FACULTEIT INGENIEURSWETENSCHAPPEN
EN ARCHITECTUUR

INFORMATICA

INLEIDING TOT
RELATIONELE DATABANKEN

UNIVERSITEIT
GENT

Hoofdstuk 1

Inleidende begrippen

Een *databank* laat toe informatie op een gestructureerde manier op te slaan. De mogelijkheden van de diverse beschikbare databanksystemen zijn erg uiteenlopend. Zo kan men bijvoorbeeld allerhande vormvereisten opleggen aan de informatie, geavanceerde zoekopdrachten uitvoeren, de informatie verspreiden over meerdere computers, enzovoort.

In deze nota's bespreken we aan de hand van een lopend voorbeeld de basisfunctionaliteit van *relationele* databanken. We hebben het o.a. over het aanmaken, opvullen, doorzoeken en beheren ervan, en geven een beknopte inleiding tot de *SQL*-taal.

Relationele databanken zijn databanken waarin de informatie onder de vorm van verschillende *tabellen* wordt opgeslagen, waartussen verbanden, de bewuste *relaties*, bestaan. Waarschijnlijk ben je het meest vertrouwd met tabellen onder de gedaante van een *rekenblad*, zoals aangeboden door de populaire toepassing *Microsoft Excel*.

Een doorsnee relationele databank bestaat uit heel wat tabellen. In ons voorbeeld gebruiken we er voor de eenvoud voorlopig maar eentje; vanaf hoofdstuk 3 wordt dit uitgebreid naar meerdere tabellen, met verbanden daartussen.

1.1 Inhoud en schema van een tabel

Beschouw volgende tabel, waarvan een (fictief) bedrijfje gebruikmaakt om de adresgegevens van zijn klanten bij te houden:

Klant	**Adres**
Huys Jasper	Kerkstraat 32B, 9920 Lovendegem
De Man Koen	Koestraat 19 bus 3 – 9420 Erpe-Mere
Verhulst Sara	Weidestraat 183, 3540 Herk-de-Stad
...	...

Net als in een rekenblad heeft een tabel de vorm van een matrix. De rijen hiervan worden *records* genoemd (spreek uit: *rékkerd*). Zo'n record bestaat uit een aantal *velden*, te vergelijken met de cellen van een rekenblad; een veld behoort dus tot een bepaald record en tot een bepaalde kolom.

In het voorbeeld heeft elk record twee velden. Het eerste record heeft bijvoorbeeld de waarde 'Huys Jasper' voor de kolom 'Klant'.

Bij een rekenblad kan elke *cel* een eigen type hebben; in Excel kan je zo vastleggen dat verschillende cellen tekst, getallen, valuta, datums, enz. bevatten. Bij een databank is dit niet het geval. Hier krijgt elke *kolom* een vast *(data)type* mee. Alle velden die tot een kolom behoren, hebben dus hetzelfde type.

Welke types er precies voorhanden zijn, hangt af van je databanksysteem. Enkele courante voorbeelden van types zijn 'Tekst', 'Geheel getal', 'Kommagetal' en 'Datum'. Verder in dit hoofdstuk besteden we hier uitgebreid aandacht aan.

De vormvereisten van een tabel, en dus van de databank, worden vastgelegd in wat men het *(databank)schema* noemt. Zo kunnen we voor bovenstaand eenvoudig tabelletje volgend schema vooropstellen:

SCHEMA

Kolom	**Type**	**Vereist**	**Beperkingen**
Klant	Tekst	Ja	–
Adres	Tekst	Ja	–

Maak een duidelijk onderscheid tussen de *inhoud* en het *schema* van een tabel! Het schema geeft enkel aan *hoe* de gegevens eruit moeten zien, terwijl de inhoud bepaalt *wat* er in de tabel zit. Ook wanneer we verderop in deze nota's een schema geven, zal je links ervan steeds het vignet SCHEMA zien staan.

In ons voorbeeld legt het *schema* vast dat zowel de kolom 'Klant' als de kolom 'Adres' het type 'Tekst' heeft, en dat voor beide kolommen de waarde *vereist* is. De *inhoud* bestaat uit de klantengegevens zelf.

Alle informatie is dan wel aanwezig in onze adrestabel, maar er zijn enkele verbeteringen mogelijk: we kunnen zowel naam als adres opsplitsen in verschillende delen, en dus kolommen. Dit biedt verschillende voordelen:

- We kunnen de kolomtypes strikter vastleggen. Een huisnummer is bijvoorbeeld steeds een strikt positief getal. Een bus kan eventueel wel worden aangeduid met een letter, dus die slaan we beter als tekst op.
- De tabel doorzoeken wordt makkelijker, zowel voor mens als computer. We kunnen bijvoorbeeld meteen alle klanten met dezelfde familienaam opsommen.
- We vermijden inconsistenties, zoals het scheiden van adresgegevens d.m.v. een komma of een liggend streepje.

Vormen we de tabel om, dan krijgen we uiteindelijk het schema en de inhoud hieronder. Vergelijk deze aandachtig met de eerste versie!

SCHEMA

Kolom	**Type**	**Vereist**	**Beperkingen**
Naam	Tekst	Ja	–
Voornaam	Tekst	Ja	–
Straat	Tekst	Ja	–
Nr	Geheel getal	Ja	Strikt positief
Bus	Tekst	Nee	–
Postcode	Geheel getal	Ja	Tussen 1000 en 9999
Gemeente	Tekst	Ja	–

Naam	**Voornaam**	**Straat**	**Nr**	**Bus**	**Postcode**	**Gemeente**
Huys	Jasper	Kerkstraat	32	B	9920	Lovendegem
De Man	Koen	Koestraat	19	3	9420	Erpe-Mere
Verhulst	Sara	Weidestraat	183		3540	Herk-de-Stad
...	...	...	...	...	...	...

