bedrag te achterhalen. Doe dat door als kolommen het klantnummer, het aantal bestellingen (met **COUNT**) en het totaalbedrag van die bestellingen (met **SUM**) op te vragen.

## 1.10 HAVING

**HAVING** hoort bij **GROUP BY**. Met **HAVING** leg je de voorwaarden vast die ervoor zorgen welke rijen met **GROUP BY** geselecteerd worden.

Voorbeeld:
In opdracht 23 hebben we de bestelde artikelen gegroepeerd op grond van de bestelde hoeveelheid en daarvan de hoogste prijs laten zien. Maar als we deze gegevens alleen willen zien als de hoeveelheid groter is dan 1 dan kan dat met **HAVING** als volgt:

```
SELECT hoeveelheid, MAX(prijs)
FROM bestellingen, artikelen
WHERE bestellingen.artikelnummer = artikelen.artikelnummer
GROUP BY hoeveelheid
```

```sql
HAVING hoeveelheid > 1;
```

Denk er dus om, wanneer je **GROUP BY** gebruikt hoort daar **HAVING** bij en niet **WHERE**! Als je in dit voorbeeld had willen selecteren op prijs dan had je dat gewoon met **WHERE** moeten doen maar omdat hoeveelheid onderwerp is van het groeperen gebruik je daar **HAVING.**

***Opdracht 27***

- Als opdracht 26 maar nu gaat het alleen om de orders in januari.

***Opdracht 28***

- Geef de **SELECT** opdracht om te bepalen hoeveel klanten er in iedere provincie (uniek) wonen. Laat de provincie en het aantal klanten zien maar alleen wanneer het aantal klanten groter is dan één.

***Opdracht 29***

- Geef de **SELECT** opdracht waarmee je ieder artikel en de maximum en minimum prijs daarvan krijgt. Maar laat de rij alleen zien als de maximum prijs hoger is dan 190.00.

***Opdracht 30***

- Geef de **SELECT** opdracht om het aantal artikelen per klant en het totaal bestede bedrag te achterhalen voor die klanten die meer dan 1 artikel besteld hebben. Het gaat hier niet om de klanten die van één artikel meer dan één exemplaar bestelden maar om klanten die meerdere (verschillende) artikelen bestelden.

***Opdracht 31***

- Geef de **SELECT** opdracht waarmee je ieder artikel en de maximum en de minimum prijs daarvan krijgt. Maar laat de rij alleen zien als er totaal meer dan 3 van besteld zijn.

# 1.11 ORDER BY

Met **ORDER BY** bepaal je hoe de uitvoer **gesorteerd** wordt. Achter **ORDER BY** kunnen meerdere kolommen komen waarna eerst op de eerste kolom gesorteerd wordt en daarbinnen op de volgende kolom enz.

*Ook hier weer eerst een voorbeeld:*

```
SELECT *
FROM artikelen
ORDER BY prijs;
```

De artikelen worden nu gesorteerd op prijs, van laag naar hoog. Wil je ze van hoog naar laag zien dan zet je er **DESC** achter. Zoals:

```
SELECT *
FROM artikelen
ORDER BY prijs DESC;
```

Je hebt voor het sorteren de keuze uit **ASC** (ascending = oplopend) en **DESC** (descending = aflopend) maar **ASC** is de default waarde. Voor cijfers betekent **ASC** van laag naar hoog, voor letters betekent het van A tot z. **DESC** sorteert dus juist andersom.
Je kunt dus op meerdere kolommen sorteren. Als je eerst op hoeveelheid wilt sorteren maar daar binnenin op prijs dan doe je dat als volgt:

```
SELECT *
FROM artikelen
```

```sql
ORDER BY hoeveelheid, prijs;
```

***Opdracht 32***

- Selecteer de voornaam, de achternaam en de woonplaats van alle klanten in de klantentabel. Zorg ervoor dat de resultaten zichtbaar worden oplopend gesorteerd op basis van de achternaam.

***Opdracht 33***

- Als opdracht 32 maar nu in aflopende volgorde.

