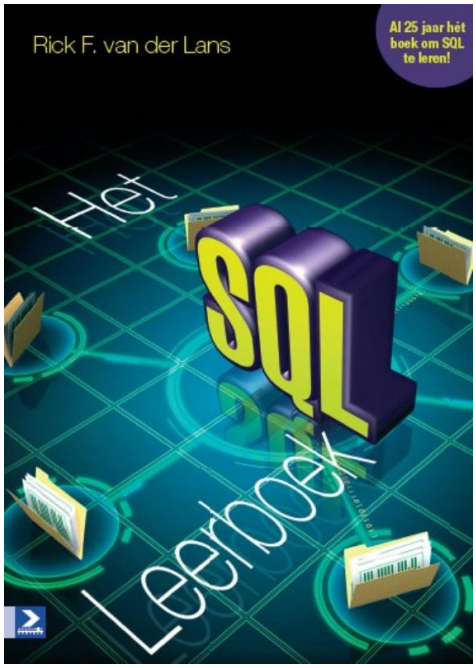
Het SQL Leerboek – zevende editie
Objectrelationele Concepten in SQL



Auteur: Rick F. van der Lans
Versie: 1.0
Datum: Februari 2012

Hoofdstuk 1

Inleiding

1.1 Inleiding

Dit document bevat twee hoofdstukken die tot aan de zesde editie onderdeel van *Het SQL leerboek* zijn geweest. Voor de zevende editie is besloten deze niet in het boek op te nemen, maar hiervoor een separaat document te creëren dat gedownload kan worden.

Er zijn twee redenen om deze hoofdstukken uit *Het SQL Leerboek* weg te laten. Ten eerste, om het boek niet al te omvangrijk te maken en daarmee de kosten relatief laag te houden, en ten tweede omdat dit onderwerp niet voor alle lezers relevant is. Door het te kunnen downloaden, blijft de tekst wel voor iedereen beschikbaar.

Als u vragen of opmerkingen hebt op dit document, laat ons dat weten via email adres sql@r20.nl

1.2 Wat is objectrelationeel?

In de jaren zeventig zijn een aantal concepten geïntroduceerd die later grote invloed hebben gehad op veel deelgebieden van automatisering. Deze *objectgeoriënteerde concepten* (OO-concepten) werden als eerste door programmeertalen geadopteerd. Talen als C en Pascal en later ook COBOL werden hiermee uitgebreid. C++ was bijvoorbeeld de objectgeoriënteerde versie van C. Smalltalk, Java en C# waren vanaf hun ontstaan objectgeoriënteerd. Later werden ook analyse- en ontwerpmethoden, besturingssystemen en CASE-tools met OO-concepten uitgebreid.

Op een gegeven moment waren ook de databases aan de beurt. Een hele groep van nieuwe databases werd geïntroduceerd, alle volledig gebaseerd op de OO-concepten. Maar deze producten ondersteunden aanvankelijk geen SQL. Wanneer een bedrijf in deze technologie geïnteresseerd was, was een zware en kostbare migratie van de bestaande SQL-database vereist. De leveranciers van relationele databases realiseerden zich dat er behoefte in de markt was voor deze OO-concepten en besloten ze aan hun eigen SQL-producten toe te voegen. Het huwelijk tussen OO en relationeel was een feit en de naam werd *objectrelationele database*. Maar er worden ook andere namen in de literatuur gebruikt, zoals *universal database*, *extensible database* en *non-first normal form database*.

Rond 1997 introduceerden de belangrijke databaseleveranciers hun eerste implementaties van hun objectrelationele producten op de markt. In 1998 is de SQL3-standaard verschenen en ook zijn veel van deze nieuwe

concepten toegevoegd. Helaas zijn de verschillen tussen wat de leveranciers geïmplementeerd hebben groot tot zeer groot.

In dit document gaan we in op wat we momenteel beschouwen als objectrelationele concepten. We hebben hier echter twee opmerkingen over. Ten eerste, over enkele jaren zal deze term waarschijnlijk vervallen. Deze nieuwe concepten zullen dan geaccepteerd zijn en niet meer als speciaal of apart worden gezien. Toekomstige gebruikers zullen niet merken dat deze objectrelationele concepten pas achteraf aan SQL zijn toegevoegd. Ten tweede, het staat nog steeds ter discussie of deze concepten wel allemaal uit de wereld van objectoriëntatie voortkomen. En bij sommige moet gezegd worden dat dit waarschijnlijk niet zo is. We zullen in dit document geen scheidsrechter spelen en kwalificeren ze daarom allemaal onder deze noemer.

De mogelijkheden die de SQL-producten en SQL3 bieden, variëren sterk en de syntaxis die ze gekozen hebben, wijkt van elkaar af. Daarom hebben we in dit deel gekozen voor een syntaxis die lijkt op die van enkele producten. Vanwege die productverschillen geven we echter geen syntaxisdefinities. Het document is voornamelijk toegevoegd om lezers een globaal beeld te geven van hoe die objectrelationele concepten er binnen SQL uitzien en welk effect ze op de SQL-instructies hebben.

1.3 Informatiebronnen over objectrelationele databases

Voor meer algemene informatie over objectrelationele databases zie de volgende bronnen:

- Wikipedia http://en.wikipedia.org/wiki/Object-relational_database
- Paul Brown, *'Object-Relational Database Development: A Plumber's Guide'*, Prentice Hall, 2000.
- C. Delobel, C. Lécluse & P. Richard, *'Databases: From Relational to Object-Oriented Systems'*, International Thomson Publishing, 1995.
- M. Stonebraker, D. Moore en P. Brown, *'Objectrelational Database Servers, the Next Great Wave'*, Morgan Kaufmann Publishers, 1999.

Hoofdstuk 2

Zelfgedefinieerde datatypes, Functies en operatoren

2.1 Inleiding

In *Het SQL Leerboek* zijn datatypes als INTEGER, CHAR en DATE beschreven. Dit zijn zogenaamde *basisdatatypes*. Basisdatatypes worden met SQL meegeleverd. Ze bieden bepaalde mogelijkheden en we kunnen er vooraf gedefinieerde operaties op loslaten. Bijvoorbeeld, met het INTEGER-datatype kunnen we berekeningen maken en we kunnen er de operatoren + en – op uitvoeren. Maar de basisdatatypes van SQL zijn zeer elementair. Sommige gebruikers hebben behoefte aan veel meer geavanceerde en specialistische datatypes. In een omgeving waar geografische gegevens worden vastgelegd, zou bijvoorbeeld het datatype tweedimensionale coördinaat zeer waardevol kunnen zijn. En in een verffabriek kan het nuttig zijn het datatype kleur te hebben. Uiteraard willen we daar dan ook de operatoren voor hebben. Voor het datatype 2D-coördinaat willen we operatoren hebben als 'bereken de afstand tussen twee coördinaten' en voor kleur 'meng twee kleuren'.

Steeds meer SQL-producten staan toe dat gebruikers hun eigen datatypes met bijbehorende operatoren kunnen definiëren. Om onderscheid te maken, worden dit *zelfgedefinieerde datatypes* genoemd (user defined datatypes). Maar niet alleen enkele SQL-producten, ook de SQL3-standaard ondersteunt zelfgedefinieerde datatypes.

SQL kent diverse soorten zelfgedefinieerde datatypes. In dit hoofdstuk behandelen we onder andere de datatypes distinct-, opaque- en named row-type. Van elk datatype wordt behandeld wat met de SQL-producten op dit gebied mogelijk is en wat in de SQL3-standaard is gedefinieerd. Tevens komt het bouwen van zelfgedefinieerde functies en operatoren aan bod.

2.2 Zelf definiëren van datatypes

Uiteraard moeten zelfgedefinieerde datatypes gecreëerd worden. Net als voor het creëren van tabellen, views en synoniemen, bestaat hier een speciale CREATE-instructie voor.

Voorbeeld 2.1: Creëer de datatypes genaamd BETALINGSNR, SPELERSNR en GELDBEDRAG en gebruik ze vervolgens in de CREATE TABLE-instructie voor de BOETES-tabel.

```
CREATE TYPE BETALINGSNR AS INTEGER

CREATE TYPE SPELERSNR AS INTEGER

CREATE TYPE GELDBEDRAG AS DECIMAL(7,2)

CREATE TABLE BOETES (
   BETALINGSNR   BETALINGSNR NOT NULL PRIMARY KEY,
   SPELERSNR     SPELERSNR,
   DATUM         DATE,
   BEDRAG        GELDBEDRAG)
```

Toelichting: Zelfgedefinieerde datatypes worden op de plaats gebruikt waar normaliter basisdatatypes staan. Dit mag in principe altijd. Overal waar een basisdatatype kan worden ingezet, mag een zelfgedefinieerd datatype worden gebruikt. Uit het voorbeeld blijkt tevens dat kolomnamen en datatypenamen hetzelfde mogen zijn.

Zelfgedefinieerde datatypes hebben veel overeenkomsten met basisdatatypes. Een van die overeenkomsten is dat een datatype geen populatie of 'inhoud' heeft (een tabel daarentegen wel). Met bijvoorbeeld INSERT-instructies worden rijen aan een tabel toegevoegd en wordt de inhoud opgebouwd. INSERT- en andere instructies zijn echter niet uit te voeren op een datatype. Datatypes zijn in hun geheel niet te manipuleren. We kunnen bijvoorbeeld niet met een SELECT-instructie alle mogelijke waarden van het INTEGER- of een zelfgedefinieerd datatype opvragen. Men zou kunnen stellen dat een datatype wel een statische, virtuele inhoud heeft. Deze virtuele inhoud bestaat uit alle waarden die kunnen voorkomen in het onderliggende datatype. Dus, bij het GELDBEDRAG-datatype zijn alle numerieke waarden tussen -9.999.999,99 en 9.999.999,99 toegestaan. In feite is een SQL-datatype vergelijkbaar met een type in Pascal, of een klasse in Java; het datatype beschrijft mogelijke waarden.