1.2 Sleutels

Een belangrijk concept bij (relationele) databanken is dat van *sleutels*. We bespreken hier een eerste soort sleutel; op de andere soorten komen we later terug.

In elke tabel wordt een *primaire sleutel* aangeduid. In de praktijk is de primaire sleutel meestal één (vereiste) kolom, waarvan de velden allemaal verschillend moeten zijn. Dat betekent dat elk record ondubbelzinnig wordt geïdentificeerd door de waarde van zijn primaire-sleutelveld.

Vaak kiest men als primaire sleutel een uniek *ID*, zoals klantnummer, rijksregisternummer, code op studentenkaart, enz. Het type daarvan hoeft echter niet numeriek te zijn; een tekstuele kolom kan bijvoorbeeld ook.

Heeft onze adressentabel een primaire sleutel? We zouden de kolommen ‘Naam’ en ‘Voornaam’ samen als primaire sleutel kunnen gebruiken, maar dat is niet zo’n goed idee. We hebben immers geen enkele garantie dat er nooit twee klanten met dezelfde naam zullen voorkomen. Om dit probleem op te lossen gebruikt ons bedrijf voor elke klant een uniek klantnummer. We voegen deze kolom toe aan ons schema en gebruiken die als primaire sleutel.

De uiteindelijke tabel wordt hieronder afgebeeld. De klantnummering begint bij 1, maar dat is helemaal geen vereiste. Zolang elke rij maar een ander nummer krijgt, is de tabel geldig. Wel is het zo dat veel databanksoftware automatisch records kan nummeren en hierbij logischerwijs bij 1 (of soms 0) begint.

SCHEMA

Kolom	**Type**	**Vereist**	**Beperkingen**	
KLANTNR	Geheel getal	Ja	Strikt positief	←
Naam	Tekst	Ja	–	
Voornaam	Tekst	Ja	–	
...	...	...	...	
Gemeente	Tekst	Ja	–	

KLANTNR	**Naam**	**Voornaam**	...	**Gemeente**
1	Huys	Jasper	...	Lovendegem
2	De Man	Koen	...	Erpe-Mere
3	Verhulst	Sara	...	Herk-de-Stad
...	...	...	...	...

Bemerk dat de primaire sleutel in KLEINKAPITALEN wordt afgedrukt. Verderop in deze nota’s blijven we deze conventie gebruiken.

1.3 Het RDBMS

Een *relational database management system*, kortweg *RDBMS*, is een applicatie die het mogelijk maakt om relationele databanken te creëren, te bewerken en te doorzoeken. RDBMS’en zijn speciaal ontworpen om het makkelijk te maken om de inhoud van een databank op allerlei manieren te benaderen. Achter de schermen voert het RDBMS een aantal geavanceerde optimalisaties uit, waardoor het heel snel kan antwoorden op complexe zoekopdrachten.

1.3.1 Eenvoudige RDBMS'en

Aan de hand van een eenvoudig RDBMS kan de gebruiker een databankschema opstellen, tabelgegevens bewerken en informatie opzoeken via een grafische gebruikersinterface. Minimale voorkennis volstaat daarbij.

Voor de eenvoud wordt de volledige databank, dus zowel schema als tabelinhouden, doorgaans samengepropt in één enkel bestand. Deze situatie is vergelijkbaar met courante rekenbladapplicaties of tekstverwerkers en is eigenlijk enkel geschikt voor *lokaal* gebruik, d.w.z. op één computer.

Een voorbeeld van een dergelijke eenvoudige RDBMS is *Microsoft Access*, dat deel uitmaakt van Microsoft Office.

1.3.2 Professionele RDBMS'en

Professionele RDBMS'en daarentegen zijn meestal *client-server*-systemen, die gebruikmaken van een netwerk. Het RDBMS zelf is dan een applicatie die op een server draait. Clientapplicaties op andere computers kunnen de centrale databank aanspreken via een netwerkverbinding. Dergelijk professioneel RDBMS is in staat om meerdere databanken tegelijk te beheren. Deze blijven steeds volledig gescheiden van elkaar, maar elke databank kan natuurlijk wel uit meerdere tabellen bestaan, met verbanden daartussen.

De databankgegevens worden opgeslagen in één of meerdere bestanden, hetzij op dezelfde server, hetzij op aparte opslagservers. In de praktijk komt een gebruiker nooit in aanraking met die bestanden, gezien de databanken altijd aangesproken worden via het RDBMS. Het RDBMS en bijhorende databank(en)worden de "back end" genoemd, de clientapplicaties de "front end". Een professioneel RDBMS is meestal ook een *multi-user*-systeem. Dat wil zeggen dat meerdere gebruikers de databank tegelijkertijd kunnen aanspreken. Wel wordt vermeden dat twee gebruikers tegelijkertijd hetzelfde gegeven aanpassen.
Het is ook mogelijk om verschillende gebruikers specifieke rechten te geven. Stel bijvoorbeeld dat we salarissen zouden opslaan in de databank, dan kunnen we ervoor zorgen dat die enkel gewijzigd kunnen worden door de personeelsdienst.