***Opdracht 34***

- Selecteer artikel en prijs uit alle artikelen in de artikelen tabel waarvan de prijs hoger is dan 10.00. Sorteer de resultaten in oplopende volgorde gebaseerd op de prijs.

***Opdracht 35***

- Met welke **`SELECT`** opdracht krijg je alle gegevens van de bestellingen tabel gesorteerd op klantnummer?

***Opdracht 36***

- Geef de **SELECT** opdracht waarmee je ieder artikel en de maximum en de minimum prijs daarvan krijgt. Maar laat de rij alleen zien als de maximum prijs groter is dan 100.00. Zorg ervoor dat de artikelen aflopend gesorteerd zijn.

# 1.12 AND en OR

Als we **WHERE** gebruiken willen we daar vaak meer dan één voorwaarde achter zetten. De opgevraagde gegevens moeten dan aan meerdere voorwaarden voldoen. Soms moeten ze aan alle voorwaarden voldoen maar soms ook aan tenminste één van de voorwaarden. Door gebruik te maken van **AND** en **OR**, twee zogenaamde logische operatoren, kunnen we dat realiseren.
Bij **AND** geldt dat de beide voorwaarden die links en rechts van de **AND** staan, waar moeten zijn, bij **OR** geldt dat één van de beide voorwaarden waar moet zijn.

*Voorbeelden:*
Veronderstel dat je een lijst wilt hebben van paraplu's vanaf 5 euro. Dat kun je opvragen met de volgende opdracht:

```
SELECT *
FROM artikelen
WHERE artikelnaam = 'Paraplu' AND prijs >= 5;
```

Om het lezen van dergelijke voorwaarden gemakkelijker te maken mag je er als volgt haakjes om heen zetten:

```
WHERE (artikelnaam = 'Paraplu') AND (prijs >= 5);
```

Haakjes zijn helemaal handig als je meerdere voorwaarden hebt en je wilt de voorrang bepalen van de afzonderlijke voorwaarden. Dit is vooral het geval wanneer je **AND** en **OR** wilt combineren.
Hieronder een voorbeeld van het gebruik van **OR**:

```
SELECT *
FROM artikelen
WHERE artikelnaam = 'Paraplu' OR artikel = 'Tent';
```

Bij deze selectie krijg je alle paraplu’s en alle tenten.

***Opdracht 37***

```
SELECT *
FROM artikelen
WHERE artikelnaam = 'Paraplu' OR prijs >= 5;
```

- Wat levert deze opdracht op?

***Opdracht 38***

- Geef de `SELECT` opdracht waarmee je het klantnummer, de order_datum en het artikel krijgt behalve van de sneeuwschoenen en van de oorwarmers.

***Opdracht 39***

- Geef de `SELECT` opdracht waarmee je alle artikelen en hun prijs krijgt waarbij de artikelnaam begint met de letters ‘S’, ‘P’ of ‘F’.

***Opdracht 40***

- Geef de `SELECT` opdracht waarmee je alle klanten selecteert die uit Friesland of uit Gelderland komen.

***Opdracht 41***

- Geef de **`SELECT`** opdracht waarmee je alle klanten selecteert die niet uit Friesland of uit Gelderland komen.

Opdracht 41 kun je oplossen zoals ook opdracht 40 werd opgelost. Maar omdat het precies het tegengestelde is van opdracht 40 kun je het hier ook anders oplossen, namelijk door voor de voorwaarden het woordje **`NOT`** te plaatsen.

Dus als volgt: `WHERE NOT(...voorwaarden...);` of:

```
WHERE NOT((...voorwaarde1...) AND|OR (...voorwaarde2...));
```

## 1.13 IN en BETWEEN

Met **`IN`** wordt bekeken of het veld voor het woord **`IN`** ook voorkomt in de lijst na het woord **`IN`**.