In de bovenstaande CREATE TYPE-instructies is duidelijk te zien dat een zelfgedefinieerd datatype rust op een basisdatatype. Zelfgedefinieerde datatypes mogen ook naar elkaar refereren.

Voorbeeld 2.2: Creëer het datatype KLEIN_GELDBEDRAG.

```
CREATE TYPE KLEIN_GELDBEDRAG AS GELDBEDRAG
```

Er bestaan diverse soorten zelfgedefinieerde datatypes. Degene die hierboven gecreëerd zijn, worden distinct-datatypes genoemd. Een *distinct-datatype* is direct of indirect (via een ander distinct-datatype) gedefinieerd op een bestaand basisdatatype. In de volgende paragrafen behandelen we de andere soorten.

Een van de grote voordelen van het werken met zelfgedefinieerde datatypes is dat appels niet meer met peren vergeleken kunnen worden. De volgende SELECT-instructie is wel toegestaan als de twee kolommen op hetzelfde basisdatatype gedefinieerd zijn, maar is nu niet meer toegestaan:

```
SELECT   *
FROM     BOETES
WHERE    BETALINGSNR > BEDRAG
```

Het mocht omdat beide kolommen numeriek waren. Nu de BEDRAG-kolom gedefinieerd is op het datatype GELDBEDRAG kan deze alleen nog maar vergeleken worden met kolommen die op hetzelfde datatype gedefinieerd zijn. Op zich klinkt dit als een beperking, maar het is eigenlijk een voordeel. De conditie in de SELECT-instructie was toch maar een vreemde vraag! Met andere woorden, het voordeel van het werken met zelfgedefinieerde datatypes is dat onzinnige instructies worden afgekeurd. In de wereld van programmeertalen wordt dit *strong typing* genoemd. Talen als Algol, Pascal en Java hebben dit concept altijd ondersteund. Let wel, het is wel mogelijk waarden van verschillende datatypes te vergelijken, maar dan moeten we dat expliciet opgeven. We komen hier later op terug.

Datatypes kunnen ook heel eenvoudig verwijderd worden. Bijvoorbeeld:

```
DROP TYPE GELDBEDRAG
```

Wat gebeurt er echter als een datatype wordt verwijderd, terwijl er wel kolommen op zijn gedefinieerd? Het antwoord op deze vraag is: dit is productafhankelijk. Sommige producten staan het verwijderen van datatypes alleen toe als er geen kolommen of andere zelfgedefinieerde datatypes op gedefinieerd zijn. Andere staan het wel toe en vervangen dan het zelfgedefinieerde datatype van de kolom door het onderliggende datatype. Met andere woorden, de specificatie van het te verwijderen datatype verhuist dan naar die van alle kolommen met dat datatype.

In de literatuur over het relationele model wordt eerder de term *domein* dan datatype gebruikt.

2.3 Toegang tot datatypes

Datatypes hebben meestal een eigenaar. Diegene die ze creëert is de eigenaar van het datatype. Andere gebruikers mogen het datatype wel gebruiken in hun eigen CREATE TABLE-instructies, maar moeten daar wel expliciet toestemming voor hebben gekregen. Voor het verstrekken van dit recht is een speciale variant van de GRANT-instructie geïntroduceerd.

Voorbeeld 2.3: Verstrek JIM het recht het GELDBEDRAG-datatype te gebruiken.

```
GRANT    USAGE
ON       TYPE GELDBEDRAG
TO       JIM
```

Toelichting: USAGE is de nieuwe vorm. Nadat deze GRANT-instructie is uitgevoerd, mag JIM tabellen met kolommen definiëren die op dit nieuwe datatype zijn gebaseerd.

Opmerking: Door sommige producten wordt hiervoor niet het woord USAGE gebruikt maar, net als bij stored procedures, het woord EXECUTE. De betekenis en het effect zijn hetzelfde.

En uiteraard bestaat de tegenhanger van deze instructie ook:

```
REVOKE   USAGE
ON       TYPE GELDBEDRAG
TO       JIM
```

Wat gebeurt er echter als het recht wordt ingetrokken nadat JIM het datatype reeds in een CREATE TABLE-instructie gebruikt heeft? Het effect van deze instructie is ook weer productafhankelijk. Maar de meeste producten hanteren de volgende regel: het recht tot gebruik van een datatype kan alleen worden ingetrokken als die gebruiker het nog nergens gebruikt heeft.

2.4 Casting van waarden

In paragraaf 2.2 gaven we aan dat we met het toepassen van zelfgedefinieerde datatypes strong typing realiseren. Maar wat als we tóch appels met peren willen vergelijken? Om dit mogelijk te maken moeten we het datatype van de waarden veranderen. Hiervoor gebruiken we een expliciete vorm van casting.

Voor elk nieuw datatype worden er door SQL automatisch twee nieuwe scalaire functies voor casting gecreëerd. De ene functie transformeert waarden van het zelfgedefinieerde datatype naar waarden van het onderliggende basisdatatype (deze functie heeft de naam van het basisdatatype) en de andere werkt andersom (en heeft de naam van het zelfgedefinieerde datatype). Deze functies worden ook wel respectievelijk de *destructor* en de *constructor* genoemd. Voor het datatype GELDBEDRAG heet de destructor DECIMAL en constructor GELDBEDRAG. Let wel, in objectgeoriënteerde programmeertalen worden deze twee termen ook gebruikt voor het respectievelijk verwijderen en creëren van objecten.

Voorbeeld 2.4: Geef de betalingsnummers van de boetes waarvan het boetebedrag hoger is dan 50.

Hiervoor bestaan twee gelijkwaardige formuleringen:

```
SELECT   BETALINGSNR
FROM     BOETES
WHERE    BEDRAG > GELDBEDRAG(50)
```

en

```
SELECT   BETALINGSNR
FROM     BOETES
WHERE    DECIMAL(BEDRAG) > 50
```

Toelichting: In de eerste SELECT-instructie wordt de waarde 50 (wat een 'gewoon' getal is met waarschijnlijk het INTEGER-datatype) getransformeerd naar een geldbedrag. Hierna kan deze met vergelijkbare waarden in de BEDRAG-kolom vergeleken worden. De constructor GELDBEDRAG construeert dus geldbedragen uit numerieke waarden. De tweede instructie toont dat boetebedragen met behulp van de destructor genaamd DECIMAL naar 'gewone' getallen omgezet kunnen worden. Het resultaat van beide instructies is uiteraard hetzelfde.

Voorbeeld 2.5: Geef de betalingsnummers van de boetes waarvan het spelersnummer hoger is dan het boetebedrag.

```
SELECT   BETALINGSNR
FROM     BOETES
WHERE    INTEGER(SPELERSNR) > INTEGER(BETALINGSNR)
```

Toelichting: Omdat het SPELERSNR- en BETALINGSNR-datatype op hetzelfde basisdatatype gebaseerd zijn, namelijk INTEGER, hebben beide ook een destructor genaamd INTEGER. Met andere woorden, er zijn nu twee functies met dezelfde naam, maar deze werken op verschillende datatypes. Dit levert binnen SQL geen problemen op. SQL kan de twee functies uit elkaar houden, omdat de parameters van de twee functies qua datatype verschillend zijn. Dit concept waarbij verschillende functies dezelfde naam hebben, wordt *overladen* (overloading) genoemd. De functienaam INTEGER is in het bovenstaande voorbeeld overladen.

Opmerking: Het veranderen van het datatype van een waarde om deze te kunnen vergelijken met waarden met een ander datatype, wordt in de Engelstalige literatuur wel eens *semantic override* genoemd.

Casting van waarden is tevens essentieel bij het invoeren van nieuwe waarden met INSERT- en UPDATE-instructies. Nu drie kolommen in de BOETES-tabel een zelfgedefinieerd datatype kennen, kunnen we geen simpele numerieke waarden meer in deze kolom plaatsen. Ook bij de INSERT-instructie moeten we nu casting gebruiken.

Voorbeeld 2.6: Voeg een nieuwe boete toe.

```
INSERT INTO BOETES (BETALINGSNR, SPELERSNR, DATUM, BEDRAG)
VALUES             (BETALINGSNR(12), SPELERSNR(6), '1980-12-08', GELDBEDRAG(100.00))
```

2.5 Zelf definiëren van operatoren

Net als elke programmeertaal ondersteunt SQL operatoren als +, -, * en /. Een paar algemene opmerkingen over deze operatoren:

- Op zich zijn deze operatoren niet noodzakelijk. Voor een operator als + had ook de functie TEL-OP en voor * had de functie VERMENIGVULDIG gecreëerd kunnen worden. Omwille van het schrijfgemak zijn deze operatoren echter toegevoegd.
- Zoals vermeld hoort bij elk basisdatatype een aantal mogelijke operaties. Met de numerieke datatypes bijvoorbeeld, kunnen we operaties als optellen, vermenigvuldigen en aftrekken toepassen. Bij het datumdatatype is het mogelijk er een aantal maanden bij op te tellen, en dan ontstaat een nieuwe datum.

- In paragraaf 2.4 is het overladen van functies behandeld. Overladen van operatoren is ook mogelijk. Of we de + gebruiken bij twee getallen of twee alfanumerieke waarden geeft een heel ander effect. Afhankelijk van de datatypes van de waarden wordt er opgeteld of worden de alfanumerieke waarden aan elkaar geplakt.

Enkele SQL-producten staan toe dat operatoren voor zelfgedefinieerde datatypes gecreëerd worden. In principe zijn dit de operaties die voor het onderliggende datatype gelden, maar we kunnen zelf ook operaties definiëren. De SQL-producten staan alleen toe dat dit gebeurt in de vorm van scalaire functies.