Bekende voorbeelden van professionele RDBMS'en zijn *Oracle Rdb*, *Microsoft SQL Server*, *PostgreSQL*, *MySQL* en *IBM DB2*. De geavanceerde mogelijkheden hiervan vallen buiten het bereik van deze inleidende cursus.

Alle interactie met een professioneel RDBMS gebeurt door er contact mee te leggen vanuit een clientapplicatie (veelal via een netwerkverbinding), opdrachten door te geven en het resultaat daarvan te verwerken.

Stel dat je bijvoorbeeld het klantnummer en de naam van alle klanten uit de provincie Oost-Vlaanderen in alfabetische volgorde te zien wil krijgen; dit is het soort vraag dat je rechtstreeks kan stellen aan een RDBMS! Volgende stappen worden doorlopen:

1. De clientapplicatie maakt verbinding met het RDBMS.
2. De applicatie stelt het RDBMS de vraag "Geef me de waarde van de velden 'Klantnr' en 'Naam' van alle records uit de tabel 'Klanten' waarvoor de waarde van het veld 'Postcode' tussen 9000 en 9999 gelegen is, en sorteer dit alles volgens de waarde van het veld 'Naam'."
3. Het RDBMS bouwt een *tijdelijke tabel* op, met de kolommen 'Klantnr' en 'Naam'. Het type hiervan wordt afgeleid uit het schema.
4. Het RDBMS vult de tijdelijke tabel op door records toe te voegen met het klantnummer en de voornaam van klanten die beantwoorden aan de zoekopdracht. Tegelijk wordt er gesorteerd.
5. Het RDBMS stuurt de tijdelijke tabel door naar het programma. Deze zou er in ons voorbeeld als volgt uitzien:

   RESULTAAT

Klantnr (geheel getal)	**Naam (tekst)**
2	De Man
1	Huys

   Het antwoord op een vraag aan een RDBMS is dus steevast een tabel. Merk ook op dat het (beperkte) schema van die tabel wordt meegestuurd.
6. De applicatie verwerkt de tijdelijke tabel verder, bijvoorbeeld door de inhoud ervan af te drukken, via e-mail te versturen, ...
7. De applicatie verbreekt de verbinding met het RDBMS.
8. Het RDBMS gooit de tijdelijke tabel weg.

1.4 SQL

Alle communicatie met een RDBMS gebeurt aan de hand van *SQL*, wat staat voor *Structured Query Language*. Het is een geavanceerde, maar ook erg leesbare taal, specifiek ontwikkeld voor relationele databanken.

Een SQL-opdracht, meestal *query* genoemd, is een tekstuele voorstelling van de vraag die we willen stellen aan het RDBMS. De zoekopdracht die we net als voorbeeld gebruikten, zou zich vertalen in volgende query:

```
SELECT Klantnr, Naam
FROM Klanten
WHERE Postcode >= 9000 AND Postcode <= 9999
ORDER BY Naam
```

Zonder ons druk te maken over de specifieke vormvereisten van de query, kunnen we al vrij goed afleiden wat er gebeurt. *Uit* de tabel 'Klanten' worden records *geselecteerd*, *waarvoor* de kolom 'Postcode' aan een voorwaarde voldoet. Deze worden *geordend* en de inhouden van de kolommen 'Klantnr' en 'Naam' vormen de uiteindelijke resultaattabel.

In het volgende hoofdstuk bekijken we de mogelijkheden van de `SELECT`-opdracht van naderbij. Dit is echter niet het enige type SQL-opdracht; het is ook mogelijk om records toe te voegen, aan te passen en te verwijderen. Men spreekt dan van SQL als *DML* of *Data Manipulation Language*.

SQL kan ook gebruikt worden om het databankschema te wijzigen. In dat geval spreekt men van SQL als *DDL* of *Data Definition Language*. Zo bestaat bijvoorbeeld de opdracht `CREATE TABLE` om tabellen aan te maken.

Een derde en laatste toepassing van SQL is het gebruik als *DCL* of *Data Control Language*. Deze voorziet in het beheer van de gebruikersrechten die we al vermeldden. Met `GRANT` en `REVOKE` worden zulke rechten toegekend en ontnomen.

In deze nota's behandelen we SQL uitsluitend als DML, met de meeste nadruk op `SELECT`-query's.

Goed om weten is dat de SQL-taal niet vreselijk strikt wordt geïnterpreteerd. Zo mag je hoofdletters en kleine letters door elkaar gebruiken, en naar believen spaties of returns toevoegen.

Tot slot wensen we nog op te merken dat er eigenlijk diverse SQL-dialecten bestaan, die alle verderbouwen op de SQL-standaard(en).

Hoofdstuk 2

Zoeken in een tabel

We weten nu wat een relationele databank is en kwamen al heel kort in contact met de SQL-taal. In dit hoofdstuk hebben we het uitgebreid over de krachtige `SELECT`-opdracht, die toelaat tabellen op allerhande manieren te doorzoeken.

2.1 De `SELECT`-opdracht

Zoals we al kort even aangehaald hebben, is `SELECT` het type query om gegevens op te zoeken in een tabel. Zo kunnen we de volledige inhoud van onze tabel 'Klanten' opvragen met de tamelijk beknopte opdracht

```
SELECT *
FROM Klanten
```

Het sterretje staat voor "alle kolommen". We kunnen ook zelf kiezen welke kolommen we willen selecteren en in welke volgorde. Willen we bijvoorbeeld enkel klantnummer, voornaam en familienaam van alle klanten, dan gebruiken we

```
SELECT Klantnr, Voornaam, Naam
FROM Klanten
```

Zoals reeds besproken werd, komt het resultaat van een SELECT-query altijd terecht in een (voorlopige) resultaattabel. In bovenstaand geval is dat dus een tabel met drie kolommen.