*Voorbeeld:*
Zoeken we de klanten in Almelo, Utrecht en Sneek dan kunnen we dat als volgt doen:

```
SELECT *
FROM klanten
WHERE woonplaats IN ('Almelo', 'Utrecht', 'Sneek');
```

Je had dit ook op een andere manier op kunnen lossen namelijk met **`OR`**. Onderstaande **`SELECT`** opdracht levert hetzelfde resultaat:

```
SELECT *
FROM klanten
WHERE woonplaats = 'Almelo' OR woonplaats = 'Utrecht' OR
woonplaats = 'Sneek';
```

Je ziet dat door het gebruik van **`IN`** de opdracht een stukje korter is.

Je kunt in plaats van **IN** ook **NOT IN** gebruiken. In het voorbeeld hieronder worden dan alle klanten geselecteerd die juist niet in één van die plaatsen wonen.

```
SELECT *
FROM klanten
WHERE woonplaats NOT IN ('Almelo', 'Utrecht', 'Sneek');
```

Met **BETWEEN** bekijk je of het veld waar je op zoekt tussen twee waarden zit.

*Voorbeeld:*
Zoek je de bestelde artikelen die tussen de 10 en 40 euro kostten dan kun je dat als volgt doen:

```
SELECT *
FROM artikelen
WHERE prijs BETWEEN 10 AND 40;
```

En met:

```
SELECT *
FROM klanten
WHERE woonplaats BETWEEN 'Almelo' AND 'Delfstrahuizen';
```

krijg je de gegevens van de klanten die in de woonplaatsen Almelo of Delfstrahuizen wonen of die in woonplaatsen wonen die daar alfabetisch tussenin liggen.

Net als bij **IN** kun je een opdracht met **BETWEEN** ook anders schrijven:

```
SELECT *
FROM artikelen
WHERE prijs >= 10 AND prijs <= 40;
```

Zoals te verwachten viel kun je ook voor **BETWEEN** het woord **NOT** gebruiken als je juist het tegenovergestelde wilt selecteren.

```
SELECT *
FROM bestelde_artikelen
WHERE prijs NOT BETWEEN 10 AND 40;
```

***Opdracht 42***

- Selecteer de orderdatum, het artikel en de prijs van de artikelen die besteld zijn voor ieder artikel met een prijs tussen 10.00 en 80.00.

***Opdracht 43***

- Selecteer de voornaam, de woonplaats en de provincie van de klanten uit de provincies Friesland, Flevoland, Noord-Holland, Zuid-Holland en Utrecht.

***Opdracht 44***

- Selecteer alle velden van de artikelen waarvan de bestelde hoeveelheid ligt tussen 1 en 3. Doe dit tweemaal, zowel met **`IN`** als met **`BETWEEN`**.

***Opdracht 45***

- Als opdracht 44 alleen nu het tegenovergestelde dus met **`NOT`**.

# 1.14 Wiskundige berekeningen

Binnen SQL kun je ook een aantal wiskundige berekeningen opnemen in jouw opdracht zoals:

| Operator | Betekenis |
|---|---|
| `+` | optellen |
| `-` | aftrekken |
| `*` | vermenigvuldigen |
| `/` | delen |
| `%` | modulus |

Maar er zijn nog meer berekeningen die je binnen SQL kunt doen. Hieronder de tabel met de standaard functies:

| Functie | Betekenis |
|---|---|
| **`ABS(x)`** | geeft de absolute waarde van x |
| **`SIGN(x)`** | geeft het teken van x in de vorm van -1 voor een negatief getal, 0 voor 0 en 1 voor een positief getal |
| **`MOD(x,y)`** | geeft de rest bij deling van x door y. Dit is hetzelfde als x%y |
| **`FLOOR(x)`** | geeft het grootste gehele getal dat kleiner is of gelijk is aan x **(werkt niet in sqlite)** |
| **`CEILING(x)` of `CEIL(x)`** | geeft het kleinste gehele getal dat groter is of gelijk is aan x **(werkt niet in sqlite)** |
| **`POWER(x,y)`** | geeft de waarde van x tot de macht y |
| **`ROUND(x)`** | rondt x af op het dichtstbijzijnde gehele getal |
| **`ROUND(x,d)`** | rondt x af op een getal met d decimalen |
| **`SQRT(x)`** | geeft de wortel van x |

Vergelijk het resultaat maar eens van de volgende drie **SELECT** opdrachten:

```
SELECT AVG(prijs)
FROM artikelen;
```

```sql
SELECT ROUND(AVG(prijs))
FROM artikelen;

SELECT ROUND(AVG(prijs),2)
FROM artikelen;
```

### *Opdracht 46*

- Geef de artikelnaam en de totaalprijs voor ieder artikel in de bestellingen tabel. (Hint: vermenigvuldig hiervoor de prijs met de hoeveelheid.)