Laten we verdergaan met het datatype GELDBEDRAG. Veronderstel dat er twee kolommen zijn, BEDRAG1 en BEDRAG2, die beide op dit datatype GELDBEDRAG zijn gedefinieerd en die we bij elkaar willen optellen. Omdat GELDBEDRAG geen normale numerieke waarde is, mogen we operatoren als + en – niet meer gebruiken. De volgende expressie zou niet meer toegestaan zijn:

```
BEDRAG1 + BEDRAG2
```

Nu moet dit met de volgende expressie gebeuren:

```
DECIMAL(BEDRAG1) + DECIMAL(BEDRAG2)
```

We kunnen dit eleganter oplossen door het symbool + ook te definiëren voor waarden van het GELDBEDRAG-datatype.

```
CREATE FUNCTION "+" (GELDBEDRAG, GELDBEDRAG)
   RETURNS GELDBEDRAG
   SOURCE "+" (DECIMAL(), DECIMAL())
```

Toelichting: De +-operator wordt nu opnieuw gedefinieerd en wederom is deze overladen. Nu is wel de expressie `BEDRAG1 + BEDRAG2` legaal.

Veronderstel dat het datatype KLEUR en de functie MIX (om twee kleuren te mixen) gedefinieerd zijn. Vervolgens kan bijvoorbeeld de +-operator worden gecreëerd als operator om kleuren te mixen.

```
CREATE FUNCTION "+" (KLEUR, KLEUR)
   RETURNS KLEUR
   SOURCE MIX (KLEUR, KLEUR)
```

Het zelf kunnen definiëren van operatoren verhoogt dus niet de functionaliteit van SQL, maar wel worden hiermee bepaalde instructies eenvoudiger om te formuleren.

2.6 Opaque-datatype

Een distinct-datatype is gebaseerd op één basisdatatype en erft daarmee alle eigenschappen van het basisdatatype. Enkele producten staan toe dat geheel nieuwe datatypes gedefinieerd worden, datatypes die niet afhankelijk zijn van een basisdatatype. Dit worden *opaque-datatypes* genoemd. Opaque staat voor niet-transparant. Men zou kunnen stellen dat een opaque-datatype een zelfgedefinieerd basisdatatype is.

Opaque-datatypes zijn nodig wanneer het te complex is om ze met behulp van een basisdatatype te definiëren. Bijvoorbeeld, als we het datatype 2D-coördinaat willen definiëren, zullen we hoe dan ook twee getallen moeten opslaan: de X- en de Y-coördinaat. Als we alleen met basisdatatypes werken, dan lukt dat niet. Met opaque-datatypes kan dit wel, getuige het volgende voorbeeld.

Voorbeeld 2.7: Creëer het datatype TWEEDIM voor het opslaan van tweedimensionale coördinaten.

```
CREATE TYPE TWEEDIM
      (INTERNALLENGTH = 4)
```

Toelichting: Wat het meest opvalt bij deze CREATE TYPE-instructie is dat inderdaad *geen* basisdatatype wordt gespecificeerd. Het enige dat er opgegeven wordt is hoeveel ruimte één waarde van dit type op schijf inneemt, namelijk vier bytes. Vier bytes is gekozen, omdat we er voor het gemak van uitgaan dat een coördinaat uit twee gehele getallen bestaat.

Echter, voordat we dit nieuwe datatype in een CREATE TABLE-instructie kunnen gebruiken, zullen we een aantal functies moeten definiëren. We moeten bijvoorbeeld een functie creëren die een waarde, die wordt ingetikt door de gebruiker, omzet naar iets wat op harde schijf wordt weggezet en een functie die andersom werkt. Voor basisdatatypes hoeven we dat niet te doen. Als we het CHAR-datatype gebruiken, gaan we ervan uit dat deze functies reeds bestaan. Nu moeten we ze zelf creëren. Hier zullen we niet op ingaan, omdat de wijze waarop sterk productafhankelijk is. Naast die verplichte functies mogen ook andere functies gedefinieerd worden om de functionaliteit te verhogen. In de meeste gevallen zullen dit externe functies zijn.

2.7 Named row-datatype

Het derde zelfgedefinieerde datatype is het *named row-datatype*. Hiermee kunnen we waarden die logisch bij elkaar horen tot een eenheid groeperen. Bijvoorbeeld, we kunnen alle waarden die behoren tot een adres samenvoegen.

Voorbeeld 2.8: Creëer het named row-datatype genaamd ADRES en gebruik het vervolgens in een CREATE TABLE-instructie.

```
CREATE TYPE ADRES AS (
   STRAAT      CHAR(15) NOT NULL,
   HUISNR      CHAR(4),
   POSTCODE    CHAR(6),
   PLAATS      CHAR(10) NOT NULL)

CREATE TABLE SPELERS (
   SPELERSNR   INTEGER PRIMARY KEY,
   NAAM        CHAR(15),
   :           :
   WOONADRES   ADRES,
   TELEFOON    CHAR(13),
   BONDSNR     CHAR(4))
```

Toelichting: In plaats van dat er nu vier kolommen in de CREATE TABLE-instructie moeten worden gedefinieerd, volstaat een, namelijk WOONADRES. Dit betekent dat in één rij in de kolom WOONADRES niet één waarde maar één rij met vier waarden wordt opgeslagen. Deze rij van vier waarden heeft een naam (of met andere woorden, is benoemd), namelijk ADRES. Vandaar de term named row. De kolom WOONADRES wordt een *samengestelde kolom* genoemd. Bij de kolommen van een named row-datatype mag een NOT NULL-specificatie opgenomen worden.

Uiteraard mag een datatype meerdere keren in een en dezelfde CREATE TABLE-instructie gebruikt worden. Voorbeeld:

```
CREATE TABLE SPELERS (
   SPELERSNR       INTEGER PRIMARY KEY,
   :               :
   WOONADRES       ADRES,
   POSTADRES       ADRES,
   VAKANTIEADRES   ADRES,
   TELEFOON        CHAR(13),
   BONDSNR         CHAR(4))
```

Het werken met samengestelde kolommen heeft invloed op de formuleringen van SELECT- en andere instructies. We illustreren dit met enkele voorbeelden.

Voorbeeld 2.9: Geef de nummers en de volledige adressen van de spelers die in Den Haag wonen.

```
SELECT   SPELERSNR, WOONADRES
FROM     SPELERS
WHERE    WOONADRES.PLAATS = 'Den Haag'
```

Resultaat:

```
SPELERSNR  <=============== WOONADRES ============>
           STRAAT        HUISNR  POSTCODE  PLAATS
---------  ------------  ------  --------  --------
        6  Hazensteinln  80      1234KK    Den Haag
       83  Mariakade     16A     1812UP    Den Haag
        2  Steden        43      3575NH    Den Haag
        7  Erasmusweg    39      9758VB    Den Haag
       57  Erasmusweg    16      4377CB    Den Haag
       39  Ericaplein    78      9629CD    Den Haag
      100  Hazensteinln  80      1234KK    Den Haag
```

Toelichting: In de SELECT-component hoeft nu maar één kolom in plaats van vier te worden gespecificeerd. Uiteraard zal het resultaat wel uit vijf kolommen bestaan. De notatie `WOONADRES.PLAATS` is nieuw. Met deze *puntnotatie* wordt aangegeven dat slechts met een deel van het adres gewerkt moet worden.

Voorbeeld 2.10: Geef de nummers van de spelers die op hetzelfde adres wonen als speler 6.

```
SELECT   ANDEREN.SPELERSNR
FROM     SPELERS AS S6, SPELERS AS ANDEREN
WHERE    S6.WOONADRES = ANDEREN.WOONADRES
AND      S6.SPELERSNR = 6
```

Toelichting: In plaats van een join-conditie op vier kolommen (STRAAT, HUISNR, PLAATS en POSTCODE) volstaat nu één simpele join-conditie waarbij de samengestelde kolom wordt gebruikt.

Casting van waarden is ook bij named row-datatypes van belang. We geven een voorbeeld van een SELECT- en een INSERT-instructie.

Voorbeeld 2.11: Geef het nummer en de naam van de speler die woont op het adres Erasmusweg 39, Den Haag met postcode 9758VB.

```
SELECT   SPELERSNR, NAAM
FROM     SPELERS
WHERE    WOONADRES =
         ADRES('Erasmusweg', 39, '9758VB', 'Den Haag')
```

Toelichting: In dit voorbeeld is duidelijk te zien hoe de vier waarden tot één ADRES-waarde worden gecast, zodat ze vergeleken kunnen worden met de kolom WOONADRES.

Voorbeeld 2.12: Voer een nieuwe speler in.

```
INSERT INTO SPELERS
       (SPELERSNR, NAAM, ..., ADRES, TELEFOON, BONDSNR)
VALUES (6, 'Permentier', ...,
        ADRES('Hazensteinln', 80, '1234KK', 'Den Haag'),
        '070-476537', 8467)
```

Named row-datatypes worden meestal op basis- en distinct-datatypes gedefinieerd, maar ze mogen ook 'gestapeld' worden. Hieronder staat een voorbeeld. Hier wordt eerst het datatype POSTCODE gedefinieerd bestaande

uit twee componenten: een viercijferig en een tweeletterig deel. Vervolgens wordt dit nieuwe named row-datatype bij de definitie van het ADRES-datatype gebruikt.

```
CREATE TYPE POSTCODE AS (
   CIJFERS    CHAR(4),
   LETTERS    CHAR(2))

CREATE TYPE ADRES AS (
   STRAAT     CHAR(15) NOT NULL,
   HUISNR     CHAR(4),
   POSTCODE   POSTCODE,
   PLAATS     CHAR(10) NOT NULL)
```

Voorbeeld 2.13: Geef de nummers en de volledige adressen van de spelers die in postcodegebied 2501 wonen.

```
SELECT   SPELERSNR, WOONADRES
FROM     SPELERS
WHERE    WOONADRES.POSTCODE.CIJFERS = '2501'
```

Voorbeeld 2.14: Geef de nummers en de volledige adressen van de spelers met postcode 1234KK.

```
SELECT   SPELERSNR, WOONADRES
FROM     SPELERS
WHERE    WOONADRES.POSTCODE = POSTCODE('1234', 'KK')
```

Toelichting: In de conditie worden twee waarden samengevoegd tot een waarde met een POSTCODE-datatype. Hiervoor wordt een casting-functie gebruikt.