Willen we geen dubbels te zien krijgen, dan voegen we het sleutelwoord `DISTINCT` toe.

Volgende opdracht geeft bijvoorbeeld alle *unieke* familienamen in het klantenbestand:

```
SELECT DISTINCT Naam
FROM Klanten
```

2.2 Resultaten ordenen: `ORDER BY`

Bij een `SELECT`-opdracht kunnen we aangeven in welke volgorde de records in de resultaattabel terecht moeten komen. Willen we deze bijvoorbeeld alfabetisch ordenen, d.w.z. oplopend volgens familienaam, dan schrijven we

```
SELECT *
FROM Klanten
ORDER BY Naam
```

Ook hier kunnen we in plaats van het sterretje willekeurige kolomnamen opgeven.

Wat met klanten met dezelfde familienaam? SQL laat toe om meerdere kolommen op te geven voor het ordenen. Bovendien is het mogelijk om de sorteervolgorde om te keren; dit doen we aan de hand van het sleutelwoord `DESC`, wat staat voor *descending*. Willen we de klantenlijst aflopend op familienaam ordenen, en klanten met dezelfde familienaam oplopend op voornaam, dan luidt de query:

```
SELECT *
FROM Klanten
ORDER BY Naam DESC, Voornaam
```

2.3 Resultaten filteren: `WHERE`

Tot nu toe hebben we steeds alle records opgevraagd. SQL maakt het echter ook mogelijk om voorwaarden op te geven waaraan records moeten voldoen. Dit gebeurt met behulp van de `WHERE`-clausule.

Willen we bijvoorbeeld op zoek gaan naar alle klanten uit de gemeente Lovendegem, dan doen we dit als volgt:

```
SELECT *
FROM Klanten
WHERE Gemeente = 'Lovendegem'
```

Let op de aanhalingstekens. Letterlijke tekst plaatsen we verplicht tussen enkele aanhalingstekens. Rond getallen worden geen aanhalingstekens geplaatst.

Merk op dat het veld 'Bus', in tegenstelling tot 'Nr', tekst mag bevatten; een klant kan immers op nummer 24 bus 3 wonen, maar evengoed op bus C. We schrijven daarom

```
WHERE Bus = '13'
```

mét aanhalingstekens, maar

```
WHERE Nr = 13
```

zonder. De waarde `13` is immers van het datatype 'Getal', maar `'13'` is 'Tekst'!

Voor zowel tekst als getallen wordt 'verschillend van' uitgedrukt met `<>`. Klanten die *niet* in Antwerpen wonen, vinden we dus terug met

```
SELECT *
FROM Klanten
WHERE Gemeente <> 'Antwerpen'
```

maar `Postcode <> 2000` zou ook werken, en vermijdt bovendien moeilijkheden met de gemeentes 'Anvers' en 'Atnweprne'. Merk op dat records met niet-ingevulde gemeente (`NULL`) *niet* in de resultaattabel opgenomen worden!

Wiskundige vergelijkingsoperatoren worden eveneens ondersteund, zowel voor getallen als voor tekstuele gegevens; bij die laatste wordt gekeken naar de alfabetische volgorde. Klanten met een huisnummer groter dan 10 vragen we zo op:

```
SELECT *
FROM Klanten
WHERE Nr > 10
```

Ook hier worden records waarvan het veld 'Nr' niet ingevuld is *niet* geselecteerd.

Herinner je dat de kolom 'Bus' niet *vereist* was. Is een veld niet ingevuld, dan krijgt het in SQL de waarde `NULL`. Klanten waarbij het veld 'Bus' geen waarde heeft, zoeken we als volgt op:

```
SELECT *
FROM Klanten
WHERE Bus IS NULL
```

Merk dus op dat we niet `WHERE Bus = NULL` schrijven! Het omgekeerde, d.w.z. alle klanten waarvoor 'Bus' wél ingevuld is, verkrijg je door

```
   WHERE Bus IS NOT NULL
```

te schrijven, en dus niet `WHERE Bus <> NULL`.

Wanneer een tekstveld de waarde `NULL` heeft, is dat niet hetzelfde als `''`, d.w.z. een stuk tekst van 0 tekens lang. Analoog zijn de waarden `NULL` en `0` voor numerieke velden wel degelijk verschillend!

Voorwaarden kunnen bovendien worden geïnverteerd d.m.v. `NOT` en samengesteld d.m.v. `AND` en `OR`. Om alle klanten uit de provincies Oost-Vlaanderen (postcodes 9000–9999) en Antwerpen (postcodes 2000–2999) te krijgen, schrijven we:

```
SELECT *
FROM Klanten
WHERE (Postcode >= 9000 AND Postcode <= 9999)
   OR (Postcode >= 2000 AND Postcode <= 2999)
```

Hierbij worden de klassieke voorrangsregels gehanteerd: eerst `NOT`, dan `AND`, dan `OR`. Je kunt ronde haakjes tussenvoegen om hiervan af te wijken. In bovenstaand voorbeeld zijn de haakjes dus zelfs optioneel.

