ATHENEUM BOOM

5INF – 2015 - 2016

ABSTRACT
SQL (Structured Query Language) is een ANSI/ISO-standaardtaal voor een relationeel databasemanagementsysteem (DBMS).

Ivo Goyens

SQL

Inhoudstafel

1 SQL Opdrachten

1.1 SELECT

Waarvoor gebruiken we deze SQL-opdrachten? Deze worden algemeen gebruikt om gegevens te selecteren in de tabellen binnen een database. U ziet onmiddellijk twee sleutelwoorden: u moet informatie **SELECT** (selecteren) **FROM** (van) een tabel. (Bemerk dat een tabel een houder is binnen de database waar de gegevens worden opgeslagen. Zo hebben we hier de absolute basis SQL-structuur:
SELECT "kolom_naam" FROM "tabel_naam";

Veronderstel als voorbeeld dat u de volgende tabel hebt:

Tabel Store_Information

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999
San Diego	250	07-Jan-1999
Los Angeles	300	08-Jan-1999
Boston	700	08-Jan-1999

Deze tabel wordt verder als voorbeeld gebruikt in de zelfstudie (de tabel komt voor in alle delen). Om de winkels te selecteren in de tabel, voert u in,

SELECT Store_Name FROM Store_Information;

Resultaat:

Store_Name
Los Angeles
San Diego
Los Angeles
Boston

U kunt meerdere kolomnamen selecteren, evenals meerdere tabelnamen.

1.2 DISTINCT

Het sleutelwoord **SELECT** laat toe alle informatie van een kolom (of kolommen) in een tabel te nemen. Dit wil noodzakelijkerwijze zeggen dat er redundanties zullen zijn. Wat doen we als we elk **DISTINCT**(afzonderlijk) element willen selecteren? Dit is makkelijk in SQL. We moeten enkel **DISTINCT** toevoegen achter **SELECT**. De syntaxis is als volgt:
SELECT DISTINCT "kolom_naam"
FROM "tabel_naam";

Als u bijvoorbeeld alle afzonderlijke winkels in tabel ***Store_Information*** wilt selecteren,

Tabel Store_Information

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999
San Diego	250	07-Jan-1999
Los Angeles	300	08-Jan-1999
Boston	700	08-Jan-1999

voert u in

```
SELECT DISTINCT Store_Name FROM Store_Information;
```

Resultaat:

Store_Name
Los Angeles
San Diego
Boston

1.3 WHERE

Vervolgens kunt u voorwaardelijk gegevens selecteren in een tabel. Wanneer u bijvoorbeeld enkel winkels wenst op te halen met een verkoop hoger dan 1 000 euro, gebruikt u het sleutelwoord **WHERE**. De syntaxis is als volgt:

```
SELECT "kolom_naam"
FROM "tabel_naam"
WHERE "voorwaarde";
```

Als u bijvoorbeeld alle afzonderlijke winkels in tabel ***Store_Information*** wil selecteren,

Tabel ***Store_Information***

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999
San Diego	250	07-Jan-1999
Los Angeles	300	08-Jan-1999
Boston	700	08-Jan-1999

voert u in

```
SELECT Store_Name
FROM Store_Information
WHERE Sales > 1000;
```

Resultaat:

Store_Name
Los Angeles

1.4 AND & OR

In het vorige deel hebt u gezien dat u het sleutelwoord **WHERE** kunt gebruiken om voorwaardelijk gegevens in een tabel te selecteren. Deze voorwaarde kan een enkele voorwaarde zijn (zoals deze in het vorige deel), of het kan een samengestelde voorwaarde zijn. Samengestelde voorwaarden bestaan uit meerdere enkele voorwaarden verbonden door **AND** of **OR**. Er is geen beperking op het aantal enkele voorwaarden die aanwezig kunnen zijn in één enkele SQL-instructie.

De syntaxis voor een samengestelde voorwaarde is als volgt:

```
SELECT "kolom_naam"
FROM "tabel_naam"
WHERE "enkele voorwaarde"
{[AND|OR] "enkele voorwaarde"}+;
```

{}+ betekent dat de expressie binnen de accolades één of meerdere malen zal voorkomen. Merk op dat **AND** en **OR** door elkaar kunnen worden gebruikt. Daarnaast kunt u de ronde haakjes ( ) gebruiken om de rangorde van de voorwaarde aan te duiden.

