



# Leerboek Oracle SQL

voor Oracle database 11g/12c

Technische redactie: Henk van der Velde



Veenendaal  
University Press

# ➤ Leerboek Oracle SQL

*voor Oracle database 11g/12c*

TOON KOPPELAARS, LEX DE HAAN



Vierde druk



# ➤ Leerboek Oracle SQL

voor Oracle database 11g/12c

TOON KOPPELAARS, LEX DE HAAN



Vierde druk

2



## **Leerboek Oracle SQL**

### **Oracle Database 11g**

Vierde druk

Lex de Haan

Toon Koppelaars

3

Meer informatie over deze en andere uitgaven kunt u verkrijgen bij: Sdu  
Klantenservice

Postbus 20014

2500 EA Den Haag

tel.: (070) 378 98 80

[www.sdu.nl/service](http://www.sdu.nl/service)

© 2013 Sdu Uitgevers bv, Den Haag

Academic Service is een imprint van Sdu Uitgevers bv

1e druk 1993

2e druk 1998

3e druk 2004

4e druk 2013

Zetwerk: Redactiebureau Ron Heijer, Markelo

Omslagontwerp: Studio Bassa, Culemborg

Omslaguitvoering: Carlito's Design, Amsterdam

ISBN: 978 90 395 2681 1 (paperback)

ISBN: 978 90 395 27719 (Bookshelf e-book)

NUR: 123

Alle rechten voorbehouden. Alle auteursrechten en databankrechten ten aanzien van deze uitgave worden uitdrukkelijk voorbehouden. Deze rechten berusten bij Sdu Uitgevers bv.

Behoudens de in of krachtens de Auteurswet gestelde uitzonderingen, mag niets uit deze uitgave worden verveelvoudigd, opgeslagen in een geautomatiseerd gegevensbestand of openbaar gemaakt in enige vorm of op enige wijze, hetzij elektronisch, mechanisch, door fotokopieën, opnemen of enige andere manier, zonder voorafgaande schriftelijke toestemming van de uitgever.

Voorzover het maken van reprografische verveelvoudigingen uit deze uitgave is toegestaan op grond van artikel 16 h Auteurswet, dient men de daarvoor wettelijk verschuldigde vergoedingen te voldoen aan de Stichting Reprorecht (postbus 3051, 2130 KB [Hoofddorp](http://Hoofddorp), [www.reprorecht.nl](http://www.reprorecht.nl)). Voor het overnemen van gedeelte(n) uit deze uitgave in bloemlezingen, readers en andere compilatiewerken (artikel 16

4

Auteurswet) dient men zich te wenden tot de Stichting PRO (Stichting Publicatie- en Reproductierechten Organisatie, Postbus 3060, 2130 KB Hoofddorp,

[www.cedar.nl/pro](http://www.cedar.nl/pro)). Voor het overnemen van een gedeelte van deze uitgave ten behoeve van commerciële doeleinden dient men zich te wenden tot de uitgever.

Hoewel aan de totstandkoming van deze uitgave de uiterste zorg is besteed, kan voor de afwezigheid van eventuele (druk)fouten en onvolledigheden niet worden ingestaan en aanvaarden de auteur(s), redacteur(en) en uitgever deswege geen aansprakelijkheid voor de gevolgen van eventueel voorkomende fouten en onvolledigheden.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the publisher's prior consent.

While every effort has been made to ensure the reliability of the information presented in this publication, Sdu Uitgevers neither guarantees the accuracy of the data contained here in nor accepts responsibility for errors or omissions or their consequences.

5

## **Voorwoord**

Dit leerboek Oracle SQL is in eerste instantie bedoeld voor het onderwijs op HBO-niveau en alle studierichtingen die Oracle als softwareomgeving gebruiken bij het leren omgaan met databases. Het is in het bijzonder geschikt als ondersteuning bij zelfstudie en/of practicum, en als zodanig ook heel goed individueel te gebruiken buiten het reguliere onderwijs.

Gezien de praktische opzet van het boek is de beschikking over een Oracle-omgeving beslist noodzakelijk; alle daarvoor benodigde software is gratis te downloaden vanaf het internet. U heeft hiervoor twee alternatieven:

1

Download Oracle Express Edition.

Oracle Express Edition kan gedownload worden vanaf

<http://www.oracle.com/technetwork/products/express->

[edition/downloads/index.html](#) (of google: “oracle xe download”). U kunt kiezen voor de Windows- of de Linux-versie. Nadat u de software gedownload heeft, dient u deze nog te installeren via de Oracle Installer.

2

Download pre-built Oracle Developer VM.

Dit is waarschijnlijk de eenvoudigere optie, aangezien u hier een virtual machine (VM) downloadt die al kant-en-klaar geïnstalleerd is. Van de pre-built VM's zijn alleen Linux-versies beschikbaar. Ze kunnen gedownload worden vanaf

[http://www.oracle.com/technetwork/community/developer-](#)  
[vm/index.html](#) (of google “oracle Pre-Built Developer VMs”). Kiest u daar voor de “Database App Development VM”. Deze VM heeft naast de database (met SQL\*Plus) ook een SQL Developer-installatie in zich.

Dit leerboek is gebaseerd op de volgende Oracle-softwareversie: Oracle Database 11g voor Windows of Linux, versie 11.2.0.3

Hoewel dit boek uitgaat van Oracle Database 11g is het ook heel goed te gebruiken met Oracle 10g en zelfs ook met de aankomende release 6

- 
- 

Oracle 12c. Het kan echter voorkomen dat bepaalde syntaxconstructies in de 10g versie van Oracle nog niet worden ondersteund; dit kunnen we eventueel controleren aan de hand van de Oracle 11g documentatie. De SQL Reference heeft een sectie ‘Oracle Database 11g New Features in the SQL Reference’ aan het einde van de inleiding, voorafgaand aan

[hoofdstuk 1.](#)

Daar waar mogelijk houden we de officiële ANSI/ISO-standaard (SQL:2003) aan; alleen in het geval van nuttige Oracle-specifieke SQL-uitbreidingen

wijken we van deze standaard af. Daardoor zal het grootste deel van de SQL-voorbeelden in dit boek ook op andere DBMS-implementaties werken.

Overigens bevat de eerdergenoemde SQL Reference een appendix B, met als titel ‘Oracle and Standard SQL’, waarin de verschillen tussen de ANSI/ISO-SQL-standaard en de Oracle SQL-implementatie worden belicht.

De mogelijkheden van SQL en SQL\*Plus worden zo veel mogelijk uitgelegd aan de hand van concrete commandovoорbeelden. De voorbeelden worden geïllustreerd met schermafbeeldingen of weergegeven in een kader. In deze voorbeeldkaders wordt de tekst die we zelf dienen in te voeren vet weergegeven, in tegenstelling tot de uitvoer. Vooral de hoofdzaken komen aan de orde, terwijl bijzaken of technische details zo veel mogelijk buiten beschouwing blijven.

Dit leerboek streeft beslist niet naar volledigheid; daarvoor is de taal SQL te omvangrijk en de Oracle-omgeving te complex. De SQL

Reference beslaat tegenwoordig meer dan 1800 pagina's, terwijl zelfs de SQL Quick Reference al een behoorlijke omvang heeft met 170

pagina's. Om over de omvang van de ANSI/ISO SQL-standaard nog maar te zwijgen. De belangrijkste uitgangspunten van dit leerboek zijn nog steeds de combinatie handzaamheid en gunstige prijsstelling. De documentatie die bij de Oracle-software wordt geleverd biedt eventueel gewenste detailinformatie; de gehele Oracle-documentatielibrary is online beschikbaar op de website van Oracle:

<http://www.oracle.com/technetwork/documentation/index.html#database>

Bevat een overzicht van documentatie voor de meest gangbare Oracle releases. Op deze pagina vindt u de volgende url;

<http://www.oracle.com/pls/db112/homepage> Bevat de hele documentatie van Oracle 11g. Op deze pagina vindt u de volgende url;

[http://docs.oracle.com/cd/E11882\\_01/server.112/e26088/toc.htm](http://docs.oracle.com/cd/E11882_01/server.112/e26088/toc.htm)

Bevat de SQL Language Reference van Oracle 11g.