### *Opdracht 47*

- Geef de `SELECT` opdracht om de afgeronde prijs (op 0 decimalen) van de sneeuwschuivers te bepalen.

### *Opdracht 48*

- Geef de `SELECT` opdracht om de afgeronde som van de prijzen van de eenwielers te bepalen

### *Opdracht 49*

- Geef de `SELECT` opdracht om te bepalen wat het kleinste bedrag (prijs) is uit de artikelentabel dat gelijk is aan of net boven 100 komt.

# 1.15 Subselect

Soms is het lastig een query op te bouwen. Zeker als het om een ingewikkelde gaat. Veronderstel dat je wilt weten welke artikel de laagste prijs heeft.
Dat is nog best lastig.

De query:

```
SELECT artikelnummer, artikelnaam, prijs
FROM artikelen
WHERE prijs = min(prijs);
```

geeft een foutmelding.

```
SELECT artikelnummer, artikelnaam, min(prijs)
FROM artikelen;
```

geeft geen foutmelding maar meldt als artikelnaam de sneeuwschuiver en dat klopt niet.

```
SELECT min(prijs)
FROM artikelen;
```

geeft wel de laagste prijs maar daarmee hebben we nog niet het artikelnummer.

Het wordt tijd voor een **subselect**, oftewel een `SELECT` binnen een `SELECT`.
Kijk maar eens naar de volgende oplossing:

```
SELECT artikelnummer, artikelnaam, prijs
FROM artikelen
WHERE prijs IN (
  SELECT min(prijs)
  FROM artikelen
);
```

Hier wordt twee keer `SELECT` gebruikt. De tweede `SELECT` levert de minimum prijs op. Door daarna in de eerste `SELECT` achter de `WHERE` alleen die artikelen te willen zien waarvan de prijs in de tweede `SELECT` voorkomt krijg je de juiste artikelen. In dit geval zijn dat er twee, namelijk een paraplu en een zaklantaarn.

Het had wel wat simpeler gekund maar dan hadden we niet hetzelfde resultaat gehad.

De volgende query levert ook een artikel met de laagste prijs:

```
SELECT artikelnummer, artikelnaam, prijs
FROM artikelen
ORDER BY prijs
LIMIT 1;
```

Door op prijs te sorteren krijg je de laagste prijs als eerste te zien. Door nu met `LIMIT 1` de uitvoer te beperken tot 1 record krijg je inderdaad een artikel met de laagste prijs. Het probleem is alleen dat er hier twee artikelen zijn met diezelfde prijs en ik krijg er zo maar 1.
Kortom de subselect variant is wel beter en in ieder geval vollediger.

Maar wat als we de vraag eens uitbreiden. Dat ik de klant wil weten die het artikel met de laagste prijs besteld heeft.
Pas op, je weet hierbij niet zeker of die artikelen met de laagste prijs wel in de bestellingen voorkomt.
Om daar zeker van te zijn wordt eerst nog een artikel aangemaakt met een nog lagere prijs, namelijk € 2,-. Dat doe ik met de regel:

```
INSERT INTO "artikelen" VALUES(36,'Fietsbel',2.0);
```

Ik weet  zeker dat dit artikel niet in de bestellingen voorkomt maar wel dat het het goedkoopste artikel is.
Met de volgende query onderzoek ik eerst welke artikelen in de bestellingen voorkomen en daaruit haal ik dan de artikelen met de laagste prijs:

```
SELECT artikelnummer, artikelnaam, prijs
FROM artikelen
WHERE prijs IN (
  SELECT min(prijs)
  FROM artikelen, bestellingen
  WHERE artikelen.artikelnummer = bestellingen.artikelnummer
);
```

En inderdaad geeft deze query nog steeds een paraplu en een zaklantaarn. Precies wat we wilden.