Wanneer u bijvoorbeeld alle winkels met een omzet van meer dan1 000 € wil selecteren of alle winkels met een omzet van minder dan 500 € maar meer dan 275 € in de tabel ***Store_Information***,

Tabel Store_Information

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999
San Diego	250	07-Jan-1999
San Francisco	300	08-Jan-1999
Boston	700	08-Jan-1999

voert u in

```
SELECT Store_Name
FROM Store_Information
WHERE Sales > 1000
OR (Sales < 500 AND Sales > 275);
```

Resultaat:

Store_Name
Los Angeles
San Francisco

1.5 IN

In SQL kan het sleutelwoord **IN** op twee manieren worden gebruikt, dit deel behandelt deze die verbonden is met de **WHERE** component. Binnen deze context kent u precies de waarde van de geretourneerde waarden die u wil zien voor minstens één kolom. de syntaxis voor het sleutelwoord **IN** is als volgt:

```
SELECT "kolom_naam"
FROM "tabel_naam"
WHERE "kolom_naam" IN ('waarde1', 'waarde2', ...);
```

Er kunnen meer waarden tussen de aanhalingstekens staan, waarbij elke waarde wordt gescheiden met een komma. De waarden kunnen numeriek of tekens zijn. Als er slechts één waarde binnen de aanhalingstekens staat, is deze opdracht gelijk aan

```
WHERE "kolom_naam" = 'waarde1'
```

U wenst bijvoorbeeld alle gegevens voor de winkels van Los Angeles en San Diego te selecteren in tabel ***Store_Information***,

Tabel Store_Information

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999
San Diego	250	07-Jan-1999
San Francisco	300	08-Jan-1999
Boston	700	08-Jan-1999

voert u in

```
SELECT *
FROM Store_Information
WHERE store_name IN ('Los Angeles', 'San Diego');
```

Resultaat:

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999
San Diego	250	07-Jan-1999

1.6 BETWEEN

Terwijl het sleutelwoord **IN** helpt om de selectiecriteria te beperken tot één of meer afzonderlijke waarden, kunt u met het sleutelwoord **BETWEEN** een bereik selecteren. De syntaxis voor de component **BETWEEN** is als volgt:

```
SELECT "kolom_naam"
FROM "tabel_naam"
WHERE "kolom_naam" BETWEEN 'waarde1' AND 'waarde2';
```

Hiermee kiest u alle rijen waarvan de kolom een waarde heeft tussen 'waarde1' en 'waarde2'.

Als u bijvoorbeeld alle omzetinformatie tussen 6 januari 1999 en 10 januari 1999 wil selecteren in de tabel ***Store_Information***,

Tabel Store_Information

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999
San Diego	250	07-Jan-1999
San Francisco	300	08-Jan-1999
Boston	700	08-Jan-1999

voert u in

```
SELECT *
FROM Store_Information
WHERE Txn_Date BETWEEN '06-Jan-1999' AND '10-Jan-1999';
```

Merk op dat gegevens in verschillende formaten kunnen zijn opgeslagen in verschillende databases. Deze zelfstudie kiest één van deze formaten.

Resultaat:

Store_Name	Sales	Txn_Date
San Diego	250	07-Jan-1999
San Francisco	300	08-Jan-1999
Boston		

1.7 LIKE

OPGELET!!! % werkt niet bij Access. Gebruik * als wildcard!

LIKE is nog een sleutelwoord dat wordt gebruikt in de **WHERE** component. Met **LIKE** kunt u zoeken op basis van een patroon in plaats van precies te specificeren wat u wenst (zoals het geval is voor **IN**) of kunt u een bereik bepalen (zoals het geval is voor **BETWEEN**). De syntaxis is als volgt:

```
SELECT "kolom_naam"
FROM "tabel_naam"
WHERE "kolom_naam" LIKE {PATROON};
```

{PATROON} bestaat dikwijls uit jokerregels. Ziehier een aantal voorbeelden:

- 'A_Z': Alle tekenreeksen die beginnen met 'A', een ander letterteken, en eindigen op 'Z'. 'ABZ' en 'A2Z' voldoen beiden aan deze voorwaarde, terwijl 'AKKZ' niet voldoet (omdat er twee lettertekens tussen A en Z staan in plaats van één).

- 'ABC%': Alle tekenreeksen die beginnen met 'ABC'. 'ABCD' en 'ABCABC' bijvoorbeeld voldoen beiden aan de voorwaarde.

- '%XYZ': Alle tekenreeksen die beginnen met 'XYZ'. 'WXYZ' en 'ZZXYZ' bijvoorbeeld voldoen beiden aan de voorwaarde.