De nadruk ligt in dit boek op het raadplegen met behulp van SQL; datadefinitie en datamanipulatie komen in mindere mate aan bod.

Beveiliging, autorisatie, en databasebeheer worden slechts volledigheidshalve genoemd in het totaaloverzicht van SQL in

hoofdstuk 2, maar verder niet behandeld.

Er wordt gebruik gemaakt van een casus bestaande uit zeven tabellen.

Hierin zijn gegevens opgeslagen over medewerkers, afdelingen, en cursussen. Het aantal rijen in deze tabellen is bewust klein gehouden: daardoor kunnen resultaten van SQL-commando's handmatig worden gecontroleerd, wat prettig is als we ons nog niet helemaal zeker voelen van de taal SQL. In een realistisch informatiesysteem is dat meestal onmogelijk, gezien de hoeveelheid gegevens in dergelijke systemen. Het gaat in dit leerboek echter niet om het volume, maar meer om de complexiteit van de databasesstructuur en de correctheid van SQL-commando's.

In het eerste hoofdstuk wordt een beknopte inleiding gegeven over de achtergronden van informatiesystemen en het bijbehorende databasejargon. Daarna volgt een globaal overzicht van de Oracle-software en een bespreking van de casus.

In hoofdstuk 2 volgt een globaal overzicht van de taal SQL, en een eerste kennismaking met SQL\*Plus en SQL Developer als omgevingen om SQL interactief in uit te voeren. In hoofdstuk 11 komen we weer op SQL\*Plus terug; dan worden enkele geavanceerde mogelijkheden behandeld (bijvoorbeeld: werken met variabelen, scripts en rapportage).

Datadefinitie komt in twee etappes aan de orde, namelijk in de hoofdstukken 3 en 7. Voor deze opzet is gekozen om zo snel mogelijk met raadpleging te kunnen beginnen. In hoofdstuk 3 komen daarom alleen de meest

noodzakelijke begrippen aan bod (tabellen, datatypes, en de datadictionary).

Ook raadpleging is verspreid over diverse hoofdstukken; vier, om precies te zijn. [In hoofdstuk 4 krijgen](#) vooral de select-, where- en order by-component onze aandacht. De belangrijkste functies komen aan de orde [in hoofdstuk 5](#). [Ook null-waarden](#) en subqueries worden in dat 8

hoofdstuk behandeld. [In hoofdstuk 8 gaan we](#) meerdere tabellen tegelijk benaderen (joinen) en queryresultaten aggregeren: dat wil dus zeggen dat dan de from-, group by- en having-componenten centraal staan.

[Hoofdstuk 9 gaat](#) verder in op subqueries en het gebruik van vensters en analytische functies, hiërarchische queries, en flashback queries. Alle vier deze hoofdstukken over raadpleging worden afgesloten met opgaven, waarvan de antwoorden op de website van de uitgever

[\(\[www.academicservice.nl\]\(http://www.academicservice.nl\)\)](http://www.academicservice.nl) te vinden zijn.

[In hoofdstuk 6 komt](#) datamanipulatie aan de orde. We behandelen de commando's insert, update, en delete. Ook wordt enige aandacht besteed aan hiermee samenhangende zaken: transactieverwerking, read-consistency, en locking.

[In hoofdstuk 7 komen](#) we terug op datadefinitie, om met name dieper in te gaan op constraints, indexen, sequences, en performance. Ook synoniemen komen in dit hoofdstuk aan bod.

Zoals eerder aangegeven gaan we in de hoofdstukken 8 en 9 verder in op raadpleging met SQL.

[Hoofdstuk 10 behandelt](#) views. Wat zijn views, wat zijn de belangrijkste toepassingsmogelijkheden van views, en wat zijn de beperkingen? We gaan in op datamanipulatie via views, views en performance, en materialized views.

[Hoofdstuk 11 is een vervolg](#) op hoofdstuk 2.

Oracle is een object-relationele database. Daartoe zijn aan de taal SQL allerlei voorzieningen toegevoegd. Om een indruk te geven van deze

mogelijkheden lichten we [in hoofdstuk 12 een tipje](#) van de sluier op; in het bijzonder zelfgedefinieerde datatypes, arrays en geneste tabellen komen aan de orde.

Als bijlage is in het boek een beschrijving opgenomen van de structuur en inhoud van de gebruikte casustabellen.

## Aanvullend materiaal

Bij het boek is aanvullend materiaal beschikbaar, waaronder drie bijlagen (quick reference SQL en SQL\*Plus, Data Dictionary overzicht, antwoorden van de opgaven), en scripts. Dit materiaal is te vinden op de 9

- 
- 
- 
- 
- 
- 
- 
- 
- 

pagina bij dit [boek op www.academicservice.nl](#)

## Bij de vierde herziene druk

Begin 2012 werd ik benaderd door de weduwe van Lex de Haan met de vraag of ik geïnteresseerd was om Lex' succesvolle titel 'Leerboek Oracle SQL' te herzien. Lex als auteur was geen vreemde voor mij aangezien wij samen 'Applied Mathematics for Database Professionals'

(Apress, 2007) hebben geschreven. Mijn verwachting was dan ook dat slechts een algehele update nodig zou zijn om de tekst weer in overeenstemming te brengen met de huidige stand van de techniek; er is het nodige veranderd sinds de derde druk van dit boek in 2004

verscheen. En dit is achteraf ook het geval gebleken. De opzet van het boek is vrijwel geheel gehandhaafd. Hierdoor is een soepele overgang naar deze nieuwe editie mogelijk. De indeling van de hoofdstukken is niet veranderd.

Wel zijn in alle hoofdstukken op diverse plaatsen tekstuile aanpassingen doorgevoerd om de hierboven vermelde overeenstemming te verkrijgen. Hieronder geven we nog kort enkele andere noemenswaardige aanpassingen.

[Hoofdstuk 1](#): behandeling van inmiddels verouderde tools en technieken is vervangen door die van hun opvolgers.

[Hoofdstuk 2](#): iSQLPlus is vervangen door SQL Developer.

[Hoofdstuk 5](#): enkele nieuwe REGEXP-functies zijn toegevoegd.

[Hoofdstuk 6](#): vermelding van INSERT-ALL-commando toegevoegd.

[Hoofdstuk 7: paragraaf 7.4 \(C\)onstraints](#) is herschreven om de diverse concepten helderder over te brengen.

[Hoofdstuk 9: paragraaf 9.5 \(Hiërachische Queries\)](#) is geheel vernieuwd en nu gebaseerd op de ANSI standaard Recursive

Subquery Factoring.

[Hoofdstuk 11](#): iSQLPlus is vervangen door SQL Developer.

Reacties op dit boek zijn van harte welkom en kunnen naar de uitgever worden gezonden of via e-mail aan de (co-)auteur.