Nu nog onderzoeken welke klanten deze artikelen besteld hebben. Maar daarvoor hebben we het bestellingnummer nodig die bij die minimum prijs hoort.
Dat is lastig want met zo'n subselect zoals we dat hier gebruikt hebben moet de uitkomst van de **SELECT** altijd gelijk zijn aan waar het mee vergeleken wordt. Dus bij **WHERE prijs IN (...)** moet die **SELECT** die daar achteraan komt dus altijd een prijs opleveren. Je kunt daar op deze manier niet ook nog een bestellingnummer uithalen.

Maar hier is de oplossing:

```
SELECT voornaam, achternaam
FROM klanten, bestellingen, artikelen
WHERE klanten.klantnummer = bestellingen.klantnummer
AND bestellingen.artikelnummer = artikelen.artikelnummer
AND artikelen.artikelnummer IN (
  SELECT artikelnummer
  FROM artikelen
  WHERE prijs IN (
    SELECT min(prijs)
    FROM artikelen, bestellingen
    WHERE artikelen.artikelnummer=bestellingen.artikelnummer
  )
);
```

Je ziet hier dus twee keer een **SELECT** binnen een **SELECT**. De binnenste levert de prijs op, de middelste levert de artikelnummers op die bij die prijs horen en de buitenste levert de voornaam en de achternaam van de klanten die die artikelnummers besteld hebben.

Nog een laatste voorbeeld van een subselect.
Iedere select levert een nieuwe (tijdelijke) tabel op. Zoiets wordt ook wel een **cursor** genoemd. Dat is de afkorting van **Cur**rent **S**et **O**f **R**ecords.
En een tabel kun je achter **FROM** zetten dus ook een **SELECT** kan achter **FROM**.

Wat denk je dat deze query oplevert?

```
SELECT voornaam, achternaam, hoeveelheid
FROM klanten, (
  SELECT hoeveelheid, klantnummer AS klant
  FROM bestellingen
  WHERE hoeveelheid > 1
)
WHERE klanten.klantnummer = klant;
```

In de binnenste **SELECT** zie je iets bijzonders. Daar wordt gevraagd om de hoeveelheid en het klantnummer uit de tabel bestellingen. Maar aangezien klantnummer ook in de tabel klanten voorkomt wordt er hier gemakshalve voor gekozen om klantnummer uit de tabel bestellingen als het ware te vervangen door klant.
Dus eigenlijk staat er in de laatste regel:

```
WHERE klanten.klantnummer = bestellingen.klantnummer;
```

maar de tabel bestellingen komt alleen maar in de binnenste query voor.

Het resultaat van deze query zal zijn dat ik de voornaam, de achternaam en de hoeveelheid krijg van de klanten die meer dan 1 stuks besteld hebben.

### *Opdracht 50*

- Geef de **SELECT** opdracht waarmee je alle gegevens van de klanten krijgt die orders geplaatst hebben met een totaalprijs hoger dan 100 (let op, het getal in de kolom prijs is de stuksprijs).

### *Opdracht 51*

Naast deze voorbeelden is het zelfs mogelijk een subselect op de eerste regel van een query te zetten. Bijvoorbeeld:

```
SELECT klantnummer, (SELECT .........
```

- Bedenk een voorbeeld waarbij je dat zou kunnen gebruiken.

### *Opdracht 52*

In plaats van **`IN`** waarbij je zoekt of een waarde in een subselect voorkomt kun je ook gebruik maken van **`ANY`** of **`ALL`**.

- Onderzoek wat deze woorden precies doen.
- Bedenk een voorbeeld waarbij je dat zou kunnen gebruiken.

Tot zover SQL. Hoewel er nog veel meer mogelijk is met SQL is dit voor ons doel voorlopig wel voldoende.