- '%AN%': Alle tekenreeksen die op eender welke plaats het patroon 'AN' bevatten. 'LOS ANGELES' en 'SAN FRANCISCO' bijvoorbeeld voldoen beiden aan deze voorwaarde.

Neem de volgende tabel als voorbeeld:

Tabel Store_Information

Store_Name	Sales	Txn_Date
LOS ANGELES	1500	05-Jan-1999
SAN DIEGO	250	07-Jan-1999
SAN FRANCISCO	300	08-Jan-1999
BOSTON	700	08-Jan-1999

U wil alle winkels vinden waarvan de naam 'AN' bevat. Hiertoe voert u in,

```
SELECT *
FROM Store_Information
WHERE store_name LIKE '%AN%';
```

Resultaat:

Store_Name	Sales	Txn_Date
LOS ANGELES	1500	05-Jan-1999
SAN DIEGO	250	07-Jan-1999
SAN FRANCISCO	300	08-Jan-1999

1.8 ORDER BY

Dusver hebt u gezien hoe gegevens uit een tabel worden gehaald met de opdrachten **SELECT** en **WHERE**. Dikwijls moet de uitvoer echter in een bepaalde volgorde worden gegeven. Dit kan in oplopende of aflopende volgorde zijn, of op basis van een numerieke of tekstwaarde. In dergelijke gevallen kunt u het sleutelwoord **ORDER BY** gebruiken om uw doel te bereiken.

De syntaxis voor aan **ORDER BY** instructie is als volgt:

```
SELECT "kolom_naam"
FROM "tabel_naam"
[WHERE "voorwaarde"]
ORDER BY "kolom_naam" [ASC, DESC];
```

De [] betekenen dat de **WHERE** instructie optioneel is. Als er echter een component **WHERE** bestaat, komt deze voor de component **ORDER BY**. **ASC**betekent dat de resultaten worden weergegeven in oplopende volgorde, terwijl **DESC** betekent dat ze worden weergegeven in aflopende volgorde. Als geen van beide wordt gespecificeerd, is de standaard **ASC**.

U kunt over meerdere kolommen rangschikken. In dat geval wordt de component **ORDER BY** hierboven

```
ORDER BY "kolom_naam1" [ASC, DESC], "kolom_naam2" [ASC, DESC]
```

Als u bijvoorbeeld oplopende volgorde kiest voor beide kolommen, wordt de uitvoer gerangschikt in oplopende volgorde volgens kolom 1. Als er een verband is voor de waarde van kolom 1, wordt de uitvoer oplopend grangschikt volgens kolom 2.

Als u bijvoorbeeld de inhoud van de tabel ***Store_Information*** in aflopende volgorde wil rangschikken volgens het dollarbedrag:

Tabel Store_Information

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999
San Diego	250	07-Jan-1999
San Francisco	300	08-Jan-1999
Boston	700	08-Jan-1999

voert u in

```
SELECT Store_Name, Sales, Txn_Date
FROM Store_Information
ORDER BY Sales DESC;
```

Resultaat:

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999
Boston	700	08-Jan-1999
San Francisco	300	08-Jan-1999
San Diego	250	07-Jan-1999

Naast de kolomnaam kunt u ook de kolompositie gebruiken (gebaseerd op de SQL-query) om aan te duiden voor welke kolom de component **ORDER BY**van toepassing is. De eerste kolom is 1, de tweede kolom is 2, enzovoort. In het bovenstaand voorbeeld krijgt u hetzelfde resultaat met de volgende opdracht:

```
SELECT Store_Name, Sales, Txn_Date
FROM Store_Information
ORDER BY 2 DESC;
```

1.9 FUNCTIES

U werkt met nummers, dus kunt u zich afvragen of u wiskundige bewerkingen kunt uitvoeren ermee, zoals optellen of een gemiddelde berekenen. Dit is mogelijk! SQL beschikt over de volgende rekenkundige functies:

- **AVG**
- **COUNT**
- **MAX**
- **MIN**
- **SUM**

De syntaxis voor het gebruik van functies is,

```
SELECT "functietype"("kolom_naam")
FROM "tabel_naam";
```

U wenst bijvoorbeeld het totaal van alle omzetten te berekenen in de volgende tabel,

Tabel Store_Information

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999
San Diego	250	07-Jan-1999
Los Angeles	300	08-Jan-1999
Boston	700	08-Jan-1999

dan voert u in

```
SELECT SUM(Sales) FROM Store_Information;
```

Resultaat:

SUM(Sales)
2750

2750 is de som van alle omzetinvoeren: 1500 + 250 + 300 + 700.