Januari 2013

Toon Koppelaars

[\(toon.koppelaars@rulegen.com\)](mailto:(toon.koppelaars@rulegen.com)) 10

## **Inhoud**

[Voorwoord](#)

1

[Inleiding relationele databasesystemen en Oracle](#)

[1.1 Informatiebehoefte en informatiesystemen](#)

[1.2 Databaseontwerp](#)

[1.3 Database-managementsysteem](#)

[1.4 Relationale databases](#)

[1.5 Relationale gegevensstructuur](#)

[1.6 Relationale operatoren](#)

[1.7 Hoe relationeel is mijn DBMS?](#)

[1.8 De Oracle-software](#)

[1.9 De casus](#)

2

[Kennismaking met SQL, SQL\\*Plus en SQL Developer](#)

[2.1 Overzicht SQL](#)

[2.1.1](#)

[Datadefinitie](#)

[2.1.2](#)

[Datamanipulatie](#)

[2.1.3 R](#)

[aadpleging](#)

[2.1.4 B](#)

[eveilicing](#)

2.2 Enkele basisbegrippen

2.3 Kennismaking met SQL\*Plus

2.3.1

De SQL-buffer

2.3.2

Het gebruik van een externe editor

2.3.3

De SQL\*Plus-editor

2.3.4 C

ommando's bewaren

2.3.5 S

QL\*Plus-instellingen

2.3.6

Nog een paar nuttige SQL\*Plus-commando's

2.4 Kennismaking met SQL Developer

3

Datadefinitie – deel I

3.1 Schema's en gebruikers

3.2 Tabellen maken

3.3 Datatypes

### 3.4 De casustabellen

11

### 3.5 De datadictionary

4

## Raadpleging – de basis

### 4.1 Overzicht van de SELECT-componenten

#### 4.2 De SELECT-component

#### 4.3 De WHERE-component

#### 4.4 De ORDER BY-component

#### 4.5 AND, OR, NOT

#### 4.6 BETWEEN, IN, LIKE

#### 4.7 CASE-expressies

#### 4.8 NULL-waarden

#### 4.9 Subqueries

#### 4.10 Opgaven

5

## Raadpleging – functies

### 5.1 Inleiding

### 5.2 Rekenfuncties

### 5.3 Tekstfuncties

[5.4 Reguliere expressies](#)

[5.5 Datumfuncties](#)

[5.6 Algemene functies](#)

[5.7 Conversiefuncties](#)

[5.8 Opgeslagen functies](#)

[5.9 Opgaven](#)

6

[Datamanipulatie](#)

[6.1 Het INSERT-commando](#)

[6.2 Het UPDATE-commando](#)

[6.3 Het DELETE-commando](#)

[6.4 Transactieverwerking](#)

[6.5 Read consistency en locking](#)

7

[Datadefinitie – deel II](#)

[7.1 CREATE TABLE](#)

[7.2 Datatypes](#)

[7.3 ALTER TABLE](#)

[7.4 Constraints](#)

[7.5 Indexen](#)

[7.6 Performance](#)

[7.7 Sequences](#)

12

[7.8 Synoniemen](#)

[7.9 DROP TABLE](#)

[7.10 Overige commando's](#)

[7.11 Opgaven](#)

8

[Raadpleging – meerdere tabellen en aggregatie](#)

[8.1 Tuple-variabelen](#)

[8.2 Joins](#)

[8.3 De ANSI/ISO standaard join syntax](#)

[8.4 De outerjoin](#)

[8.5 De GROUP BY-component](#)

[8.6 Groepsfuncties](#)

[8.7 De HAVING-component](#)

[8.8 Extra mogelijkheden van de GROUP BY-component](#)

[8.9 Verzamelingsoperatoren](#)

[8.10 Opgaven](#)

9

## Raadpleging – enkele geavanceerde mogelijkheden

### 9.1 Subqueries: vervolg

### 9.2 Subqueries in de SELECT-component

### 9.3 Subqueries in de from-component

### 9.4 De WITH-component

### 9.5 Hiërarchische queries

### 9.6 Vensters en analytische functies

### 9.7 Flashback queries

### 9.8 Opgaven

## 10 Views

### 10.1 Wat zijn views?

### 10.2 Toepassingsmogelijkheden

### 10.3 Datamanipulatie via views

### 10.4 De CHECK OPTION

### 10.5 Datamanipulatie via inline views

### 10.6 Views en performance

### 10.7 Materialized views

### 10.8 Opgaven

## 11 SQL\*Plus en SQL Developer

### 11.1 SQL\*Plus versus SQL Developer

## 11.2 SQL\*Plus-variabelen

13

### 11.2.1 S

#### ubstitutievariabelen

### 11.2.2

#### Gebruikersvariabelen

### 11.2.3 S

#### ysteemvariabelen

## 11.3 SQL\*Plus-scripts

## 11.4 Rapportage met SQL\*Plus

## 11.5 BREAK en COMPUTE

# 12 Object-relationele features

## 12.1 Nog meer datatypes

## 12.2 Arrays

## 12.3 Geneste tabellen

## 12.4 Zelfgedefinieerde types

## 12.5 Multiset-operatoren

## Appendix A De casus

## Index

14

## **Hoofdstuk 1**

### **Inleiding relationele databasesystemen en Oracle**

Dit eerste hoofdstuk geeft een beknopte inleiding in het werken met relationele databases in het algemeen en met Oracle in het bijzonder. Doelstelling hierbij is vooral: thuisraken in de jungle van het (relationele) databasejargon.

De eerste drie paragrafen behandelen de redenen om een informatiesysteem te automatiseren met behulp van een database, wat er zoal komt kijken bij het ontwerpen en bouwen van een database, en wat de diverse onderdelen zijn van een databaseomgeving.

Vervolgens gaan een aantal paragrafen dieper in op de theoretische achtergronden van relationele databases: de categorie waarin Oracle thuishoort.

Dan volgt een overzicht van de Oracle-softwareomgeving; wat zijn de diverse onderdelen van dit pakket, wat zijn de kenmerken, en wat kunnen we ermee doen.

Ten slotte behandelt dit hoofdstuk de voorbeeldtabellen, waarop we in de loop van dit boek onze SQL-vaardigheden zullen ontwikkelen. Inzicht in de betekenis van deze zeven tabellen en hun kolommen, evenals hun onderlinge verbanden, is natuurlijk noodzakelijk om SQL te kunnen uitvoeren.

We besteden hier nog geen aandacht aan object-relationele databases; dat doen we [in hoofdstuk 12](#), waarin de mogelijkheden van Oracle op dat gebied aan de orde zullen komen.

## 1.1

### Informatiebehoefte en informatiesystemen

Organisaties hebben doelstellingen. Voor het realiseren van die doelstellingen moeten op vele momenten beslissingen genomen worden.

Voor het nemen van juiste beslissingen is dikwijls veel informatie nodig; deze informatie zal echter niet altijd kant-en-klaar beschikbaar zijn. Er is dus behoefte aan een systeem, dat op het juiste moment de 15

- 
- 
- 
- 
- 
- 

benodigde informatie produceert. Een dergelijk systeem noemen wij een informatiesysteem. Een informatiesysteem is een vereenvoudigde afspiegeling (een model) van de werkelijkheid binnen de organisatie.

We hoeven daarbij niet onmiddellijk te denken aan een geautomatiseerd informatiesysteem; ook kaartenbakken, ordners, of hangmappen kunnen de gegevens bevatten, die via bepaalde procedures (handelingen) tot de gewenste informatie leiden. Er zijn echter twee belangrijke redenen om een informatiesysteem te automatiseren:

#### *Complexiteit:*

De gegevensstructuur en/of de verwerking van de gegevens wordt te ingewikkeld.

#### *Volume:*

De hoeveelheid te beheren gegevens wordt te groot.

Als we besluiten een informatiesysteem te automatiseren, komt daar meestal databasetechnologie aan te pas. Enkele voordelen die databases bieden zijn:

### *Toegankelijkheid:*

Ad hoc bevrágingsmogelikheden, uitgebreide rapportagefaciliteiten, gemeenschappelijk gebruik van gegevens.

### *Beveiliging:*

Gedetailleerde autorisatiemogelikheden, herstelfaciliteiten na systeemstoringen.