Behalve het named row-datatype kennen enkele SQL-producten tevens het *unnamed row-datatype*. Bij dit datatype worden waarden ook tot een groep samengevoegd, maar deze groep krijgt geen aparte naam; zie het volgende voorbeeld.

Voorbeeld 2.15: Creëer een tabel met een unnamed row-datatype.

```
CREATE TABLE SPELERS
   SPELERSNR   INTEGER PRIMARY KEY,
   NAAM        CHAR(15),
   :           :
   WOONADRES   ROW (STRAAT    CHAR(15) NOT NULL,
                    HUISNR    CHAR(4),
                    POSTCODE  CHAR(6),
                    PLAATS    CHAR(10) NOT NULL),
   TELEFOON    CHAR(13),
   BONDSNR     CHAR(4))
```

Toelichting: Ook hier worden vier waarden samengevoegd tot één waarde. Echter, er wordt niet expliciet een datatype voor gedefinieerd. Het effect van een unnamed row-datatype op SELECT- en andere instructies is gelijk aan dat van een named row-datatype. Het verschil is echter dat de specificatie niet op meerdere plaatsen herbruikt kan worden. Als deze tabel ook een POSTADRES-kolom zou bevatten, zouden de vier deelkolommen daar opnieuw gedefinieerd moeten worden.

Voor casting van waarden wordt het woord ROW gebruikt:

```
INSERT INTO SPELERS
       (SPELERSNR, NAAM, ..., ADRES, TELEFOON, BONDSNR)
VALUES (6, 'Permentier', ...,
        ROW('Hazensteinln', 80, '1234KK', 'Den Haag'),
        '070-476537', 8467)
```

2.8 De getypeerde tabel

Tot zover is het named row-datatype alleen gebruikt voor het specificeren van kolommen. Dit datatype kan echter ook gebruikt worden om een datatype aan een tabel toe te kennen. Het effect is dat dan niet meer expliciet de kolommen en hun datatypes gespecificeerd moeten worden, maar dat de kolommen van het named row-datatype de kolommen van de tabel vormen.

Voorbeeld 2.16: Creëer een type voor de SPELERS-tabel.

```
CREATE TYPE T_SPELERS AS (
   SPELERSNR     INTEGER NOT NULL,
   NAAM          CHAR(15) NOT NULL,
   VOORLETTERS   CHAR(3) NOT NULL,
   GEB_DATUM     DATE,
   GESLACHT      CHAR(1) NOT NULL,
   JAARTOE       SMALLINT NOT NULL,
   STRAAT        CHAR(15) NOT NULL,
   HUISNR        CHAR(4),
   POSTCODE      CHAR(6),
   PLAATS        CHAR(10) NOT NULL,
   TELEFOON      CHAR(13),
   BONDSNR       CHAR(4))

CREATE TABLE SPELERS OF T_SPELERS (
   PRIMARY KEY SPELERSNR)
```

Toelichting: Met de specificatie `OF T_SPELERS` in de CREATE TABLE-instructie geven we aan dat de SPELERS-tabel alle kolommen krijgt van het datatype T_SPELERSNR. Uiteraard moeten er nog wel bepaalde integriteitsregels worden gespecificeerd, vandaar de specificatie van de primaire sleutel. Alleen de NOT NULL-integriteitsregel kan binnen de CREATE TYPE-instructie worden opgegeven. Een tabel die op deze wijze gedefinieerd wordt, wordt een *getypeerde tabel* genoemd. SELECT en mutatie-instructies zijn voor getypeerde en niet-getypeerde tabellen gelijk.

Het voordeel van getypeerde tabellen is dat tabellen met dezelfde structuur op zeer simpele wijze gedefinieerd kunnen worden. Stel dat er nog een SPELERS-tabel is met spelers die vroeger lid waren van de tennisvereniging. Deze tabel heeft waarschijnlijk dezelfde kolommen, dus het creëren ervan is nu eenvoudig:

```
CREATE TABLE OUDE_SPELERS OF T_SPELERS (
   PRIMARY KEY SPELERSNR)
```

2.9 Integriteitsregels op datatypes

Enkele SQL-producten staan toe dat bij een datatype ook integriteitsregels gespecificeerd worden. Deze integriteitsregels beperken dan de toegestane waarden van het datatype en daarmee de populaties van de kolommen die op dat datatype zijn gedefinieerd.

Voorbeeld 2.17: Definieer het datatype AANTAL_SETS en specificeer dat alleen de waarden 1, 2 of 3 legaal zijn.

```
CREATE TYPE AANTAL_SETS AS SMALLINT
   CHECK (VALUE IN (0, 1, 2, 3))

CREATE TABLE WEDSTRIJDEN (
   WEDSTRIJDNR  INTEGER PRIMARY KEY,
   TEAMNR       INTEGER NOT NULL,
   SPELERSNR    INTEGER NOT NULL,
   GEWONNEN     AANTAL_SETS NOT NULL,
   VERLOREN     AANTAL_SETS NOT NULL)
```

Toelichting: Bij de CREATE TYPE-instructie wordt een check-integriteitsregel opgenomen. Hiermee wordt met behulp van een conditie aangegeven wat legale waarden zijn. Waarden zijn legaal als ze aan de conditie voldoen. Het gereserveerde woord VALUE staat voor een mogelijke waarde van dat desbetreffende datatype. Elke simpele conditie mag hier gebruikt worden. Dat wil zeggen dat vergelijkingsoperatoren AND, OR, NOT, BETWEEN, IN, LIKE en IS NULL allemaal gebruikt mogen worden. Subquery's zijn echter niet toegestaan.

Het voordeel is nu dat als de integriteitsregel voor AANTAL_SETS verandert, dit slechts op één plek gewijzigd moet worden.

Voorbeeld 2.18: Wijzig het datatype AANTAL_SETS zodanig dat nu ook de waarde 4 is toegestaan.

```
ALTER TYPE AANTAL_SETS AS SMALLINT
   CHECK (VALUE BETWEEN 0 AND 4)
```

Bij het wijzigen van de conditie kan een probleem ontstaan als we de conditie 'strakker' definiëren. Stel dat AANTAL_SETS gedefinieerd wordt als alleen de waarden 0, 1 en 2. Wat gebeurt er dan als de kolommen gedefinieerd op dit datatype reeds een waarde hebben die buiten dit bereik valt? De producten lossen dit op door een dergelijke verandering van het datatype niet toe te staan. Eerst moeten de kolommen dan gecorrigeerd worden.

2.10 Sleutels en indexen

Primaire sleutels, refererende sleutels en indexen mogen op kolommen met zelfgedefinieerde datatypes gecreeerd worden. Bij de named row-datatypes kunnen ze op de volledige waarde of op een deel ervan gedefinieerd worden.

Voorbeeld 2.19: Definieer een index op de kolom WOONADRES in de SPELERS-tabel.

```
CREATE INDEX I_WOONADRES
   ON SPELERS(WOONADRES)
```

Voorbeeld 2.20: Definieer een index op alleen het POSTCODE-deel van de kolom WOONADRES in de SPELERS-tabel.

```
CREATE INDEX I_WOONADRES
   ON SPELERS(WOONADRES.POSTCODE)
```

De enige speciale situatie is wanneer indexen op opaque-datatypes gedefinieerd moeten worden. Daar verschillen de mogelijkheden per product zeer sterk.

Hoofdstuk 3

Overerving, references en collecties

3.1 Overerving van datatypes

In paragraaf 1.2 staat vermeld dat niet van alle objectrelationele concepten wordt gezegd dat ze objectgeoriënteerd van aard zijn. De concepten die we in dit hoofdstuk behandelen zijn dat wel. De concepten die aan bod komen zijn overerving, references ofwel rij-identificaties en collecties.

Het belangrijkste OO-concept is *overerving* (inheritance). Voor de meeste specialisten is dit tevens het meest aansprekende begrip. Bij overerving van datatypes erft het ene datatype alle eigenschappen van een ander datatype en kan het er zelf nog een paar extra hebben. Met eigenschappen bedoelen we bijvoorbeeld de kolommen waaruit het datatype bestaat of de functies die op het datatype gedefinieerd zijn.

Voorbeeld 3.1: Definieer de named row-datatypes ADRES en BUITENLANDS_ADRES.

```
CREATE TYPE ADRES AS (
   STRAAT     CHAR(15) NOT NULL,
   HUISNR     CHAR(4),
   POSTCODE   POSTCODE,
   PLAATS     CHAR(10) NOT NULL)

CREATE TYPE BUITENLANDS_ADRES AS (
   LAND       CHAR(20) NOT NULL) UNDER ADRES
```

Toelichting: Het datatype ADRES bestaat uit vier kolommen, het datatype BUITENLANDS_ADRES uit vijf. Het datatype BUITENLANDS_ADRES is nu een zogenaamd *sub*type van ADRES ofwel ADRES is een *super*type van BUITENLANDS_ADRES. Elk buitenlands adres is in principe een adres, maar niet alle adressen zijn buitenlandse adressen.

Vervolgens definiëren we een tabel waarbij we het subtype gebruiken:

```
CREATE TABLE SPELERS (
   SPELERSNR       INTEGER PRIMARY KEY,
   :               :
   WOONADRES       ADRES,
   VAKANTIEADRES   BUITENLANDS_ADRES,
   TELEFOON        CHAR(13),
   BONDSNR         CHAR(4))
```

Met het volgende voorbeeld tonen we het effect van het werken met subtypes op de SELECT-instructie.