Naast deze functies kunt u met SQL ook eenvoudige taken uitvoeren zoals optellen (+) en aftrekken (-). Voor tekengegevens zijn er verschillende tekenreeksen beschikbaar, zoals samenvoegen, knippen en subtekenreeksfuncties. De verschillende RDBMS-leveranciers hebben verschillende tekenreeksfunctie-implementaties, raadpleeg de referenties voor uw RDBMS om te zien hoe deze functies worden gebruikt.

1.10 COUNT

COUNT is een andere rekenkundige functie. Hiermee kunt u het aantal rijen in een bepaalde tabel optellen **COUNT**. De syntaxis is,

```
SELECT COUNT("kolom_naam")
FROM "tabel_naam";
```

Als u bijvoorbeeld het aantal winkelinvoeren wil zien in de tabel,

Tabel Store_Information

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999
San Diego	250	07-Jan-1999
Los Angeles	300	08-Jan-1999
Boston	700	08-Jan-1999

voert u in

```
SELECT COUNT (Store_Name)
FROM Store_Information;
```

Resultaat:

COUNT (Store_Name)
4

COUNT en **DISTINCT** kunnen samen worden gebruikt in een instructie om het aantal afzonderlijke invoeren in een tabel te zien.
Als u bijvoorbeeld het aantal afzonderlijke winkels wenst te kennen, voert u in,

SELECT COUNT (DISTINCT Store_Name) → **OPGELET!! Werkt niet in Access. Zie tekstbox!**
FROM Store_Information;

Resultaat:

COUNT (DISTINCT Store_Name)
3

Code in ACCESS!

```
SELECT COUNT (Store_Name)
FROM
(SELECT DISTINCT Store_Name
FROM Store_Information);
```

1.11 GROUP BY

Nu komt u terug bij statistische functies. U herinnert zich dat het sleutelwoord **SUM** werd gebruikt om de totale omzet van alle winkels te berekenen? Wat doet u als u de totale omzet van *elke* winkel wenst te berekenen? Daar moet u twee zaken bij in acht nemen: Ten eerste moet u maken dat zowel de winkelnaam als de totale omzet wordt geselecteerd. Ten tweede moet u maken dat alle omzetcijfers gegroepeerd zijn per winkel **grouped by**. De betreffende SQL-syntaxis is,

```
SELECT "kolom_naam1", SUM("kolom_naam2")
FROM "tabel_naam"
GROUP BY "kolom_naam1";
```

U kunt de volgende tabel als voorbeeld nemen,

Tabel Store_Information

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999
San Diego	250	07-Jan-1999
Los Angeles	300	08-Jan-1999
Boston	700	08-Jan-1999

U wilt de totale omzet van elke winkel berekenen. Hiertoe voert u in,

```
SELECT Store_Name, SUM(Sales)
FROM Store_Information
GROUP BY Store_Name;
```

Resultaat:

Store_Name	SUM(Sales)
Los Angeles	1800
San Diego	250
Boston	700

Het sleutelwoord **GROUP BY** wordt gebruikt om meerdere kolommen uit een tabel (of meerdere tabellen) te selecteren en er is minstens één rekenkundige operator in de instructie **SELECT**. Wanneer dit voorvalt, moet u de andere geselecteerde kolommen groeperen (**GROUP BY**), *d.w.z*, alle kolommen behalve deze die word(en)t bewerkt door de rekenkundige operator.

1.12 HAVING

Het is ook mogelijk dat u de uitvoer wenst te beperken op basis van de overeenstemmende som (of een andere statistische functie). U wilt bijvoorbeeld enkel de winkels met een omzet hoger dan 1 500 € zien. In plaats van de component **WHERE** moet in de SQL-instructie de component **HAVING** worden gebruikt, deze is voorbehouden voor statistische functies. De component **HAVING** wordt altijd dicht bij het einde van de SQL-instructie geplaatst, en een SQL-instructie met de component **HAVING** kan al dan niet de component **GROUP BY** omvatten. De syntaxis voor **HAVING** is,

```
SELECT "kolom_naam1", SUM("kolom_naam2")
FROM "tabel_naam"
GROUP BY "kolom_naam1"
HAVING (rekenkundige functievoorwaarde);
```

Opmerking: de component **GROUP BY** is optioneel.