Wat betreft informatiebehoefte dienen we een duidelijk onderscheid te maken tussen de volgende twee informatieaspecten:

#### *Het WAT:*

De *inhoud* van de gewenste informatie (het logische niveau).

#### *Het HOE:*

De *vorm* waarin de informatie dient te worden verstrekt, de manier waarop het resultaat moet worden afgeleid, de maximaal toelaatbare responstijden, ... (het fysieke niveau).

Databasesystemen zoals Oracle stellen ons in staat deze scheiding duidelijk aan te brengen, zodat we ons voornamelijk kunnen toeleggen op het eerstgenoemde aspect. Dat is een gevolg van het feit dat ze gebaseerd zijn op het relationele model, waarover binnenkort meer in dit hoofdstuk; zie de paragrafen 4, 5 en 6.

16

- 
- 

1.2

## Databaseontwerp

In een database slaan we feiten op over objecten. In het vakjargon gebruikt men voor een object meestal de term *entiteit*. We zijn vooral geïnteresseerd in

waarneembare kenmerken van dat object, ook wel *attributen* genaamd.

De bepaling van de informatiebehoefte bestaat nu in eerste instantie uit het beantwoorden van de volgende twee vragen:

Welke entiteiten zijn voor het informatiesysteem relevant?

Welke attributen zijn voor elke entiteit relevant?

We zullen hier binnenkort nog een derde stap aan toevoegen. Laten we als voorbeeld een opleidingsinstituut beschouwen dat

automatiseringscursussen verzorgt. Relevante entiteiten zouden kunnen zijn: cursus, cursist, cursusuitvoering, lokaal, docent, inschrijving, bevestiging, factuur, et cetera. Een (onvolledige) lijst relevante attributen van de entiteit *cursist* zou er als volgt uit kunnen zien: **Entiteit**

## **Attribuut**

cursist

Registratienummer

Naam

Adres

Woonplaats

Geboortedatum

E-mailadres

Leeftijd

Geslacht

En van de entiteit *cursus* zou die lijst er als volgt uit kunnen zien: **Entiteit**

## Attribuut

cursus

Titel

Lengte (in dagen)

Prijs

Frequentie

Maximaal aantal

17

cursisten

Zodra we een databaseontwerp gaan implementeren door een relationeel databasesysteem, zullen entiteiten geïmplementeerd worden als *tabellen*, en attributen binnen een entiteit als *kolommen*.

Wat de terminologie betreft: de keuze van de namen voor de entiteiten en attributen is uitermate belangrijk. Zij vormen immers de allereerste opstap in het begrijpen van (de semantiek van) een databaseontwerp.

Belangrijk is ook het *niveau* van de attributen. Zo heeft een cursus een titel en een zekere lengte, en heeft een cursusuitvoering een locatie, een startdatum en een docent. Dit verschil van niveau is des te meer reden om zorgvuldig na te denken over de namen die we kiezen voor de entiteiten; in de natuurlijke taal zijn we bijvoorbeeld genegen om altijd de term cursus te gebruiken en is wat we daarmee bedoelen – cursus danwel cursusuitvoering – vaak impliciet bepaald.

Bovendien moeten we duidelijk onderscheid maken tussen een entiteit zelf (generiek) en een specifiek voorkomen van die entiteit; in het laatste geval spreken we van een *occurrence* van die entiteit. Op dezelfde manier is er een verschil tussen een attribuut van een entiteit (generiek) en een specifieke attribuutwaarde binnen een occurrence van die entiteit.

Er zijn twee soorten gegevens: *basisgegevens* en *afleidbare gegevens*.

Een basisgegeven is een gegeven dat op geen enkele manier is af te leiden uit de overige gegevens van het informatiesysteem; een afleidbaar gegeven kan (bijvoorbeeld met een formule) wél worden afgeleid uit andere gegevens. Voorbeeld: als wij van elke cursist de leeftijd en de geboortedatum opnemen, zijn deze twee gegevens onderling afleidbaar (ervan uitgaande dat de actuele datum op ieder moment vorhanden is).

Goedbeschouwd levert iedere vraag aan een informatiesysteem als resultaat afgeleide gegevens op. Het is dus ondoenlijk om alle afleidbare gegevens in een informatiesysteem op te nemen. Opslag van afleidbare gegevens wordt ook wel *redundantie* (letterlijk vertaald: overbodigheid) genoemd.

18

- 
- 

Soms wordt toch selectief besloten tot het opslaan van redundante gegevens; vooral in gevallen waarin snelheid (performance) cruciaal is, en waarin het steeds opnieuw berekenen of afleiden van de gewenste gegevens te veel tijd zou kosten.

Het opslaan van afleidbare gegevens in een database moet met terughoudendheid gebeuren. Ten eerste is het jammer van de verspilde opslagcapaciteit. Dat is echter niet het grootste probleem, omdat gigabytes schijfcapaciteit tegen steeds lagere kosten beschikbaar komen.

Het grootste probleem ligt op een ander vlak: met het opslaan van afleidbare gegevens komt de plicht om deze gegevens steeds opnieuw af te leiden zodra één van de (basis) gegevens waaruit ze afgeleid zijn, wijzigt. Als hierbij iets misgaat of vergeten wordt, kan dat leiden tot een informatiesysteem dat interne tegenstrijdigheden bevat. Men zegt dan dat de gegevens niet meer *consistent* zijn. Redundantie in een informatiesysteem is dus een voortdurende bedreiging van de consistentie.

Met betrekking tot het al dan niet opslaan van redundante gegevens in een informatiesysteem is het belangrijk onderscheid te maken tussen twee types

informatiesystemen:

Online transactieverwerkende systemen (OLTP, on line transaction processing) waarin voortdurend wijzigingen worden doorgevoerd, veelal met een hoog volume.

Beslissingsondersteunende systemen (DSS, decision support system) waarin voornamelijk (of zelfs uitsluitend) gegevens worden geraadpleegd, die op regelmatige tijdstippen worden gevoed of vervaard vanuit OLTP-systemen.

In DSS-systemen is het vrij gebruikelijk om redundante gegevens op te slaan, om zodoende betere responsijken te realiseren. Het raadplegen van opgeslagen redundante gegevens is sneller dan het afleiden van deze gegevens ten tijde van raadpleging, terwijl het bovengenoemde inconsistentierisico veel minder aan de orde is omdat DSS-systemen doorgaans alleen maar geraadpleegd worden.

Consistentie is uiteraard een eerste vereiste voor ieder

informatiesysteem, wil men er betrouwbare informatie uit kunnen putten. In de tweede plaats moet de *integriteit* van de gegevens onder alle omstandigheden gewaarborgd zijn. We zullen dit begrip aan de hand van enkele voorbeelden verduidelijken. Stel dat de volgende 19

gegevens aan het informatiesysteem zijn ontleend:

1

Cursist 6749 is geboren op 13 februari 2930.

2

Diezelfde cursist is van het geslacht ‘Q’.

3

Er is nog een andere cursist met hetzelfde nummer 6749.

4

Er bestaat een inschrijving voor cursist 8462, maar deze cursist komt niet in de administratie voor.

In geen van bovenstaande gevallen is de consistentie in het geding; het informatiesysteem is ondubbelzinnig in zijn uitspraken. Toch is er iets mis; deze gegevens zijn niet in overeenstemming met ons idee van de werkelijkheid. We zouden het namelijk normaal vinden, als: 1

Een geboortedatum niet in de toekomst ligt.

2

Het attribuut geslacht als waarde ‘M’ of ‘V’ heeft.

3

Iedere cursist een uniek nummer heeft.

4

Alleen inschrijvingen voorkomen van bestaande – dat wil zeggen in het systeem geregistreerde – cursisten.