Bij tekstvelden kunnen we *jokertekens* gebruiken. Een underscore staat voor eender welk karakter, terwijl een procentteken eender welke opeenvolging van karakters voorstelt. Om te zoeken met jokertekens gebruiken we `LIKE` en `NOT LIKE`, en dus niet = en <>. We gaan bijvoorbeeld op zoek naar alle klanten met een familienaam die begint met de letter D én een voornaam *zonder* a op de derde plaats:

```
SELECT *
FROM Klanten
WHERE Naam LIKE 'D%'
  AND Voornaam NOT LIKE '__a%'
```

Je kan hier bijvoorbeeld nog een `ORDER`-clausule aan toevoegen om alfabetisch te sorteren.

2.4 Statistische functies

Soms ben je niet zozeer geïnteresseerd in de records die aan je zoekopdracht voldoen, maar bijvoorbeeld in het *aantal* resultaten of de *totale som* over een bepaald veld. SQL heeft een aantal statistische functies die hiervoor gebruikt kunnen worden.

De eenvoudigste van deze functies is `COUNT`, die het aantal overeenkomende records teruggeeft. Zo vragen we het totale aantal records in de tabel 'Klanten' op:

```
SELECT COUNT(*)
FROM Klanten
```

Tussen de haakjes kunnen ook kolomnamen worden opgegeven. In combinatie met het sleutelwoord `DISTINCT` geeft dit interessante mogelijkheden. Zo kunnen we bijvoorbeeld het aantal verschillende voornamen tellen:

```
SELECT COUNT(DISTINCT Voornaam)
FROM Klanten
```

Uit numerieke kolommen kunnen we een heleboel statistische eigenschappen afleiden. Zo kunnen we de som van alle postcodes bepalen met de functie `SUM`:

```
SELECT SUM(Postcode)
FROM Klanten
```

Op gelijkaardige wijze geeft `AVG` de gemiddelde waarde en `MIN` en `MAX` respectievelijk de kleinste en grootste waarde. Merk op dat `NULL`-waarden door deze numerieke functies genegeerd worden.

Ook hier kunnen we een `WHERE`-clausule toevoegen. Het kleinste huisnummer in Erpe-Mere vinden we bijvoorbeeld als volgt:

```
SELECT MIN(Nr)
FROM Klanten
WHERE Gemeente = 'Erpe-Mere'
```

2.5 Resultaten groeperen: `GROUP BY`

Statistische functies worden vaak in combinatie met `GROUP`-clausules gebruikt. Deze laten toe om resultaten op te splitsen volgens de waarde van één of meerdere kolommen.

Zo wordt geavanceerde analyse van de gegevens mogelijk.

Het aantal klanten per postcode vragen we bijvoorbeeld als volgt op:

```
SELECT Postcode, COUNT(*)
FROM Klanten
GROUP BY Postcode
```

Elk record in het resultaat heeft dan twee numerieke velden: een postcode en het bijhorende aantal records in de tabel 'Klanten'.

Opnieuw kunnen we een `WHERE`-clausule toevoegen. Ook een `ORDER`-clausule kan zinvol zijn. Zo geeft onderstaande query het aantal verschillende huisnummers per postcode, waarbij de postcodes aflopend worden geordend.

```
SELECT Postcode, COUNT(DISTINCT Nr)
FROM Klanten
GROUP BY Postcode
ORDER BY Postcode DESC
```

Wat als we dit resultaat willen ordenen volgens het aantal huisnummers? Hoewel SQL toelaat om letterlijk `COUNT(DISTINCT Nr)` te herhalen in de `ORDER`-clausule, is er een beter alternatief. Het sleutelwoord `AS` maakt het mogelijk om de kolom te benoemen en gebruik te maken van haar tijdelijke naam:

```
SELECT Postcode, COUNT(DISTINCT Nr) AS Aantal
FROM Klanten
GROUP BY Postcode
ORDER BY Aantal DESC
```

2.6 Resultaten filteren na groepering: `HAVING`

Ook ná een `GROUP`-clausule te hebben toegepast, kunnen we de resultaten nog filteren. We grijpen even terug naar onze query voor het aantal klanten per postcode. Stel dat we nu enkel de postcodes willen afdrukken waar vijf of meer klanten mee overeenkomen. Een `WHERE`-clausule kunnen we hiervoor niet gebruiken, maar er is wel iets zeer gelijkaardigs beschikbaar: de `HAVING`-clausule.

```
SELECT Postcode, COUNT(*)
FROM Klanten
GROUP BY Postcode
HAVING COUNT(*) >= 5
```

Bemerk dat we bij `HAVING` de kolommen helaas niet vooraf kunnen benoemen. De `COUNT(*)` moet hier dus letterlijk worden herhaald.

Overigens is het wel degelijk mogelijk om `WHERE` en `HAVING` in dezelfde opdracht op te nemen. Beschouw deze uitbreiding van bovenstaande query, die alle klanten genaamd Kris negeert bij het aggregeren:

```
SELECT Postcode, COUNT(*)
FROM Klanten
WHERE Voornaam <> 'Kris'
GROUP BY Postcode
HAVING COUNT(*) >= 5
```

Hoofdstuk 3

Zoeken in meerdere tabellen

Misschien begon je je al af te vragen waar het woord "relationeel" vandaan kwam. Welnu, in een *relationele* databank worden verbanden gelegd tussen verschillende tabellen, of beter, de kolommen ervan. Deze verbanden zijn precies de *relaties*.