In de voorbeeldtabel ***Store_Information***,

Tabel Store_Information

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999
San Diego	250	07-Jan-1999
Los Angeles	300	08-Jan-1999
Boston	700	08-Jan-1999

voert u in,

```
SELECT Store_Name, SUM(Sales)
FROM Store_Information
GROUP BY Store_Name
HAVING SUM(Sales) > 1500;
```

Resultaat:

Store_Name	SUM(Sales)
Los Angeles	1800

1.13 ALIAS

Nu richt u zich op het gebruiken van aliassen. Er bestaan twee soorten aliassen die dikwijls worden gebruikt: de kolomalias en de tabelalias.

Kolomaliassen helpen kort gezegd om uitvoer te organiseren. In het vorige voorbeeld, wanneer u de totale omzet ziet, staat dit vermeld als SUM(Sales). Ook als is dit begrijpelijk, toch kunnen er gevallen zijn met een gecompliceerde kolomkoptekst (vooral wanneer er verschillende rekenkundige bewerkingen bij betrokken zijn). Het gebruik van een kolomalias verbetert de leesbaarheid van de uitvoer.

De tweede soort alias is de tabelalias. U verkrijgt dit door een alias direct na de tabelnaam te plaatsen in de component **FROM**. Dit is praktisch wanneer u informatie wenst uit twee afzonderlijke tabellen (de technische term hiervoor is 'joins uitvoeren').

Het voordeel van een tabelalias bij het uitvoeren van joins zal duidelijk worden bij de bespreking van joins.

Voor u echter over joins leert, kunt u eerst de syntaxis voor de kolom- en tabelalias bekijken:

```
SELECT "tabel_alias"."kolom_naam1" "kolom_alias"
FROM "tabel_naam" "tabel_alias";
```

→ OPGELET!! Kolom_alias werkt niet in Access. Zie tekstbox!

Beide soorten aliassen worden direct na het te aliassen item geplaatst, gescheiden door een spatie. U gebruikt terug de tabel ***Store_Information***,

Tabel Store_Information

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999
San Diego	250	07-Jan-1999
Los Angeles	300	08-Jan-1999
Boston	700	08-Jan-1999

Hier gebruikt u hetzelfde voorbeeld als in het deel GROUP BY, behalve voor de toevoeging van de kolom- en tabelalias:

```
SELECT A1.Store_Name Store, SUM(A1.Sales) "Total Sales"
FROM Store_Information A1
GROUP BY A1.Store_Name;
```

Resultaat:

Store	Total Sales
Los Angeles	1800
San Diego	250
Boston	700

Code in ACCESS!

```
SELECT A1.Store_Name Store, SUM(A1.Sales) AS "Total Sales"
FROM Store_Information A1
GROUP BY A1.Store_Name;
Of
SELECT A1.Store_Name Store, SUM(A1.Sales) AS Total_Sales
FROM Store_Information A1
GROUP BY A1.Store_Name;
```

Bemerk het verschil in het resultaat: de kolomtitels zijn anders. Dit komt door het gebruik van de kolomalias. U ziet dat er in plaats van het ietwat cryptische "Sum(Sales)", nu "Total Sales" staat, hetgeen meer begrijpelijk is als kolomkoptekst.

1.14 JOIN

Nu gaat u leren over joins. Het correct maken van joins in SQL vereist veel van de elementen die tot nog toe werden behandeld. veronderstel dat u de volgende twee tabellen hebt,

Tabel ***Store_Information***

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999
San Diego	250	07-Jan-1999
Los Angeles	300	08-Jan-1999
Boston	700	08-Jan-1999

Tabel ***Geography***

Region_Name	Store_Name
East	Boston
East	New York
West	Los Angeles
West	San Diego

en u wilt de omzet per regio. U merkt dat de tabel ***Geography*** informatie omvat over regio's en winkels, en tabel ***Store_Information*** informatie over de omzet voor elke winkel. Om de omzetinfo per regio te krijgen, moet de informatie van beide tabellen worden gecombineerd. Wanneer u beide tabellen bekijkt ziet u dat ze verbonden zijn door het gemeenschappelijk veld "store-name". U bekijkt nu eerst de SQL-instructie en later krijgt u de uitleg over het gebruik van elk segment:

```
SELECT A1.Region_Name REGION, SUM(A2.Sales) SALES
FROM Geography A1, Store_Information A2
WHERE A1.Store_Name = A2.Store_Name
GROUP BY A1.Region_Name;
```

Resultaat:

REGION	SALES
East	700
West	2050

De eerste twee lijnen laten SQL twee velden kiezen, het eerste is het veld "region_name" in de tabel ***Geography*** (gealiassed als REGION), en het tweede is de som van het veld "Sales" in de tabel***Store_Information*** (gealiassed als SALES). Merk op hoe de tabelaliassen hier worden gebruikt: Geography is gealiassed als A1, en Store_Information is gealiassed als A2. Zonder alias, zou de eerste lijn als volgt zijn

Opgelet, dit is anders in Access:

Als we in het voorbeeld hiernaast alle chauffeurs willen zien, ook deze die geen ritten gereden hebben, dat werken we met LEFT JOIN (omdat we hier alles willen zien uit de linkse tabel (Chauffeurs A3). Hetzelfde geldt voor een RIGHT JOIN.

Code in ACCESS!

```
SELECT A2.klantnaam, A3.[naam WN], A1.vertrekpunt, A1.aankomst, A1.prijs
FROM Chauffeurs A3 INNER JOIN (Ritten A1 INNER JOIN Klanten A2
ON A1.klantnummer = A2.klantnummer) ON A3.[nr WN] = A1.[nr WN];
```

Code in ACCESS met LEFT JOIN!

```
SELECT A2.klantnaam, A3.[naam WN], A1.vertrekpunt, A1.aankomst, A1.prijs
FROM Chauffeurs A3 LEFT JOIN (Ritten A1 LEFT JOIN Klanten A2
ON A1.klantnummer = A2.klantnummer) ON A3.[nr WN] = A1.[nr WN];
```

1.15 INSERT

In de vorige delen werd behandeld hoe u informatie uit tabellen kunt halen. Maar hoe komen de rijen met gegevens in de tabel? Dit wordt behandeld in het huidige deel over de instructie **INSERT** en in het volgende deel over de instructie **UPDATE**.

In SQL bestaan er in feite twee manieren om gegevens toe te voegen **INSERT** aan een tabel: Bij de eerste voegt u rij per rij toe, bij de tweede voegt u meerdere rijen tegelijk toe. Hoe voegt u rij per rij gegevens toe **INSERT**:

De syntaxis om één rij toe te voegen aan een tabel is als volgt:

```
INSERT INTO "tabel_naam" ("kolom 1", "kolom 2", ...)
VALUES ("waarde 1", "waarde 2", ...);
```

Veronderstel dat u een tabel hebt met de volgende structuur:

Tabel ***Store_Information***

Kolom Naam	Gegevens Type
Store_Name	char(50)
Sales	float
Txn_Date	datetime

u wenst nu een extra rij toe te voegen aan de tabel met de omzetgegevens voor Los Angeles op 10 januari 1999. Die dag maakte de winkel 900 € omzet. Dit kan met het volgende SQL-script:

```
INSERT INTO Store_Information (store_name, Sales, Txn_Date)
VALUES ('Los Angeles', 900, '10-Jan-1999');
```

1.16 UPDATE

Zodra er gegevens aanwezig zijn in een tabel, kan het nodig zijn deze te wijzigen. Hiertoe kunt u de opdracht **UPDATE** gebruiken. De syntaxis is als volgt

```
UPDATE "tabel_naam"
SET "kolom 1" = [nieuwe waarde]
WHERE "voorwaarde";
```

Als u bijvoorbeeld een tabel hebt zoals hieronder:

Tabel ***Store_Information***

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999
San Diego	250	07-Jan-1999
Los Angeles	300	08-Jan-1999
Boston	700	08-Jan-1999

en u merkt dat de omzet voor Los Angeles op 01/08/1999 in feite 500 € is in plaats van 300 €, en de invoer dus moet worden geüpdatet. Dit kan a.d.h.v. de volgende SQL:

```
UPDATE Store_Information
SET Sales = 500
WHERE Store_Name = 'Los Angeles'
AND Txn_Date = '08-Jan-1999'
```

De tabel ziet er dan als volgt uit

Tabel ***Store_Information***

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999
San Diego	250	07-Jan-1999
Los Angeles	500	08-Jan-1999
Boston	700	08-Jan-1999

In dit geval is er slechts één rij die voldoet aan de voorwaarde in de component **WHERE**. Indien er meerdere rijen zijn die aan de voorwaarde voldoen, worden ze allen aangepast.