Dit soort beperkende voorwaarden met betrekking tot de toegestane inhoud van een database noemen we *constraints*. Door nu de juiste constraints te definiëren kunnen we de eisen met betrekking tot de integriteit vastleggen. Het derde voorbeeld is overigens een *primaire sleutel* constraint, en implementeert de *entiteitsintegriteit*; het vierde voorbeeld is een *refererende sleutel* constraint, waarmee de *referentiële integriteit* wordt bewaakt. Hier komen we in een later stadium nog op terug.

Constraints worden vaak geklassificeerd door te kijken naar hun reikwijdte (scope) binnen het databaseontwerp. Dit zijn de vier klassen, met elk een voorbeeld:

1

*Attribuutconstraints, scope is één attribuut: geslacht is ‘M’of ‘V’.*

2

*Rijconstraints, scope is meerdere attributen binnen één rij:* voor verkopers is het attribuut commissie een verplicht veld.

3

*Tabelconstraints, scope is meerdere rijen binnen één tabel:* iedere medewerker heeft een uniek e-mailadres.

4

*Databaseconstraints, scope is meerdere tabellen:* iedere medewerker is verbonden aan een bestaande afdeling.

In hoofdstuk 7 komen we terug op constraints en de manier waarop ze in SQL kunnen worden gespecificeerd.

In het begin van deze paragraaf hebben we gezien dat de

20

informatiebehoefte in eerste instantie wordt bepaald door vast te leggen welke entiteiten voor het informatiesysteem relevant zijn, en daarna per entiteit vast te leggen welke attributen relevant zijn. Als we dit nu completeren met het vastleggen van de bijbehorende relevante constraints, is het *datamodel* voor het informatiesysteem gereed.

Het ontwikkelen van een deugdelijk datamodel is bepaald geen sinecure, en een zaak voor automatiseringsspecialisten (informatieanalisten en/of databaseontwerpers). Het is anderzijds uitgesloten dat een specialist een goed datamodel kan ontwerpen zonder de actieve inbreng van de (toekomstige) gebruikers van het systeem. Bij hen is immers de materiekennis aanwezig, en ligt tevens de uiteindelijke beslissing tot acceptatie van het systeem. Intensieve samenwerking tussen beide partijen is dus vereist, opdat het databaseontwerp een juiste en zinvolle afspiegeling is van de werkelijkheid.

In de loop der tijd zijn vele *methoden* ontwikkeld ter ondersteuning van het ontwikkelproces, voor het genereren van documentatie, ten behoeve van

communicatie en projectbeheersing (tijd en kosten). Traditionele methoden kenmerken zich door een duidelijke fasering van het ontwikkelproces, en een omschrijving van wat in welke volgorde dient te gebeuren. Deze methodes worden ook wel ‘waterval’-methodes genoemd. Vrij algemeen (en populair) geformuleerd kunnen we in dergelijke methoden de volgende fasen onderscheiden:

1

analyse;

2

logisch ontwerp;

3

fysiek ontwerp;

4

bouw.

Binnen de diverse fasen kunnen *technieken* worden toegepast om de werkzaamheden te ondersteunen. Hierbij kan men denken aan

diagramtechnieken om een datamodel grafisch weer te geven. Bekende voorbeelden hiervan zijn ERD (entity relationship diagram) en UML

(unified modeling language). In de laatste paragraaf van dit hoofdstuk – waarin de casus van dit boek wordt geïntroduceerd – zullen we een ERD van de casus zien.

Een ander voorbeeld van een bekende techniek is het *normaliseren*, waarmee eventuele redundantie uit een ontwerp kan worden verwijderd.

Ook *prototyping* wordt vaak als techniek toegepast; men bouwt 21

- 
- 
- 
- 
- 
- 
- 
- 
- 
- 
- 

betrekkelijk snel stukjes programmatuur om een systeem te simuleren, met als doel reacties van gebruikers los te krijgen. Dit kan tijdwinst opleveren in de analysefase van het ontwikkelproces, maar vooral ook kwaliteitswinst en daarmee een hogere kans op acceptatie van het eindresultaat.

Modernere methoden in deze wereld zijn:

RAD, zie

[http://en.wikipedia.org/wiki/Rapid\\_application\\_development](http://en.wikipedia.org/wiki/Rapid_application_development)

RUP, zie

[http://en.wikipedia.org/wiki/IBM\\_Rational\\_Unified\\_Process](http://en.wikipedia.org/wiki/IBM_Rational_Unified_Process)

DSDM, zie

[http://en.wikipedia.org/wiki/Dynamic\\_systems\\_development\\_method](http://en.wikipedia.org/wiki/Dynamic_systems_development_method)

SCRUM, zie [http://en.wikipedia.org/wiki/Scrum\\_\(development\)](http://en.wikipedia.org/wiki/Scrum_(development))

Bovenstaande methoden worden beschouwd als *agile* development methoden (zie

[http://en.wikipedia.org/wiki/Agile\\_software\\_development](http://en.wikipedia.org/wiki/Agile_software_development)).

Wil men op de juiste wijze gebruikmaken van een informatiesysteem, dan zal

men goed op de hoogte moeten zijn van de *semantiek* (de betekenis van de termen) van het datamodel. Zoals eerder vermeld is een weloverwogen keuze van namen van de tabellen en kolommen hierbij een belangrijk uitgangspunt, evenals een consequente toepassing daarvan. Het attribuut ‘adres’ kan bijvoorbeeld vele betekenissen hebben: woonadres, werkadres, correspondentieadres, etc. De betekenissen van attributen die tot dit soort twijfel blijven leiden, kunnen explicet worden vastgelegd in een *semantische toelichting* op het datamodel. Deze toelichting vormt weliswaar geen onderdeel van de formele gegevensstructuur, maar kan wél als commentaar in een *datadictionary* – een term die in de volgende paragraaf nader wordt toegelicht – worden vastgelegd.

In deze paragraaf zijn achtereenvolgens de volgende begrippen geïntroduceerd:

entiteiten en attributen;

tabellen en kolommen;

occurrences en attribuutwaarden;

basisgegevens en afleidbare gegevens;

redundantie en consistentie;

integriteit en constraints;

22

- 
- 
- 
- 
- 
- 
- 
- 



- 

-

- - 
  - 
  - 
  - 
  -
- datamodelleren;
- methoden en technieken;
- logisch en fysiek ontwerp;
- normaliseren;
- agile;
- semantiek.

### 1.3

#### Database-managementsysteem

In de voorgaande paragrafen is uiteengezet wat we verstaan onder een informatiesysteem. Als de automatisering te hulp wordt geroepen om een dergelijk informatiesysteem te realiseren, kunnen we het begrip *database* als volgt definiëren:

Een database is een verzameling gegevens die nodig is om aan een informatiesysteem de gewenste informatie te kunnen onttrekken, beheerd door een afzonderlijk programmatuursysteem.

Dat afzonderlijke programmatuursysteem is het

*databasemanagementsysteem* (afgekort tot DBMS). Er zijn vele soorten databasemanagementsysteem, variërend wat betreft:

prijs;

realiseerbaarheid van complexe informatiesystemen;

hardwareomgeving;

flexibiliteit voor de bouwers;

flexibiliteit voor de gebruikers;

koppelingsmogelijkheden met andere programmatuur;

gebruiksvriendelijkheid.

Een DBMS heeft verschillende onderdelen. De basis bestaat uit de programmatuur die de fysieke opslag van de gegevens voor zijn rekening neemt, het gegevenstransport (I/O) regelt van extern naar intern geheugen, de integriteit bewaakt, enzovoorts. We zullen dit centrale onderdeel van het DBMS aanduiden met de term *kernel*.