De klantendatabank die we tot nu toe gebruikten, bestaat slechts uit één tabel. We voeren er nu een tweede in en koppelen beide door middel van een relatie.

3.1 Een tweede tabel

Stel even dat ons fictief bedrijf meerdere vestigingen heeft. Hoewel onze klanten natuurlijk welkom zijn in al onze vestigingen, hebben we van elke klant een fysiek *dossier*, dat zich bevindt in de vestiging waar de klant het meest komt.

Natuurlijk willen we die informatie ook bijhouden in de databank. In eerste instantie willen we dus weten bij welke vestiging een klant hoort, maar bovendien houden we van elke vestiging ook telefoonnummer, postcode en gemeente bij.

Indien we onze klantentabel hiervoor zouden uitbreiden, dan kregen we dit:

KLANTNR	Naam	Voornaam	...	Vestiging	Postcode vest.	Telefoon vest.
1	Huys	Jasper	...	Gent	9000	09 243 87 87
2	De Man	Koen	...	Gent	9000	09 243 87 87
3	Verhulst	Sara	...	Hasselt	3500	–
...	...	...	...	...	...	...

Niet alleen past de tabel niet meer op deze pagina, maar bovendien ontstaat een prangend probleem, namelijk dat van *dubbele informatie*[1]. De eerste twee klanten horen bij dezelfde vestiging, dus alle kolommen die betrekking hebben op die vestiging, bevatten dezelfde gegevens.

Als bijvoorbeeld het telefoonnummer van een vestiging gewijzigd moet worden, dan moet elke record afzonderlijk worden aangepast. Bovendien kunnen er makkelijk inconsistenties ontstaan, bijvoorbeeld door tikfouten.

Hoe kunnen we dit probleem oplossen? In plaats van de klantentabel uit te breiden, voorzien we een tweede tabel, waarin we de vestigingsgegevens opslaan:

SCHEMA

Kolom	**Type**	**Vereist**	**Beperkingen**
Postcode	Geheel getal	Ja	Tussen 1000 en 9999
Gemeente	Tekst	Ja	–
Telefoon	Tekst	Nee	–

Postcode	**Gemeente**	**Telefoon**
9000	Gent	09 243 87 87
3500	Hasselt	–
...	...	...

3.2 Een relatie vastleggen

Nu hebben we wel twee tabellen, maar we weten nog niet welke klanten bij welke vestiging geregistreerd zijn. Precies daarvoor voegen we nu een *relatie* toe: bij elke klant willen we een verwijzing bijhouden naar het record met zijn vestiging.

Maar hoe kunnen we verwijzen naar records in de tabel ‘Vestigingen’? Hiervoor moeten we elk record in die tabel uniek kunnen identificeren. Als dit je bekend in de oren klinkt, dan is dat omdat het precies het doel is van een *primaire sleutel*.

Als ons bedrijf slechts één vestiging per gemeente had, dan konden we eventueel de postcode ervan als primaire sleutel instellen. We zijn echter ambitieus en voegen veiligheidshalve de nieuwe kolom ‘Vestcode’ toe aan het schema:

[1] Dubbele informatie hoeft niet steeds een slechte zaak te zijn. Om de prestaties van een databank te verbeteren, kan het soms juist nuttig zijn om gegevens op verschillende plaatsen te dupliceren, zodat ze sneller terug te vinden zijn.

SCHEMA

Kolom	Type	Vereist	Beperkingen	
VESTCODE	Tekst	Ja	–	←
Postcode	Geheel getal	Ja	Tussen 1000 en 9999	
Gemeente	Tekst	Ja	–	
Telefoon	Tekst	Nee	–	

VESTCODE	Postcode	Gemeente	Telefoon
GEN1	9000	Gent	09 243 87 87
HAS1	3500	Hasselt	–
...	...	...	

Hoe leggen we nu het verband tussen beide tabellen? Elke klant is bij één vestiging geregistreerd, maar omgekeerd hoort elke vestiging bij één of meerdere klanten. Dit noemen we een *één-op-veel-* of *één-op-n-relatie*.

Om die relatie te modelleren, slaan we bij elke klant een verwijzing naar een vestigingsrecord op. Zo'n verwijzing noemen we een *verwijssleutel* of *foreign key*.

SCHEMA

Kolom	Type	Vereist	Beperkingen	
KLANTNR	Geheel getal	Ja	Strikt positief	
Naam	Tekst	Ja	–	
Voornaam	Tekst	Ja	–	
Straat	Tekst	Ja	–	
Nr	Geheel getal	Ja	Strikt positief	
Bus	Tekst	Nee	–	
Postcode	Geheel getal	Ja	Tussen 1000 en 9999	
Gemeente	Tekst	Ja	–	
Vestcode	Tekst	Ja	Verwijst naar 'Vestcode' in 'Vestigingen'	←

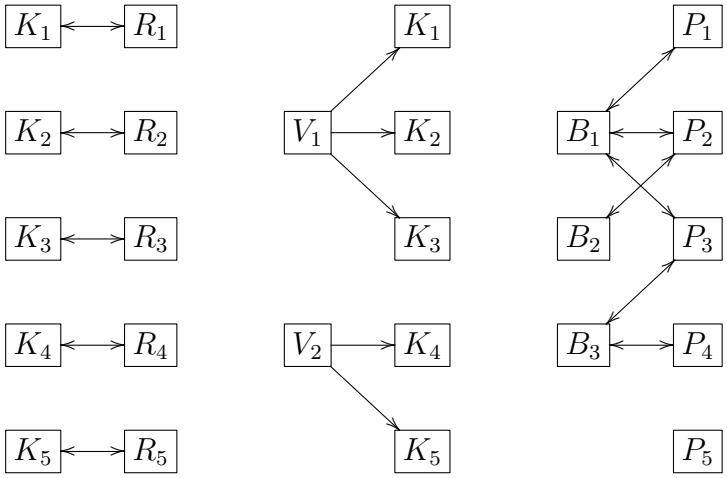
KLANTNR	Naam	Voornaam	...	Vestcode
1	Huys	Jasper	...	GEN1
2	De Man	Koen	...	GEN1
3	Verhulst	Sara	...	HAS1
...	...	...	...	...