U kunt ook meerdere kolommen samen updaten **UPDATE**. In dat geval is de syntaxis als volgt:

```
UPDATE "tabel_naam"
SET kolom 1 = [waarde 1], kolom 2 = [waarde 2]
WHERE "voorwaarde";
```

1.17 DELETE

Soms is het nodig records te verwijderen uit een tabel. Hiertoe kunt u de opdracht **DELETE FROM** gebruiken. De syntaxis is als volgt

```
DELETE FROM "tabel_naam"
WHERE {voorwaarde};
```

Dit kan het best worden getoond a.d.h.v. een voorbeeld. Als u bijvoorbeeld een tabel hebt zoals hieronder:

Tabel ***Store_Information***

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999
San Diego	250	07-Jan-1999
Los Angeles	300	08-Jan-1999
Boston	700	08-Jan-1999

en u beslist geen informatie over Los Angeles in de tabel te bewaren. Dit kan a.d.h.v. de volgende SQL:

```
DELETE FROM Store_Information
WHERE Store_Name = 'Los Angeles';
```

Nu ziet de tabel er als volgt uit:

Tabel ***Store_Information***

Store_Name	Sales	Txn_Date
San Diego	250	07-Jan-1999
Boston	700	08-Jan-1999

1.18 CREATE TABLE

Tabellen vormen de basisstructuur waarbinnen gegevens worden opgeslagen in de database. Vermits de databaseleverancier slechts zelden vooraf weet welke uw noden zijn op gebied van gegevensopslag, is het mogelijk dat u zelf tabellen moet aanmaken in de database. Veel databasetools laten toe tabellen aan te maken zonder SQL te schrijven, maar omdat tabellen de houders zijn van alle gegevens, is het belangrijk de syntaxis **CREATE TABLE** op te nemen in deze zelfstudie.

Voor u dieper ingaat op de SQL syntaxis voor **CREATE TABLE**, moet u zeker begrijpen waaruit een tabel bestaat. Tabellen worden verdeeld in rijen en kolommen. Elke rij vertegenwoordigt een gegeven, en elke kolom kan worden gezien als een onderdeel van dit gegeven. Als u bijvoorbeeld een tabel hebt voor de opslag van klantengegevens, dan kunnen de kolommen informatie omvatten zoals Voornaam, Familienaam, Adres, Plaats, Land, Geboortesatum enz. Daarom worden de

kolomkopteksten en gegevenstypes voor de kolom in kwestie opgenomen wanneer we een tabel specificeren.

wat zijn gegevenstypes? Gegevens bestaan in verschillende vormen. Het kan een integer zijn (zoals 1), een getal (zoals 0,55), een tekenreeks (zoals 'sql'), een datum/tijdsuitdrukking (zoals '2000-JAN-25 03:22:22'), of zelfs een binair formaat. Wanneer u een tabel specificeert, moet het gegevenstype geassocieerd met elke kolom worden gespecificeerd (d.w.z. u specificeert dat 'First Name' een type char(50) is - een tekenreeks van 50 tekens). Verschillende relationele databases aanvaarden verschillende gegevenstypes, dus raadpleegt u beter eerst een databasespecifieke referentie.

De SQL-syntaxis voor **CREATE TABLE** is:

```
CREATE TABLE "tabel_naam"
("kolom 1" "gegevens_type_voor_kolom_1",
"kolom 2" "gegevens_type_voor_kolom_2", ... );
```

Als u dus de hierboven vermelde klantentabel moet aanmaken, voert u in

```
CREATE TABLE Customer
(First_Name char(50),
Last_Name char(50),
Address char(50),
City char(50),
Country char(25),
Birth_Date datetime);
```

Soms kunt u een standaardwaarde willen voorzien voor elke kolom. U gebruikt een standaardwaarde wanneer de waarde van de kolom niet wordt gespecificeerd bij het toevoegen van gegevens aan de tabel. Voeg "Default [value]" toe achter de declaratie van het gegevenstype om een standaardwaarde te specificeren. Als u in het bovenstaand voorbeeld de kolom "Adres" wilt standardiseren als "Onbekend" en Plaats als "Amsterdam", voert u in

```
CREATE TABLE Customer
(First_Name char(50),
Last_Name char(50),
Address char(50) default 'Unknown',
City char(50) default 'Amsterdam',
Country char(25),
Birth_Date datetime);
```

1.19 ALTER TABLE

Wanneer een tabel is aangemaakt in de database, zijn er veel scenario's mogelijk waardoor u de structuur ervan wenst te wijzigen. Hierna volgen een aantal typische gevallen:

- Een kolom toevoegen
- Een kolom verwijderen
- Een kolomnaam wijzigen
- Het gegevenstype van een kolom wijzigen

Gelieve op te merken dat deze lijst niet volledig is. Er zijn andere gevallen waarin ALTER TABLE wordt gebruikt om de tabelstructuur te wijzigen, zoals de specificatie van de primaire sleutel wijzigen of een unieke beperkingen toevoegen voor een kolom.