Daarnaast wordt door het DBMS een *datadictionary* onderhouden, waarin alle gegevens over de gegevens (de metagegevens) worden bijgehouden. In een datadictionary worden onder meer de volgende zaken ondergebracht:

totaaloverzicht van entiteiten en attributen;

23

- 卷之三

- 
- 
- 
- 

constraints (integriteit);

toegangsrechten tot de gegevens;

eventuele semantische toelichtingen;

(gebruikers)autorisatiegegevens;

applicatiegegevens.

Bovendien zal elk DBMS een of meer talen ondersteunen, om de gegevens die in de database zijn opgeslagen te kunnen benaderen. Men spreekt hierbij meestal over *vraagtaLEN*, hoewel deze benaming nogal misleidend is, aangezien naast het opvragen ook het aanmaken en manipuleren van gegevens onderdeel vormt van zo'n taal. SQL, de taal waar dit boek over gaat, is al jaren de marktstandaard.

Ten slotte zullen leveranciers allerlei programma's als bijproduct rondom hun DBMS leveren. We zullen deze programma's samenvatten onder de term *tools*. Deze tools stellen de gebruikers bijvoorbeeld in staat om:

rapporten te genereren;

standaard in- en uitvoerschermen te bouwen;

databasegegevens in tekst of spreadsheets te verwerken;

databasebeheer te plegen (denk bijvoorbeeld aan het maken van backups).

Voor de duidelijkheid zullen we in deze paragraaf nog een extra begrip introduceren: *databaseapplicaties*. Hieronder verstaan we toepassingsprogramma's die voor hun gegevensopslag gebruikmaken van een onderliggende database. Deze kunnen schermen/of

menugestuurde invoerprogramma's zijn (tegenwoordig vaak browser-based), spreadsheets, lijstgeneratoren, et cetera. Databaseapplicaties worden vaak ontwikkeld met behulp van een tool van de DBMS-leverancier, maar kunnen ook met tools van andere leveranciers ontwikkeld worden.

In deze paragraaf zijn de volgende begrippen toegelicht:

database;

databasemanagementsysteem;

kernel;

datadictionary;

vraagtalen;

tools;

databaseapplicaties.

24

1.4

## Relationele databases

De theoretische basis voor relationele databases werd in 1970 gelegd door Ted Codd in het artikel ‘A relational model of data for large shared data banks’ (Codd, 1970). Hij baseerde zich hierbij op enkele klassieke onderdelen van de wiskunde: de verzamelingenleer, de relationele calculus en de algebra. We zullen dit wiskundig fundament van relationele databases in dit boek zo min mogelijk aan de orde laten komen, maar in deze paragraaf kunnen we daar niet onderuit, als we de term relationeel proberen te verklaren. Overigens zal in de praktijk blijken dat enig wiskundig inzicht bepaald geen kwaad kan bij het oplossen van vragen in SQL die het triviale niveau ontstijgen.

Het heeft tot omstreeks 1980 geduurd, voordat de eerste DBMS'sen op de markt kwamen die de ideeën van Ted Codd (min of meer) in de praktijk

brachten. Tot de leveranciers van het eerste uur behoren onder andere Oracle en Ingres, enkele jaren later gevolgd door IBM met SQL/DS en DB2.

De ideeën van Ted Codd kwamen in essentie op het volgende neer:

1 Maak zowel bij het ontwerp als bij het gebruik van een database een duidelijk onderscheid tussen de logische taak (het wat) en de fysieke taak (het hoe).

2 Zorg ervoor dat een DBMS van de gebruiker alleen nog uitvoering van de logische taak verlangt, en vervolgens de fysieke taak voor eigen rekening neemt.

3 Deze ideeën waren, hoe voor de hand liggend ze nu ook klinken, in die tijd beslist revolutionair. De meeste DBMS'en brachten dit onderscheid absoluut niet aan, waren nauwelijks of niet gebaseerd op een solide theorie, en zaten voor de gebruikers vol verrassingen, ad-hoc oplossingen en uitzonderingen.

4 Het artikel van Ted Codd heeft als grootste verdienste gehad dat men vanaf dat moment op een andere manier is gaan denken over databases.

5 Sindsdien is de theorie steeds in ontwikkeling voorgebleven op de praktijk; en zo hoort het eigenlijk ook.

6 Wat maakt een DBMS nu relationeel? Of anders geformuleerd: hoe is 25 het relationele gehalte van een DBMS te bepalen?

Om deze vragen te kunnen beantwoorden moeten we naar de theorie. In de volgende paragrafen worden twee aspecten van het relationele model belicht: de relationele gegevensstructuur en de relationele operatoren. In de daaropvolgende paragraaf wordt ten slotte geprobeerd om een antwoord te geven op de vraag: ‘Hoe relationeel is mijn DBMS?’

1.5 Relationele gegevensstructuur

Het centrale begrip in de relationele gegevensstructuur is de *tabel* of *relatie* (vandaar de naam van het model). Een tabel wordt beschouwd als een verzameling *rijen*, of *tupels*. Alle gegevens van een relationele database liggen vast in de vorm van *kolomwaarden* binnen een rij van een tabel.

Dit is zeer consequent doorgevoerd; de enige manier om in een relationele database gegevens met elkaar in verband te brengen bestaat uit de vergelijking van kolomwaarden. Een kenmerk van het wiskundige begrip verzameling is dat de volgorde der elementen betekenisloos is; dit geldt dus ook voor de rijen van elke willekeurige tabel, evenals voor de volgorde van de kolommen.

Nogmaals: er bestaan op zich geen ingebakken verbanden tussen tabellen onderling. Dit betekent dat we in een vraagstelling steeds expliciet aan moeten geven welk verband gelegd moet worden tussen de diverse rijen. Een gevolg hiervan is de vrijwel onbegrensde flexibiliteit om ad-hoc vragen aan een relationele database te stellen. De keerzijde van de medaille is het risico van (denk)fouten, en het probleem van de correctheid. Nagenoeg iedere in SQL geformuleerde vraag levert een antwoord op, maar is het wel het bedoelde antwoord?

Meestal is er een één-op-één-afbeelding mogelijk van entiteiten van het datamodel op tabellen in de database. De rijen kunnen dan worden opgevat als de occurrences van die entiteit, en de kolomkoppen van de tabel kunnen dan worden gezien als de attributen van die entiteit.

Samenvattend:

1

Een database is een verzameling tabellen.

2

Een tabel is een verzameling rijen.

26



## ■ 3

Een rij is een verzameling kolomwaarden.

Iets preciezer geformuleerd: een rij is een verzameling geordende paren, waarbij elk geordend paar bestaat uit een attribuut(naam) met een bijbehorende attribuutwaarde.

Tijdens datamodellering wordt vaak vastgelegd welke waarden voor een attribuut zijn toegestaan. Zo'n verzameling toegestane waarden voor een bepaald attribuut wordt ook wel een *domein* genoemd. Men spreekt in dit verband ook wel van datatypes, of kortweg types; elk attribuut wordt gedefinieerd op een bepaald type. Dat kan een standaardtype zijn, of een zelfgedefinieerd type.

Iedere tabel dient minstens één *kandidaatsleutel* (candidate key) te hebben; dit is een attribuut (of combinatie van attributen) waarvan de waarde iedere rij uniek identificeert, en waarvoor bovendien geldt dat deze eigenschap verloren gaat zodra we één of meer attributen uit de combinatie weglaten (het is dus een *minimale* combinatie van attributen). Er zullen dus nooit twee rijen in een tabel mogen voorkomen met dezelfde waarden voor een kandidaatsleutel. Als een tabel meerdere kandidaatsleutels heeft, wijzen we er (over het algemeen) één aan als *primaire sleutel*. Het is *niet* toegestaan om voor de primaire sleutel geen waarde te verstrekken. Primaire sleutels bewaken daarmee een belangrijke vorm van *database-integriteit*.