Voorbeeld 3.2: Geef het spelersnummer, de plaats en het land van het vakantieadres van elke speler wiens woonplaats begint met de hoofdletter *J* en van wie het cijferdeel van de postcode van het vakantieadres onbekend is.

```
SELECT   SPELERSNR, VAKANTIEADRES.PLAATS, VAKANTIEADRES.LAND
FROM     SPELERS
WHERE    WOONADRES.PLAATS LIKE 'J%'
AND      VAKANTIEADRES.POSTCODE.CIJFERS IS NULL
```

Toelichting: In de SELECT-component wordt de PLAATS-kolom van het VAKANTIEADRES gevraagd. Deze kolom staat niet expliciet in het BUITENLANDS_ADRES-datatype gedefinieerd. SQL realiseert zich bij zo'n vraag dat als de gevraagde kolom er niet is, er bij een supertype gekeken moet worden. In dit geval is dat ADRES en deze heeft wel een kolom genaamd PLAATS. Tevens wordt de LAND-kolom opgevraagd. De WHERE-component bevat twee condities. De eerste heeft een bekende vorm en bij de tweede wordt eerst gevraagd naar de POSTCODE-waarde van het VAKANTIEADRES (geërfd van het supertype ADRES) en vervolgens wordt het cijferdeel van de postcode opgevraagd.

Functies die gedefinieerd zijn voor een bepaald datatype, kunnen ook gebruikt worden voor het bewerken van waarden van een subtype van dat datatype. Veronderstel dat de functie POPULATIE als uitvoerparameter een geheel getal heeft dat het aantal inwoners van die stad voorstelt. De invoerparameter is een waarde van het datatype ADRES. Omdat BUITENLANDS_ADRES alles erft van het datatype ADRES, mogen we dus ook een buitenlands adres als invoer gebruiken.

Voorbeeld 3.3: Geef het spelersnummer van elke speler die op vakantie is in een stad met een populatie groter dan één miljoen.

```
SELECT   SPELERSNR
FROM     SPELERS
WHERE    POPULATIE(VAKANTIEADRES) > 1000000
```

3.2 Koppelen van tabellen via rij-identificaties

In OO-databases hebben alle (equivalenten van) rijen een unieke identificatie. Deze identificatie wordt niet door de gebruiker met bijvoorbeeld een INSERT-instructie opgevoerd, maar wordt door het systeem zelf gegenereerd. Deze identificaties worden veelal *rij-identificaties*, *object-identifiers* of *surrogate keys* genoemd. Deze unieke rij-identificaties kunnen gebruikt worden om rijen met elkaar te koppelen en rijen naar elkaar te laten verwijzen.

Dit idee is ook binnen SQL overgenomen. Ook hier wordt aan elke rij een unieke identificatie toegekend. De rij-identificaties hebben geen waarde die voor de gebruikers zinvol is, maar wel voor SQL zelf. Ze zijn op te vragen en af te drukken, maar dragen geen informatie. Als een rij een identificatie heeft ontvangen, blijft die voorgoed bij die rij horen. Als de rij wordt verwijderd, zal de bijbehorende identificatie nooit meer herbruikt worden. Let wel, unieke rij-identificaties zijn niet hetzelfde als primaire sleutels (al hebben ze er wel wat van weg). We komen later in dit hoofdstuk terug op wat de verschillen precies zijn.

Rij-identificaties worden samen met de rij opgeslagen, maar hiervoor hoeven niet apart kolommen te worden gedefinieerd. Deze kolommen worden automatisch gegenereerd. Men zou kunnen stellen dat elke tabel een verborgen kolom heeft waarin deze rij-identificaties opgeslagen worden.

De rij-identificatie (ofwel de waarde van de verborgen kolom) is op te vragen met de *REF-functie*.

Voorbeeld 3.4: Geef van de speler met nummer 6 de rij-identificatie.

```
SELECT   REF(SPELERS)
FROM     SPELERS
WHERE    SPELERSNR = 6
```

Resultaat:

```
REF(SPELERS)
--------------------------------------------------
000028020915A58C5FAEC1502EE034080009D0DADE15538856
```

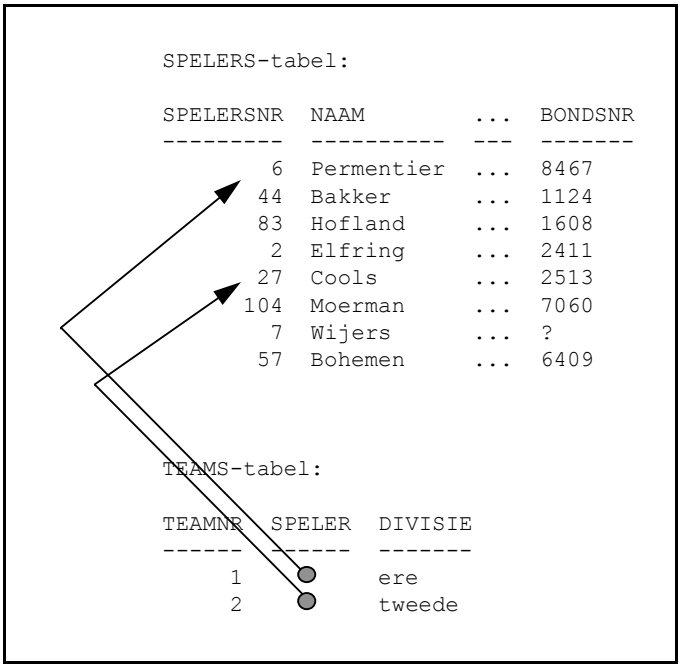
Toelichting: De *REF-functie* heeft als parameter de naam van een tabel en retourneert de rij-identificatie. Hoe rij-identificaties er werkelijk (op schijf) uitzien, is productafhankelijk. Als voorbeeld is een mogelijke rij-identificatie van Oracle weergegeven.

Zoals vermeld, rij-identificaties kunnen gebruikt worden om rijen te 'koppelen.' De identificatie van de ene rij wordt dan opgeslagen binnen een andere rij. Met andere woorden, de ene rij refereert of wijst dan naar de andere.

Voorbeeld 3.5: Definieer de tabellen van de tennisvereniging opnieuw, maar nu gebruikmakend van rij-identificaties.

```
CREATE TABLE SPELERS (
   SPELERSNR     INTEGER PRIMARY KEY,
   NAAM          CHAR(15) NOT NULL,
   :             :
   BONDSNR       CHAR(4))

CREATE TABLE TEAMS (
   TEAMNR        INTEGER PRIMARY KEY,
   SPELER        REF(SPELERS) NOT NULL,
   DIVISIE       CHAR(6) NOT NULL)

CREATE TABLE WEDSTRIJDEN (
   WEDSTRIJDNR   INTEGER PRIMARY KEY,
   TEAM          REF(TEAMS) NOT NULL,
   SPELER        REF(SPELERS) NOT NULL,
   GEWONNEN      SMALLINT NOT NULL,
   VERLOREN      SMALLINT NOT NULL)

CREATE TABLE BOETES (
   BETALINGSNR   INTEGER PRIMARY KEY,
   SPELER        REF(SPELERS) NOT NULL,
   DATUM         DATE NOT NULL,
   BEDRAG        DECIMAL(7,2) NOT NULL)

CREATE TABLE BESTUURSLEDEN (
   SPELER        REF(SPELERS) PRIMARY KEY,
   BEGIN_DATUM   DATE NOT NULL,
   EIND_DATUM    DATE,
   FUNCTIE       CHAR(20))
```

Toelichting: Overal waar een refererende sleutel stond, staat nu een kolom die wijst naar een andere tabel. Dit worden *reference-kolommen* genoemd. De koppeling die nu tussen bijvoorbeeld de SPELERS- en de TEAMS-tabel is ontstaan, zouden we kunnen weergeven als in figuur 3.1



Figuur 3.1 *Reference-kolommen*

Reference-kolommen moeten gevuld worden met rij-identificaties. Hiervoor zijn de INSERT- en UPDATE-instructie aangepast.

Voorbeeld 3.6: Voeg een nieuw team toe. De aanvoerder van dit team is speler 112.

```
INSERT   INTO TEAMS (TEAMNR, SPELER, DIVISIE)
VALUES   (3, (SELECT REF(SPELERS)
              FROM   SPELERS
              WHERE  SPELERSNR = 112), 'ere')
```

Toelichting: Met de SELECT-instructie wordt de rij-identificatie van speler 6 opgehaald en vervolgens opgeslagen in de SPELER-kolom.

Voorbeeld 3.7: De aanvoerder van team 1 is niet meer speler 6, maar 44.

```
UPDATE   TEAMS
SET      SPELER = (SELECT REF(SPELERS)
                   FROM   SPELERS
                   WHERE  SPELERSNR = 44)
WHERE    TEAMNR = 1
```

Het koppelen van tabellen met rij-identificaties heeft tevens grote invloed op de wijze waarop joins geformuleerd kunnen worden. De meeste worden veel eenvoudiger om te formuleren.

Voorbeeld 3.8: Geef van elk team het teamnummer en de naam van de aanvoerder.

```
SELECT   TEAMNR, SPELER.NAAM
FROM     TEAMS
```

Toelichting: Van elke rij in de TEAMS-tabel worden twee waarden getoond: de waarde van de TEAMNR-kolom en de waarde van de expressie SPELER.NAAM. Dit is een expressie die nog niet behandeld is, maar waar we nu aandacht aan zullen besteden.

Voor het gemak noemen we dit een *reference-expressie*. Een reference-expressie begint altijd met een reference-kolom. In dit geval is dat de kolom SPELER. Deze kolom wijst naar de SPELERS-tabel. Achter de reference-kolom staat de NAAM-kolom uit die tabel. Het uiteindelijke effect is dat de naam van de speler, die de aanvoerder van het team is, wordt getoond.

Het lijkt alsof er met deze instructie helemaal geen join wordt uitgevoerd, maar dat is schijn. De join zit verstopt in de reference-expressie SPELER.NAAM. SQL zoekt voor elk team de rij-identificatie op van de speler (de aanvoerder). Deze identificatie is opgeslagen in de kolom SPELER. Vervolgens wordt in de verborgen kolom van de SPELERS-tabel gezocht naar de rij met deze identificatie. Als deze gevonden is, wordt de waarde in de NAAM-kolom opgepakt. Met andere woorden, bij deze instructie wordt inderdaad geen klassieke join gespecificeerd. De bovenstaande SELECT-instructie wordt achter de schermen vertaald naar de volgende:

```
SELECT   TEAMS.TEAMNR, SPELERS.NAAM
FROM     TEAMS, SPELERS
WHERE    TEAMS.SPELER = REF(SPELERS)
```

Eerst wordt aan de FROM-component de tabel toegevoegd waar de SPELER-kolom naar wijst. Vervolgens wordt een join-conditie aan de instructie toegevoegd. In deze join-conditie wordt TEAMS.SPELER vergeleken met de REF van de tabel waar de kolom naar toe wijst.