Wanneer we nu de vestigingsgegevens nodig hebben die horen bij een klant, moet zijn vestigingscode worden opgezocht in de vestigingentabel. Aangezien die code de primaire sleutel is, gaat dat opzoeken zeer snel.

Het omgekeerde is al minder evident: als je alle klanten horend bij een bepaalde vestiging wil oplijsten, moet de klantentabel worden doorzocht naar records met de betreffende vestigingscode.

3.3 Soorten relaties

Naast de één-op-veel-relatie die we net besproken hebben, kennen relationele databanken nog twee soorten relaties:



(a) Eén-op-één (b) Eén-op-veel (c) Veel-op-veel

Figuur 3.1: De drie types relaties

- Een **één-op-één-relatie** is de eenvoudigste in zijn soort: met elk uniek gegeven stemt juist één ander uniek gegeven overeen en vice versa.

  In onze databank zouden we bijvoorbeeld het rijksregisternummer van elke klant kunnen bijhouden. Dit zou niet meer zijn dan een nieuwe kolom in de tabel 'Klanten', maar toch hebben we hier te maken met een één-op-één-relatie, namelijk tussen klantnummer en rijksregisternummer: met elk klantnummer komt juist één rijksregisternummer overeen, met elk rijksregisternummer één klantnummer. Figuur 3.1(a) geeft dit schematisch weer.

  Eén-op-één-relaties vereisen dus geen extra tabel (al mag dit ook). Wel geven ze aanleiding tot het derde en laatste type sleutel: de *unieke sleutel* of *unique key*, in ons voorbeeld het rijksregisternummer van de klant.

- Met een **één-op-veel-** of **één-op-n-relatie** kwamen we reeds in contact. Bij dergelijke relatie wordt telkens één uniek gegeven geassocieerd met nul, één of meerdere gegevens. In het voorbeeld koppelden we een klant aan zijn vestiging. Op Figuur 3.1(b) wordt dit nogmaals geïllustreerd.

  Doordat we binnen één relatie te maken hebben met meerdere koppels, volstaat één enkele tabel niet meer. Zowel bij de "één"- als bij de "veel"-kant van de relatie hoort een afzonderlijke tabel, en in de eerste wordt een verwijzing bijgehouden naar de tweede, namelijk de *verwijssleutel* of *foreign key*.

- Een **veel-op-veel-** of **n-op-n-relatie** ten slotte is de meest complexe vorm. Hierbij bestaat een verband tussen nul, één of meerdere gegevens van de ene soort, en nul, één of meerdere gegevens van de andere.

  Een klassiek voorbeeld is dat van bestellingen. Stel dat we twee tabellen toevoegen, eentje voor producten en eentje voor bestellingen daarvan:

PRODUCTNR	Beschrijving
61	Broodrooster
62	Microgolfoven
63	Espressomachine
...	...

BESTELLINGNR	Klantnr
1001	1
1002	2
1003	1
...	...

  Als we bij elke bestelling de daarbij horende producen willen opslaan, dan ontstaat een veel-op-veel-relatie. Immers, een bestelling zal vaak uit meerdere producten bestaan, maar elk product kan ook meerdere keren besteld worden, al dan niet door verschillende klanten. Figuur 3.1(c) geeft een schematische voorstelling van dit alles.

  We kunnen aan geen van beide tabellen een kolom toevoegen om de veel-op-veel-relatie bij te houden. Wat we nodig hebben is nóg een tabel: een zogenaamde *koppeltabel*, die alle associaties tussen bestellingnummers en productnummers oplijst. Bovendien kunnen we deze tabel bijvoorbeeld gebruiken om voor elk product het gewenste aantal op te slaan.

BESTELLINGNR	PRODUCTNR	Aantal
1001	62	1
1001	63	5
1002	62	1
...	...	...

  Al is de veel-op-veel-relatie waarschijnlijk het interessantste type, we gaan er in deze inleidende nota's niet dieper op in.

3.4 Zoeken in meerdere tabellen

Nu onze databank uit twee tabellen bestaat, met daartussen een relatie, kunnen we hiervan gebruikmaken om de `SELECT`-opdracht verder uit te diepen.

Zoals we intussen weten, bestaat in onze databank een één-op-veel-relatie tussen de tabellen 'Vestigingen' en 'Klanten'. We kunnen een SQL-query deze relatie laten reflecteren door middel van een `JOIN`-clausule.