Een tabel kan ook *refererende sleutels* (foreign keys) bevatten; dit zijn attributen waarvan men eist dat iedere voorkomende waarde elders in een primaire sleutelkolom terug te vinden is. Hiermee wordt de *referentiële integriteit* van de database bewaakt.

De hierboven genoemde domein- en sleutelconcepten zijn feitelijk bijzondere gevallen van constraints: een domein kan beschouwd worden als een attribuut-constraint, en sleutels kunnen beschouwd worden als tabel- of database-constraints. Vanuit de wiskundige basis die Ted Codd geïntroduceerd heeft, zijn willekeurig complexe beperkingsregels mogelijk. Enkele voorbeelden kunnen zijn:

Uitvoeringen van cursussen die slechts één dag duren, dienen minimaal tien cursisten te hebben;

Een docent kan niet op hetzelfde moment meerdere cursusuitvoeringen geven.

27

- 
- 
- 
- 
- 
- 
- 
- 
- 
- 

Voor dit soort constraints is in de taal SQL het concept *assertion* geïntroduceerd. Met SQL assertions kunnen willekeurig complexe constraints gespecificeerd worden. Helaas wordt dit concept door nog geen enkele databaseleverancier ondersteund.

Een RDBMS behoort op een systematische manier om te gaan met *ontbrekende informatie*. Als ergens een attribuutwaarde ontbreekt, is het niet altijd mogelijk om te bepalen of er al dan niet wordt voldaan aan een gestelde voorwaarde. Het ontbreken van informatie wordt in de relationele wereld aangegeven met *null-waarden*. Deze null-waarden leiden tot een *driewaardige logica*, zoals die in de taal SQL is geïmplementeerd. Hierover zijn de meningen overigens nogal verdeeld; Chris Date bijvoorbeeld is een fervent tegenstander van deze driewaardige logica. Zijn verhandelingen over dit onderwerp zijn zeer leesbaar en de moeite waard.

Ter afsluiting van deze paragraaf: er is nog een manier om (vanuit een ander perspectief) tegen tabellen en rijen in een relationele database aan te kijken, ontleend aan de wiskundige logica. Iedere tabel kunnen we associëren met een *predikaat*, en alle rijen van een tabel bestaan uit corresponderende *proposities*. Een predikaat is een uitspraak (waarin over het algemeen variabelen voorkomen) die waar of onwaar kan zijn, bijvoorbeeld: “Er is een

cursus met titel T en lengte L, prijs P, frequentie F, en een maximaal aantal deelnemers M.” Als we nu in dit predikaat voor de vijf variabelen (T, L, P, F en M) waarden invullen, dan krijgen we een propositie. Een propositie is een predikaat zonder variabelen; met andere woorden, een propositie is altijd waar of onwaar. Welnu, dat betekent dat we de rijen in een tabel kunnen beschouwen als de proposities die, in de werkelijkheid die we gemodelleerd hebben, waar zijn.

In deze paragraaf werden de volgende begrippen geïntroduceerd: tabel of relatie;

rij of tupel;

kolom, domein;

primaire en refererende sleutels;

integriteitsbewaking op databaseniveau, SQL assertions;

ontbrekende informatie en driewaardige logica;

predikaten en proposities.

28

1.6

## Relationele operatoren

Om met gegevens te kunnen manipuleren zijn bewerkingen nodig. De wiskundige term voor bewerking is *operator*. De vermenigvuldiging en de optelling zijn voorbeelden van operatoren: je stopt er twee getallen in, en er komt als resultaat van de operatie één getal uit. Omdat er precies dezelfde dingen uitkomen als er worden ingestopt (namelijk getallen), noemen we deze operatoren *gesloten*. Dat is een prettige eigenschap, omdat de resultaten weer kunnen worden gebruikt als invoer voor een volgende bewerking.

In een database hebben we óók behoefte aan operatoren, om uit de opgeslagen gegevens informatie af te kunnen leiden. Binnen een RDBMS behoren dat

operatoren te zijn, die op een hoog *logisch niveau* liggen. Dat betekent onder andere dat ze niet op afzonderlijke rijen maar op tabellen worden losgelaten, en als resultaat weer een tabel opleveren.

Omdat tabellen zijn gedefinieerd als verzamelingen, zullen de relationele operatoren dus bewerkingen op verzamelingen moeten verrichten. Vandaar dat enkele operatoren uit de klassieke verzamelingenleer – zoals de vereniging, het verschil en de doorsnede –

ook als relationele operator opduiken.

Daarnaast komen relationele operatoren voor die specifiek op tabellen (als een bijzondere vorm van verzamelingen) zijn gedefinieerd. Men kan in principe net zo veel relationele operatoren verzinnen als men zelf wil; over het algemeen zijn ze allemaal te herleiden tot een aantal basisoperatoren. Hieronder volgen enkele voorbeelden van relationele operatoren.

*Restrictie:* Op basis van een bepaalde voorwaarde worden uit de tabel die wordt benaderd bepaalde rijen wél, en andere niet in de resultaattabel toegelaten. Deze operator wordt ook wel aangeduid met de term *selectie*.

*Projectie:* Van een tabel worden slechts bepaalde kolommen in de resultaattabel toegelaten.

*Vereniging* (of union): Uit twee tabellen wordt één resultaattabel afgeleid, die alle rijen bevat die in de éne ofwel in de andere tabel voorkomen, of in beide tabellen.

29

*Doorsnede* (of intersection): Uit twee tabellen wordt één resultaattabel afgeleid, die uitsluitend die rijen bevat die zowel in de ene als in de andere tabel voorkomen.

*Verschil* (of minus): Uit twee tabellen wordt één resultaattabel afgeleid, die uitsluitend die rijen bevat die wél in de ene maar niet in de andere tabel voorkomen.

*Product:* Uit twee tabellen wordt één resultaattabel afgeleid, waarvan de rijen bestaan uit alle mogelijke combinaties van een rij uit de ene met een rij uit de andere tabel, door ze aan elkaar te plakken. Men spreekt ook wel van het *Cartesiaans* of *Cartesisch product*.

*Natuurlijke join:* Uit twee tabellen wordt één resultaattabel afgeleid, waarvan de rijen bestaan uit alle mogelijke combinaties van een rij uit de ene met een rij uit de andere tabel, mits beide rijen voor alle gelijknamige attributen dezelfde waarden hebben. Een combinatie van twee rijen ontstaat door ze aan elkaar te plakken, en de dubbele attribuutwaarde slechts eenmaal in de resultaatrij op te nemen.

De natuurlijke join is een voorbeeld van een operator die strikt genomen niet nodig is, omdat hij kan worden samengesteld. Het resultaat van de natuurlijke join kan namelijk ook worden bereikt door toepassing van achtereenvolgens de operatoren product, restrictie en projectie.

## 1.7

### Hoe relationeel is mijn DBMS?

De term relationeel wordt door veel leveranciers gehanteerd. Als wij willen vaststellen in hoeverre deze leveranciers de waarheid spreken, stuiten we op het probleem dat relationeel een theoretisch begrip is. De vraag: ‘Is een DBMS relationeel?’ is daarom moeilijk met ‘ja’ of ‘nee’

te beantwoorden. Het is verstandiger om te spreken van het *relationele gehalte* van een DBMS.

Dit probleem is door Ted Codd ook onderkend; vandaar dat hij twaalf regels heeft geformuleerd waaraan een RDBMS zou moeten voldoen.

Zonder in details te treden sommen wij ze hier op, met een summiere toelichting:

## 1

### *Representatie gegevens*

Alle gegevens in de database worden explicet op logisch niveau 30 voorgesteld op precies één manier: als kolomwaarden in tabellen.

2

### *Toegankelijkheid gegevens*

Iedere opgeslagen waarde is gegarandeerd bereikbaar door de combinatie van de tabelnaam, een waarde van de primaire sleutel en de naam van het attribuut waarvan we de waarde zoeken.