Er zijn twee zaken waar we apart rekening mee moeten houden. Ten eerste, een reference-kolom mag uiteraard een null-waarde bevatten. In dat geval wordt er geen join met de andere tabel uitgevoerd. De waarde van de reference-expressie is dan ook null. Ten tweede, de reference-kolom kan een rij-identificatie bevatten die niet in de andere tabel voorkomt. Veronderstel dat de rij-identificatie in de SPELER-kolom van de TEAMS-tabel niet in de SPELERS-tabel voorkomt. In dat geval zou dat team bij het uitvoeren van een inner-join niet in het resultaat voorkomen. Bij dit soort expressies wordt echter altijd een outer-join uitgevoerd. In feite wordt dus niet de bovenstaande, maar de volgende instructie uitgevoerd:

```
SELECT   TEAMS.TEAMNR, SPELERS.NAAM
FROM     TEAMS LEFT OUTER JOIN SPELERS
         ON (TEAMS.SPELER = REF(SPELERS))
```

Voorbeeld 3.9: Geef van elke wedstrijd gespeeld door iemand uit Zoetermeer en voor een team uit de eredivisie het wedstrijdnummer, de naam van de speler en de naam van de aanvoerder van het team.

```
SELECT   WEDSTRIJDNR, SPELER.NAAM, TEAM.SPELER.NAAM
FROM     WEDSTRIJDEN
WHERE    SPELER.PLAATS = 'Zoetermeer'
AND      TEAM.DIVISIE = 'ere'
```

Toelichting: De instructie bevat drie reference-expressies: SPELER.NAAM, TEAM.SPELER.NAAM en SPELER.PLAATS. Nummers een en drie zijn bekende vormen: een reference-kolom gevolgd door een 'gewone' kolom. De tweede expressie is echter een nieuwe vorm. Hier wordt de reference-kolom TEAM gevolgd door een andere reference-kolom, namelijk SPELER. En daar staat NAAM achter. Deze expressie moet gelezen worden als: geef van de desbetreffende rij de NAAM van de SPELER die de aanvoerder is van het TEAM. Deze reference-expressie vervangt als het ware een join-specificatie van de WEDSTRIJDEN-tabel met eerst die van TEAMS en vervolgens met SPELERS.

Er bestaat geen beperking wat betreft de lengte van reference-expressies. De enige beperking is dat de laatste kolom geen reference-kolom mag zijn.

Voorbeeld 3.10: Creëer twee tabellen met werknemer- en afdelingsgegevens.

```
CREATE TABLE WERKNEMERS (
   WERKNEMERNR   INTEGER PRIMARY KEY,
   NAAM          CHAR(15) NOT NULL,
   AFDELING      REF(AFDELINGEN))

CREATE TABLE AFDELINGEN (
   AFDELINGNR    INTEGER PRIMARY KEY,
   NAAM          CHAR(15) NOT NULL,
   BAAS          REF(WERKNEMERS))
```

De volgende instructie is nu geldig:

```
SELECT   AFDELINGNR, BAAS.AFDELING.BAAS.NAAM
FROM     AFDELINGEN
```

Toelichting: Van elke afdeling wordt gevraagd naar de naam van de baas van de afdeling waar de baas van de afdeling werkt.

Reference-kolommen mogen ook naar de tabel wijzen waar ze onderdeel van zijn.

Voorbeeld 3.11: Creëer de SPELERS-tabel met de nieuwe kolommen VADER en MOEDER. Deze twee kolommen worden gebruikt als de vader en/of moeder ook lid van de tennisvereniging zijn.

```
CREATE TABLE SPELERS (
   SPELERSNR   INTEGER PRIMARY KEY,
   NAAM        CHAR(15) NOT NULL,
   VADER       REF(SPELERS),
   MOEDER      REF(SPELERS),
   :           :
   BONDSNR     CHAR(4))
```

Voorbeeld 3.12: Geef het spelersnummer en de naam van de vader van elke speler wiens moeder ook bij de tennisvereniging speelt.

```
SELECT   SPELERSNR, VADER.NAAM
FROM     SPELERS
WHERE    MOEDER IS NOT NULL
```

Voorbeeld 3.13: Geef het spelersnummer van elke speler wiens opa ook bij de tennisvereniging speelt.

```
SELECT   SPELERSNR
FROM     SPELERS
WHERE    MOEDER.VADER IS NOT NULL
OR       VADER.VADER IS NOT NULL
```

Het werken met references heeft de volgende voordelen:

- **Voordeel 1:** Er kunnen geen fouten gemaakt worden bij het toekennen van een datatype aan een refererende sleutel. Het datatype van een refererende sleutel moet altijd gelijk zijn aan dat van de primaire sleutel. Dit kan nu niet fout gaan, omdat bij een reference-kolom alleen de tabelnaam wordt gespecificeerd.

- **Voordeel 2:** Sommige primaire sleutels zijn wat betreft het aantal kolommen en/of aantal bytes zeer breed. Het effect is dat de refererende sleutels (die er naar verwijzen) ook breed zullen zijn en dus veel opslagruimte in beslag zullen nemen. Bij het werken met reference-kolommen wordt alleen de rij-identificatie opgeslagen. Deze zou kleiner kunnen zijn en dat bespaart opslagruimte.

- **Voordeel 3:** Primaire sleutels mogen gewijzigd worden. Als dit gebeurt, moeten ook de refererende sleutels aangepast worden. Dit vertraagt uiteraard de mutatie. Bij references speelt dit niet, omdat ten eerste rij-identificaties (de verborgen kolommen) niet gewijzigd kunnen worden en ten tweede er daardoor nooit extra wijzigingen op de andere tabellen zullen plaatsvinden. Het wijzigen van één waarde in de primaire sleutel blijft dus altijd beperkt tot het wijzigen van die ene waarde.

- **Voordeel 4:** Bepaalde SELECT-instructies worden eenvoudiger om te formuleren; zie voorbeelden 3.8 en 3.9.

Het werken met references kent echter ook een aantal nadelen:

- **Nadeel 1:** Bepaalde mutatie-instructies worden wat lastiger om te formuleren; zie voorbeelden 3.6 en 3.7.

- **Nadeel 2:** De reference biedt wat betreft het koppelen van tabellen slechts eenrichtingsverkeer. Het is nu eenvoudig om van wedstrijden gegevens over spelers op te vragen, maar niet andersom. We illustreren dit met een voorbeeld.

  Voorbeeld 3.14: Geef van elke speler het spelersnummer en de nummers van zijn of haar wedstrijden.

  ```
  SELECT   S.SPELERSNR, W.WEDSTRIJDNR
  FROM     SPELERS AS S, WEDSTRIJDEN AS W
  WHERE    REF(S) = W.SPELER
  ```

- **Nadeel 3:** Het ontwerpen van databases wordt nu ook lastiger. Was er eerst slechts één methode om relaties tussen twee tabellen te definiëren, nu zijn dat er twee. De vraag wordt dan welke van de twee moeten we wanneer gebruiken? En moeten we overal dezelfde methode gebruiken, of moeten we dat van de situatie laten afhangen? Gebruiken we niet overal dezelfde methode, dan zullen de gebruikers ook bij het formuleren van hun SQL-instructies goed moeten opletten. Bij database-ontwerp geldt altijd: hoe meer keuzes hoe moeilijker het wordt.

- **Nadeel 4:** Een reference-kolom is niet hetzelfde als een refererende sleutel. De populatie van een refererende sleutel is altijd een deelverzameling van die van een primaire sleutel. Dit geldt echter niet voor reference-kolommen. Als een bijvoorbeeld speler uit de SPELERS-tabel wordt verwijderd, dan worden niet in de andere tabellen waar die rij-identificaties voorkomen, al deze meeverwijderd. Er ontstaan dan in de WEDSTRIJDEN-tabel zogenaamde *dangling references*. Reference-kolommen kunnen dus niet de integriteit van gegevens bewaken zoals refererende sleutels dat kunnen.

3.3 Collecties

Tot nu zijn we er in dit boek van uitgegaan dat een kolom voor elke rij slechts één waarde bevat. We introduceren hier een nieuwe term: *cel*. Een cel is de kruising van een kolom en een rij. Dus tot nu toe zijn we ervan uitgegaan dat een cel maar één waarde mag bevatten. Uiteraard kunnen we wel meerdere waarden in een cel opslaan. We kunnen bijvoorbeeld een volledig adres, bestaande uit een straatnaam, huisnummer, postcode, enzovoorts, gescheiden door komma's in een cel opslaan. Wij zullen deze waarde dan interpreteren als bestaande uit meerdere waarden. SQL daarentegen zal deze waarde nog steeds als één atomaire waarde zien. En daar gaat het om.

Met de adoptie van OO-concepten binnen SQL, gaat dit veranderen. We kunnen nu verzamelingen met waarden in een cel opslaan. SQL zal deze verzameling echt als een verzameling en niet als één atomaire waarde beschouwen. Een dergelijke verzameling wordt een *collectie* genoemd. Met een collectie kunnen we bijvoorbeeld voor één speler een willekeurig aantal telefoonnummers in de kolom TELEFOONS registreren.