De meest eenvoudige JOIN-query ziet er voor onze databank als volgt uit:

```
SELECT *
FROM Klanten
  INNER JOIN Vestigingen
  ON Klanten.Vestcode = Vestigingen.Vestcode
```

Het begin van deze query is ons niet vreemd: we vragen alle velden op. In het FROM-gedeelte staan nu echter twee tabellen vermeld i.p.v. één, gekoppeld door het sleutelwoord JOIN. Bij zo'n JOIN hoort verplicht ON, gevolgd door de voorwaarde die de relatie beschrijft. In ons geval wordt de relatie vastgelegd door de kolom 'Vestcode' van 'Klanten', dat een verwijzing is naar de gelijknamige kolom van 'Vestigingen'. Het resultaat van deze query is volgende (tijdelijke) tabel:

RESULTAAT

Klantnr	**Naam**	...	**Vestcode**	...	**Telefoon**
1	Huys	...	GEN1	...	09 243 87 87
2	De Man	...	GEN1	...	09 243 87 87
3	Verhulst	...	HAS1	...	–
...	...	...	...	...	...

Zoals je kan zien, is het effect van deze JOIN dat elk klantenrecord wordt aangevuld met het bijhorende vestigingsrecord. Misschien valt het je op dat dit precies de tabelstructuur geeft waarin we de dubbele informatie elimineerden.

Er is nog een alternatieve schrijfwijze voor de JOIN, die enkel gebruikmaakt van een WHERE-clausule. Voor bovenstaande query ziet die er als volgt uit:

```
SELECT *
FROM Klanten, Vestigingen
WHERE
  Klanten.Vestcode = Vestigingen.Vestcode
```

Resultaat en achterliggende werking van beide query's zijn identiek. Je mag dus zelf kiezen welke van de twee je gebruikt.

3.4.1 Velden kiezen

Om bij de JOIN de velden op te geven die we te zien willen krijgen, moeten we voorzichtig zijn. Het kan nu immers voorkomen dat beide tabellen gelijknamige kolommen bevatten.

In onze databank hebben beide tabellen een kolom met de naam ‘Vestcode’. Als we dit veld willen afdrukken, moeten we expliciet de naam van de tabel vermelden, zelfs al is de waarde in dit geval dezelfde! In onderstaande query is het voorvoegsel `"Klanten."` dus verplicht.

```
SELECT Klanten.Vestcode, Naam, Straat
FROM Klanten
  INNER JOIN Vestigingen
  ON Klanten.Vestcode = Vestigingen.Vestcode
```

Het is zelfs aan te raden om het voorvoegsel bij élke kolom te vermelden! Dit vermijdt verwarring en maakt de query beter bestand tegen eventuele aanpassingen aan het databankschema.

3.4.2 Verdere mogelijkheden

Alle besproken mogelijkheden van zoeken blijven ook bij een `JOIN`-query geldig. Je kan dus met een gerust hart `WHERE`, `ORDER`, `GROUP` en `HAVING` blijven gebruiken. Wees ook hier echter op je hoede voor kolommen met dezelfde naam!

Hoofdstuk 4

Gegevens manipuleren

Totnogtoe hebben we het enkel gehad over zoeken in een databank. Zoals vermeld, is SQL echter de taal die gebruikt wordt voor álle bewerkingen op de databank. In dit hoofdstuk bespreken we dan ook hoe SQL wordt ingezet voor het toevoegen, aanpassen en verwijderen van gegevens.

4.1 Gegevens toevoegen: `INSERT`

Met een `INSERT`-opdracht voeg je één of meerdere records toe aan een tabel. Deze opdracht voegt bijvoorbeeld drie vestigingen toe:

```
INSERT INTO Vestigingen (Vestcode, Postcode, Telefoon)
VALUES
  ('BRU1',     1000,       '021111111'),
  ('BRU2',     1040,       '024441111'),
  ('ZZB1',     3770,       NULL)
```

4.2 Gegevens wijzigen: `UPDATE`

Een `UPDATE`-opdracht past nul, één of meerdere records van een tabel tegelijk aan. Door een optionele `WHERE`-clausule toe te voegen, kunnen we aangeven welke record(s) we willen wijzigen.

Om de voornaam van alle klanten in te stellen op 'Jan', doen we dit:

```
UPDATE Klanten
SET Voornaam = 'Jan'
```

Echt nuttig is dat natuurlijk niet. We kunnen echter een `WHERE`-clausule toevoegen om slechts bepaalde rijen aan te passen. Willen we bijvoorbeeld de familienaam van klant 18 wijzigen naar 'Van de Velde', dan voeren we deze opdracht uit:

```
UPDATE Klanten
SET Naam = 'Van de Velde'
WHERE Klantnr = 18
```

Meerdere records tegelijk aanpassen is echter het interessantst. Stel dat we alle gemeenten met postcode 1000 voortaan 'Broekzele' willen noemen en bovendien alle busnummers willen verwijderen, dan typen we:

```
UPDATE Klanten
SET Gemeente = 'Broekzele', Bus = NULL
WHERE Postcode = 1000
```

4.3 Gegevens verwijderen: `DELETE`

Nul, één of meerdere records verwijderen uit een tabel doen we aan de hand van de `DELETE`-opdracht. Tenzij we de volledige tabel willen leeg maken, is die telkens vergezeld van een `WHERE`-clausule.

Volgende opdracht verwijdert de vestiging met code 'ZZB1':

```
DELETE FROM Vestigingen
WHERE Vestcode = 'ZZB1'
```

Willen we bijvoorbeeld alle klanten uit Hasselt verwijderen, dan doen we dit:

```
DELETE FROM Klanten
WHERE Gemeente = 'Hasselt'
```