De SQL-syntaxis voor **ALTER TABLE** is:

```
ALTER TABLE "tabel_naam"
[alter specification];
```

[alter specification] is afhankelijk van het type wijziging die u wenst door te voeren. Binnen het kader van de hierboven vermelde toepassingen, zijn de [alter specification] instructies:

- Een kolom toevoegen: ADD "kolom 1" "gegevenstype voor kolom 1"
- Een kolom verwijderen: DROP "kolom 1"
- Een kolomnaam wijzigen: CHANGE "oude kolomnaam" "nieuwe kolomnaam" "gegevenstype voor de nieuwe kolomnaam"
- Het gegevenstype van een kolom wijzigen: MODIFY "kolom 1" "nieuw gegevenstype"

Hierna volgen voorbeelden voor elk van de bovenstaande gevallen, a.d.h.v. de tabel "customer" aangemaakt in het deel **CREATE TABLE**:

Tabel ***Customer***

Kolom Naam	Gegevens Type
First_Name	char(50)
Last_Name	char(50)
Address	char(50)
City	char(50)
Country	char(25)
Birth_Date	datetime

Eerst wordt een kolom genaamd "Gender" toegevoegd aan de tabel. Hiertoe voert u in,

```
ALTER TABLE Customer ADD Gender char(1);
```

De tabelstructuur wordt als volgt:

Tabel ***Customer***

Kolom Naam	Gegevens Type
First_Name	char(50)
Last_Name	char(50)
Address	char(50)
City	char(50)
Country	char(25)
Birth_Date	datetime
Gender	char(1)

Daarna moet "Address" worden gewijzigd in "Addr". Hiertoe voert u in,

```
ALTER TABLE Customer CHANGE Address Addr char(50);
```

De tabelstructuur wordt als volgt:

Tabel ***Customer***

Kolom Naam	Gegevens Type
First_Name	char(50)
Last_Name	char(50)
Addr	char(50)
City	char(50)
Country	char(25)
Birth_Date	datetime
Gender	char(1)

Nu wordt het gegevenstype voor "Addr" gewijzigd naar char(30). Hiertoe voert u in,

```
ALTER TABLE Customer MODIFY Addr char(30);
```

De tabelstructuur wordt als volgt:

Tabel ***Customer***

Kolom Naam	Gegevens Type
First_Name	char(50)
Last_Name	char(50)
Addr	char(30)
City	char(50)

Country	char(25)
Birth_Date	datetime
Gender	char(1)

Ten laatste wordt de kolom "Gender" terug verwijderd. Hiertoe voert u in,

ALTER TABLE Customer DROP Gender;

De tabelstructuur wordt als volgt:

Tabel ***Customer***

Kolom Naam	Gegevens Type
First_Name	char(50)
Last_Name	char(50)
Addr	char(30)
City	char(50)
Country	char(25)
Birth_Date	datetime

1.20 DROP TABLE

Soms is het nodig een tabel te verwijderen, voor welke reden dan ook. Het zou problematisch zijn indien dit niet mogelijk was, want dit zou een onderhoudsnachtmerrie worden voor de DBA's. Gelukkig laat SQL dit toe met de opdracht **DROP TABLE**. De syntaxis voor **DROP TABLE** is:

DROP TABLE "tabel_naam";

Als u dus de tabel customer die werd aangemaakt in het deel **CREATE TABLE** wenst te verwijderen, voert u eenvoudigweg in

DROP TABLE Customer;

1.21 TRUNCATE TABLE

Soms is het nodig alle gegevens in een tabel te verwijderen. Een van de mogelijke manieren is **DROP TABLE**. Maar hoe verwijdert u de gegevens zonder aan de tabel te raken? Hiervoor gebruikt u de opdracht **TRUNCATE TABLE**. De syntaxis voor **TRUNCATE TABLE** is:

TRUNCATE TABLE "tabel_naam";

Als u dus de tabel customer die werd aangemaakt in het deel **SQL CREATE TABLE** wenst leeg te maken, voert u eenvoudigweg in:
TRUNCATE TABLE Customer;