Voorbeeld 3.15: Definieer de SPELERS-tabel zodanig dat een verzameling van telefoonnummers kan worden opgeslagen.

```
CREATE TABLE SPELERS (
   SPELERSNR   INTEGER PRIMARY KEY,
   :           :
   TELEFOONS   SETOF(CHAR(13)),
   BONDSNR     CHAR(4))
```

Toelichting: Met de term SETOF wordt aangegeven dat binnen de kolom TELEFOONS een verzameling waarden kan worden opgeslagen. De tabel zelf zou er dan als volgt kunnen uitzien (net als bij de verzamelingenleer worden accolades gebruikt om een verzameling aan te geven). Duidelijk is dat enkele spelers twee en sommigen zelfs drie telefoonnummers hebben.

```
SPELERSNR ... PLAATS     TELEFOONS                                BONDSNR
--------- --- ---------- ---------------------------------------- -------
        6 ... Den Haag   {070-476537, 070-478888}                 8467
       44 ... Rijswijk   {070-368753}                             1124
       83 ... Den Haag   {070-353548, 070-235634, 079-344757}     1608
        2 ... Den Haag   {070-237893, 020-753756}                 2411
       27 ... Zoetermeer {079-234857}                             2513
      104 ... Zoetermeer {079-987571}                             7060
        7 ... Den Haag   {070-347689}                             ?
       57 ... Den Haag   {070-473458}                             6409
       39 ... Den Haag   {070-393435}                             ?
      112 ... Rotterdam  {010-548745, 010-256756, 015-357347}     1319
        8 ... Rijswijk   {070-458458}                             2983
      100 ... Den Haag   {070-494593}                             6524
       28 ... Leiden     {071-659599}                             ?
       95 ... Voorburg   {070-867564, 055-358458}                 ?
```

Uiteraard heeft het gebruik van collecties invloed op andere SQL-instructies. We geven eerst enkele voorbeelden van hoe gegevens in deze speciale kolom ingevoerd kunnen worden en vervolgens voorbeelden van hoe ze met de SELECT-instructie geraadpleegd kunnen worden.

Voorbeeld 3.16: Voeg een nieuwe speler toe met twee telefoonnummers.

```
INSERT   INTO SPELERS (SPELERSNR, ... , TELEFOONS, ...)
VALUES   (213, ..., {'071-475748', '071-198937'}, ...)
```

Toelichting: Met accolades wordt de verzameling met telefoonnummers gespecificeerd. Binnen de accolades mogen nul, een of meer waarden opgenomen worden. Nul is interessant als deze speler geen telefoon heeft.

Voorbeeld 3.17: Geef speler 44 een nieuw telefoonnummer.

```
UPDATE   SPELERS
SET      TELEFOONS = {'070-658347'}
WHERE    SPELERSNR = 44
```

Voorbeeld 3.18: Geef de nummers van de spelers die bereikbaar zijn onder telefoonnummer 070-476537.

```
SELECT   SPELERSNR
FROM     SPELERS
WHERE    '070-476537' IN (TELEFOONS)
```

Resultaat:

```
SPELERSNR
---------
        6
```

Toelichting: In deze SELECT-instructie is een nieuwe vorm van de IN-operator toegepast. Normaliter wordt achter de IN-operator een lijst met constanten of expressies gespecificeerd of een subquery. Bij beide vormen stelt datgene wat tussen haakjes staat een verzameling met waarden voor. Voor deze nieuwe vorm geldt hetzelfde, want de kolom TELEFOONS stelt ook een verzameling waarden voor. Deze vorm van de IN-operator mag alleen gebruikt worden voor collecties, niet voor andere kolommen.

Voorbeeld 3.19: Geef de nummers van de spelers met meer dan twee telefoonnummers.

```
SELECT   SPELERSNR
FROM     SPELERS
WHERE    CARDINALITY(TELEFOONS) > 2
```

Resultaat:

```
SPELERSNR
---------
       83
      112
```

Toelichting: Voor het bepalen van het aantal waarden in een collectie kan de *CARDINALITY-functie* worden gebruikt. Bij het bepalen van het aantal waarden worden null-waarden niet meegerekend en tellen dubbele waarden als een.

De instructie had ook als volgt gedefinieerd kunnen worden.

```
SELECT   SPELERSNR
FROM     SPELERS
WHERE    2 < (SELECT COUNT(*) FROM TABLE(SPELERS.TELEFOONS))
```

Toelichting: De instructie lijkt bijna een normale instructie, behalve dat de FROM-component in de subquery een nieuwe constructie bevat: TABLE(TELEFOONS). Deze constructie verandert de verzameling in een tabel bestaande uit één kolom met een aantal rijen. Het aantal rijen is uiteraard gelijk aan het aantal waarden in de collectie. En voor elke speler is er een andere tabel.

De reden waarom deze complexere oplossing is toegevoegd, is omdat deze meer mogelijkheden geeft dan die met de CARDINALITY-functie.

Voorbeeld 3.20: Geef de nummers van de spelers met het grootste aantal telefoonnummers.

```
SELECT   SPELERSNR
FROM     SPELERS
WHERE    CARDINALITY(TELEFOONS) >= ALL
        (SELECT   CARDINALITY(TELEFOONS)
         FROM     SPELERS)
```

Resultaat:

```
SPELERSNR
---------
       83
      112
```

Voorbeeld 3.21: Geef de nummers van de spelers die dezelfde verzameling telefoonnummers hebben als speler 6.

```
SELECT   SPELERSNR
FROM     SPELERS
WHERE    TELEFOONS =
        (SELECT   TELEFOONS
         FROM     SPELERS
         WHERE    SPELERSNR = 6)
```

Toelichting: De instructie spreekt voor zichzelf. In plaats van de vergelijkingsoperator = mogen ook > en < worden gebruikt. De vergelijkingsoperator > zou in dit geval betekenen: wie heeft minimaal dezelfde telefoonnummers als speler 6? Maar deze persoon mag er meer hebben. De vergelijkingsoperator < betekent: wie heeft minimaal één telefoonnummer van speler 6?

Voorbeeld 3.22: Geef een oplopend gesorteerde lijst met alle telefoonnummers uit de SPELERS-tabel.

Helaas is deze vraag niet zo simpel als hij lijkt. De volgende instructie is bijvoorbeeld niet goed. De kolom TELEFOONS geeft namelijk niet één verzameling waarden die vervolgens te sorteren is, maar een verzameling bestaande uit verzamelingen.

```
SELECT   TELEFOONS
FROM     SPELERS
ORDER BY TELEFOONS
```

We moeten deze kolom eerst, zoals dat heet, 'platslaan.'

```
SELECT   TS.TELEFOONS
FROM     THE (SELECT TELEFOONS
              FROM   SPELERS) AS TS
ORDER BY TS.TELEFOONS
```

Toelichting: Een FROM-component kan een subquery bevatten. Hier maken we nu weer gebruik van, echter we plaatsen het woord THE er voor. Het effect hiervan is dat het resultaat van de subquery, dat uit een verzameling met verzamelingen bestaat, wordt getransformeerd naar een verzameling bestaande uit atomaire waarden. De verzameling wordt dus platgeslagen.

Het resultaat van de subquery kan als volgt worden voorgesteld:

```
TELEFOONS
-----------------------------------
{070-476537, 070-478888}
{070-368753}
{070-353548, 070-235634, 079-344757}
{070-237893, 020-753756}
{079-234857}
{079-987571}
{070-347689}
{070-473458}
{070-393435}
{010-548745, 010-256756, 015-357347}
{070-458458}
{070-494593}
{071-659599}
{070-867564, 055-358458}
```

Het resultaat na de THE-operator ziet er als volgt uit:

```
TELEFOONS
----------
070-476537
070-478888
070-368753
070-353548
070-235634
079-344757
070-237893
020-753756
079-234857
079-987571
070-347689
070-473458
070-393435
010-548745
010-256756
015-357347
070-458458
070-494593
071-659599
070-867564
055-358458
```

Nu is het weer een 'gewone' tabel bestaande uit één kolom met een verzameling waarden. Deze resultaattabel wordt in de FROM-component TS genoemd. In de SELECT-instructie vragen we dan naar deze kolom en in de ORDER BY-component worden de waarden gesorteerd.

Het platslaan van deze collecties geeft diverse mogelijkheden.

Voorbeeld 3.23: Geef het aantal telefoonnummers van spelers 6 en 44 bij elkaar.

```
SELECT   COUNT(DISTINCT TS.TELEFOONS)
FROM     THE (SELECT TELEFOONS
              FROM   SPELERS
              WHERE  SPELERSNR IN (6, 44)) AS TS
```

Toelichting: De subquery zelf heeft als resultaat twee verzamelingen met telefoonnummers: een voor speler 6 en een voor speler 44. De THE-operator slaat de twee verzamelingen plat tot één verzameling rijen elk bestaande uit één atomaire waarde. Met deze operator worden niet-dubbele waarden automatisch verwijderd. Vandaar dat we in de COUNT-functie DISTINCT gebruiken.

Voorbeeld 3.24: Geef de telefoonnummers die spelers 6 en 44 gemeenschappelijk hebben.

```
SELECT   TS1.TELEFOONS
FROM     THE (SELECT TELEFOONS
              FROM   SPELERS
              WHERE  SPELERSNR = 6) TS1
INTERSECT
SELECT   TS2.TELEFOONS
FROM     THE (SELECT TELEFOONS
              FROM   SPELERS
              WHERE  SPELERSNR = 44) TS2
```

In de bovenstaande voorbeelden is de collectie gedefinieerd op een kolom met een basisdatatype. Zelfgedefinieerde datatypes mogen ook gebruikt worden. Andersom kunnen zelfgedefinieerde datatypes ook gebruikmaken van collecties. We geven van beide een voorbeeld.

Voorbeeld 3.25: Definieer de SPELERS-tabel zodanig dat een verzameling van telefoonnummers kan worden opgeslagen, maar waarbij het TELEFOON-datatype wordt gebruikt.

```
CREATE   TYPE TELEFOON AS
        (NETNR          CHAR(3),
         ABONNEENR      CHAR(6))

CREATE TABLE SPELERS
   SPELERSNR   INTEGER PRIMARY KEY,
   :           :
   TELEFOONS   SETOF(TELEFOON),
   BONDSNR     CHAR(4))
```

Voorbeeld 3.26: Definieer de SPELERS-tabel zodanig dat een verzameling van telefoonnummers kan worden opgeslagen, maar waarbij de verzameling waarden binnen het TELEFOONS-datatype wordt gedefinieerd.

```
CREATE TYPE TELEFOONS AS (
   TELEFOON    SETOF(CHAR(13))

CREATE TABLE SPELERS (
   SPELERSNR   INTEGER PRIMARY KEY,
   :           :
   TELEFOONS   TELEFOONS,
   BONDSNR     CHAR(4))
```

3.4 Overerving van tabellen

In de eerste paragraaf van dit hoofdstuk is overerving van datatypes uitvoerig behandeld. Deze paragraaf bespreekt *overerving van tabellen*. Wordt overerving van datatypes algemeen gezien als zeer nuttig, overerving van tabellen is een enigszins omstreden onderwerp.

Om uit te leggen hoe dit principe werkt, introduceren we de volgende twee named row-datatypes. Het tweede, OUDE_SPELERS, is een subtype van het eerste en heeft een kolom extra. Deze VERTROKKEN-kolom geeft met een datum aan wanneer iemand de vereniging heeft verlaten.

```
CREATE   TYPE T_SPELERS AS
        (SPELERSNR      INTEGER NOT NULL,
         NAAM           CHAR(15) NOT NULL,
         VOORLETTERS    CHAR(3) NOT NULL,
         GEB_DATUM      DATE,
         GESLACHT       CHAR(1)  NOT NULL,
         JAARTOE        SMALLINT NOT NULL,
         STRAAT         CHAR(15) NOT NULL,
         HUISNR         CHAR(4),
         POSTCODE       CHAR(6),
         PLAATS         CHAR(10) NOT NULL,
         TELEFOON       CHAR(13),
         BONDSNR        CHAR(4))

CREATE   TYPE T_OUDE_SPELERS AS
        (VERTROKKEN     DATE NOT NULL) UNDER T_SPELERS
```

Nadat deze twee datatypes zijn gecreëerd, kunnen we er twee tabellen voor definiëren:

```
CREATE TABLE SPELERS OF T_SPELERS (
   PRIMARY KEY SPELERSNR)

CREATE TABLE OUDE_SPELERS OF T_OUDE_SPELERS UNDER SPELERS
```

Toelichting: In paragraaf 2.8 hebben we reeds gezien hoe getypeerde tabellen gecreëerd worden. Wat nieuw is aan de bovenstaande constructie, is dat de OUDE_SPELERS-tabel als UNDER SPELERS gedefinieerd wordt. Hiermee wordt het een *subtabel* van SPELERS; met andere woorden, wordt SPELERS een *supertabel* van OUDE_SPELERS. Door dit op deze manier te definiëren, erft OUDE_SPELERS alles van de SPELERS-tabel, dus ook de primaire sleutel.

Supertabellen kunnen meerdere subtabellen hebben en subtabellen mogen zelf ook subtabellen hebben. Een verzameling tabellen gekoppeld als sub- en supertabellen wordt een *tabelhiërarchie* genoemd. SQL kent enkele beperkingen voor een tabelhiërarchie:

- Een tabel mag niet direct of indirect een subtabel of supertabel van zichzelf zijn. Veronderstel dat OUDE_SPELERS een subtabel heeft genaamd STOKOUDE_SPELERS. We kunnen nu niet SPELERS als subtabel van STOKOUDE_SPELERS definiëren. De tabelhiërarchie zou dan een cyclische structuur bevatten en dat is niet toegestaan.
- Een subtabel kan maar één directe supertabel hebben. Dit wordt enkelvoudige overerving genoemd. Meervoudige overerving (multiple inheritance) is niet toegestaan.
- De SQL-producten die momenteel overerving van tabellen ondersteunen, staan alleen getypeerde-tabellen in de tabelhiërarchie toe.
- De tabelhiërarchie moet overeenkomen met de typehiërarchie. Dit betekent dat als het type T_OUDE_SPELERS niet gedefinieerd was als subtype van T_SPELERS, we de twee bovenstaande CREATE TABLE-instructies niet hadden mogen invoeren.

Het werken met tabelovererving heeft invloed op de SELECT-instructie. We illustreren dit met enkele voorbeelden. Hierbij veronderstellen we dat de SPELERS-tabel slechts vier rijen bevat: spelers 6, 44, 83 en 2 en dat de OUDE_SPELERS-tabel drie extra spelers bevat: 211, 260 en 280.

Voorbeeld 3.27: Geef de gehele SPELERS-tabel.

```
SELECT   *
FROM     SPELERS
```

Toelichting: Deze instructie geeft alle spelers uit de SPELERS-tabel en uit alle onderliggende subtabellen. Het resultaat bevat dus de spelers 6, 44, 83, 2, 211, 260 en 280. Echter, alleen de kolommen van de SPELERS-tabel worden getoond.

Voorbeeld 3.28: Geef alle oude spelers.

```
SELECT   *
FROM     OUDE_SPELERS
```

Toelichting: Deze instructie geeft alle oude spelers uit de OUDE_SPELERS-tabel, dit zijn spelers 211, 260 en 280. Het is uiteraard een deelverzameling van het resultaat van de vorige instructie, omdat niet alle spelers oud zijn. Van elke oude speler worden alle kolommen van de OUDE_SPELERS-tabel getoond, dus ook de VERTROKKEN-kolom.

Voorbeeld 3.29: Geef alle kolommen van alle spelers, maar niet die welke in OUDE_SPELERS-tabel voorkomen. Dus geef alleen de jonge spelers.

We kunnen deze vraag oplossen met behulp van de EXCEPT-operator:

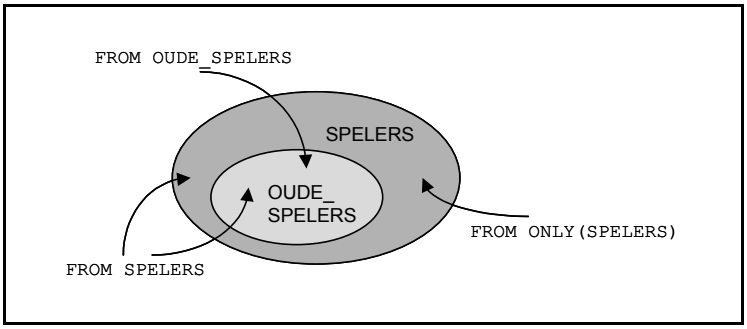
```
SELECT   *
FROM     SPELERS
EXCEPT
SELECT   *
FROM     OUDE_SPELERS
```

Voor deze vraag is echter een speciale constructie toegevoegd:

```
SELECT   *
FROM     ONLY(SPELERS)
```

Toelichting: Deze instructie geeft *alleen* spelers uit de SPELERS-tabel die niet in de subtabellen voorkomen: 6, 44, 83 en 2.

In figuur 3.2 is grafisch weergegeven wat de verschillen tussen de drie FROM-componenten zijn.



Figuur 3.2 *Welke FROM-component geeft welk resultaat?*

Voor INSERT-instructies gelden geen speciale regels wat betreft het werken met super- en subtabellen. Maar veel van de opmerkingen en regels die gelden voor de SELECT-instructie gelden ook voor de UPDATE en DELETE-instructies.

Voorbeeld 3.30: Wijzig het jaar van toetreding in 1980 voor alle spelers geboren vóór 1980.

```
UPDATE   SPELERS
SET      JAARTOE = 1980
WHERE    GEB_DATUM < '1980-01-01'
```

Toelichting: Deze mutatie wijzigt het jaar van toetreding van alle spelers, inclusief de oude spelers, geboren voor 1980. Als alleen de jonge spelers gewijzigd moeten worden, moeten we weer ONLY gebruiken:

```
UPDATE   ONLY(SPELERS)
SET      JAARTOE = 1980
WHERE    GEB_DATUM < '1980-01-01'
```

Voorbeeld 3.31: Verwijder alle spelers geboren voor 1980.

```
DELETE
FROM     SPELERS
WHERE    GEB_DATUM < '1980-01-01'
```

Toelichting: Met deze DELETE-instructie worden ook oude spelers geboren voor 1980 verwijderd. Als alleen de jonge spelers gewijzigd moeten worden, moet wederom in de FROM-component ONLY gespecificeerd worden.

De Auteur

Rick F. van der Lans is auteur van vele boeken over SQL. Naast dit SQL Leerboek dat in diverse talen vertaald is, waaronder Engels, Duits, Chinees enItaliaans, heeft hij SQL boeken geschreven voor producten als MySQL, Oracle, SQLite, Ingres en Pervasive PSQL.

Hij is onafhankelijk adviesur, auteur en docent gespecialiseerd in databasetechnologie, datawarehousing en applicatie-integratie. Hij is oprichter en directeur van R20/Consultancy. Door de jaren heen heeft hij veel organisaties geadviseerd op het gebied van IT-architecturen.

Als spreker op conferenties en seminars wordt hij internationaal gerespecteerd. Al meer dan vijfentwintig jaar geeft hij over de gehele wereld lezingen, inclusief in de meeste Europese landen, Noord- en Zuid-Amerika en Australië. Hij is voorzitter van het jaarlijkse European Data Warehouse and Business Intelligence Conference. Hij schrijft een column voor Database Magazine en voor het internationale Beye-Network.com. Zeven jaar lang was hij lid van de Nederlandse ISO commissie verantwoordelijk voor ISO SQL Standaard.

Rick kan via de volgende kanalen bereikt worden:

Email: rick@r20.nl
Twitter: http://twitter.com/Rick_vanderlans
LinkedIn: http://www.linkedin.com/pub/rick-van-der-lans/9/207/223

Cursussen over de volgende onderwerpen kunnen door Rick F. van der Lans verzorgd worden

- Database-ontwerp en informatiemodellering
- De basis van SQL
- Het ontwikkelen van geavanceerde SQL queries
- Datawarehousing en business intelligence
- Data virtualisatie

Andere boeken geschreven door Rick F. van der Lans