3

### *Ontbrekende informatie*

NULL-waarden moeten op een systematische manier worden ondersteund om het ontbreken van informatie te representeren en te manipuleren.

4

### *Dynamische online datadictionary*

De beschrijving van de database wordt op dezelfde manier logisch gerepresenteerd als gewone gegevens, zodat een en dezelfde taal kan worden gehanteerd voor de bevraging van beide soorten gegevens.

5

### *Datagerichte subtaal*

Er moet minstens één taal worden ondersteund, die voorziet in: datadefinitie, view-definitie, datamanipulatie, constraintdeclaratie, autorisatie en transactieafhandeling.

6

## *Updateable views*

Het systeem moet datamanipulatie (insert/update/delete) toestaan op alle views die theoretisch updateable zijn.

7

## *High-level insert, update en delete*

Niet alleen raadpleging, maar ook datamanipulatie dient op het logische niveau van tabellen (verzamelingen) te geschieden.

8

## *Fysieke gegevensonafhankelijkheid*

Applicaties hoeven niet te worden aangepast als onderliggende fysieke opslag- of toegangsmogelijkheden worden veranderd.

9

## *Logische gegevensonafhankelijkheid*

Applicaties hoeven niet te worden aangepast als niet-destructieve wijzigingen worden aangebracht op logisch niveau.

## *10 Integriteitsonafhankelijkheid*

Constraints moeten met behulp van de datagerichte subtaal kunnen worden gedefinieerd, en moeten worden opgeslagen in de datadictionary.

## *11 Distributieonafhankelijkheid*

Applicaties hoeven zich niet bewust te zijn van het feit dat de gegevens verspreid over een gedistribueerde database zijn opgeslagen.

## 12 Geen ondermijning

Als ook een low-level taal wordt ondersteund, mag die taal niet in staat zijn de constraints die op een hoger niveau zijn gedefinieerd 31

(op tabel- of databaseniveau) met voeten te treden.

### 1.8

#### De Oracle-software

Oracle is een softwareomgeving die op een groot aantal platforms beschikbaar is, variërend van PC's via diverse minicomputers (HP, Sun) tot grote mainframes en massief-parallelle systemen. Dit is één van de sterkste kanten van Oracle: het garandeert onafhankelijkheid van hardwareleveranciers, groeimogelijkheden zonder verlies van gedane investeringen, en uitgebreide data transport- en

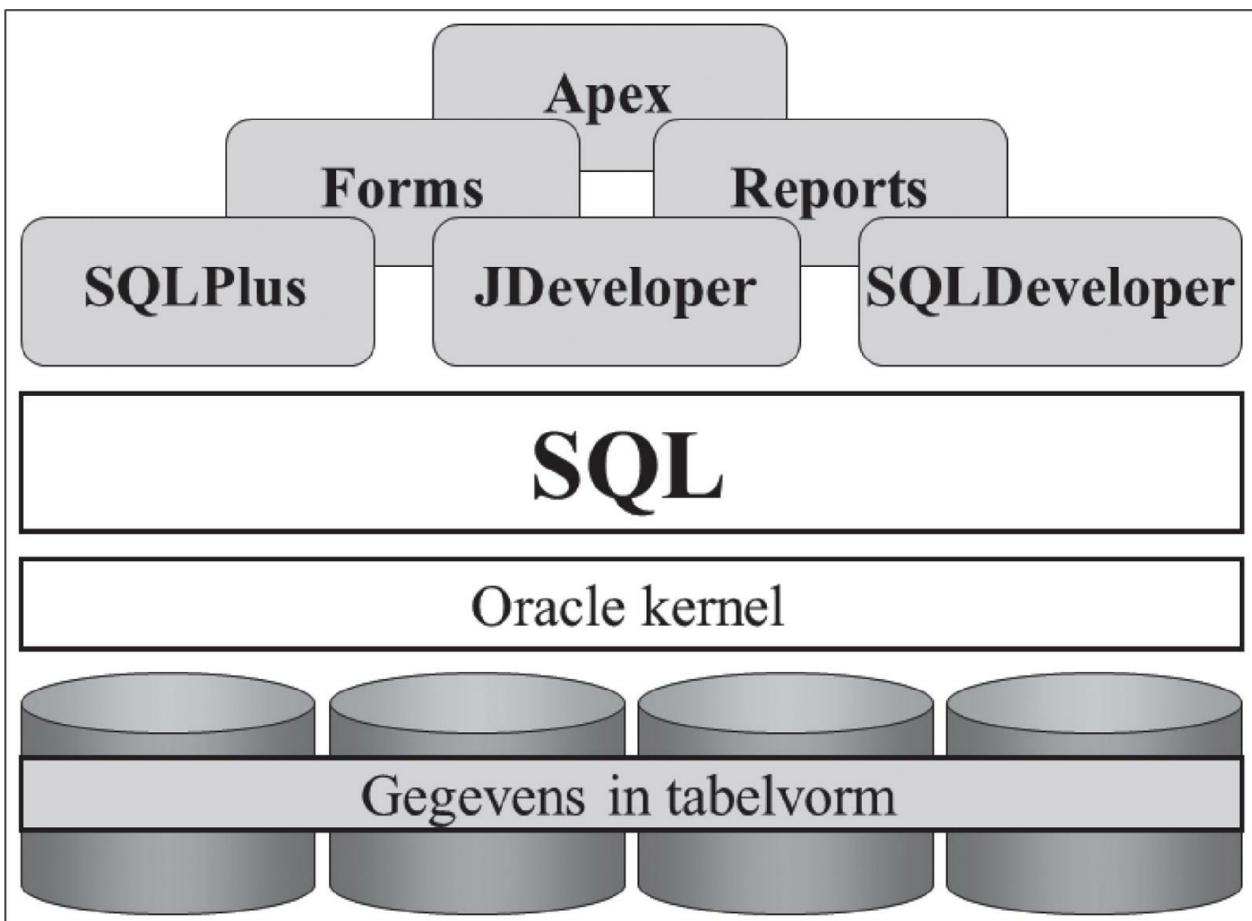
communicatiemogelijkheden in een heterogene omgeving.

Oracle is een softwareomgeving die bestaat uit vele onderdelen, waarvan de basis gevormd wordt door het RDBMS zelf: de *kernel*. De kernel handelt onder meer alle fysieke transport van en naar de database af, en zorgt feitelijk voor de logische representatie van gegevens in tabelvorm. Een belangrijk onderdeel van de kernel is de *optimizer*; de optimizer bepaalt de strategie waarmee de SQL-commando's zullen worden afgehandeld.

Applicaties en gebruikers kunnen met de kernel communiceren met de *taal SQL*, het belangrijkste onderwerp van dit boek. De Oracle-implementatie van deze taal is een vrijwel volledige implementatie van de ISO/IEC SQL:2003-standaard; Oracle speelt als marktleider een vooraanstaande rol in het SQL-standaardisatieproces.

Verder is een hele reeks Oracle- *tools* beschikbaar, die het werken met Oracle moeten veraangenamen. De samenwerking van deze tools met de database wordt [in figuur 1.1 geïllustreerd](#). Hieruit blijkt duidelijk dat SQL onder alle omstandigheden het communicatiemiddel is met de kernel, ongeacht de tool die wordt gebruikt.

Let op het volgende belangrijke verschil tussen SQL en SQL\*Plus: SQL  
is een *taal*, en SQL\*Plus is een *tool* (waarmee op eenvoudige wijze SQL aan  
het DBMS aangeboden kan worden).



**Figuur 1.1**

Behalve tools waarmee applicaties kunnen worden gebouwd, levert Oracle ook vele kant-en-klare applicaties, zoals de Oracle Fusion Applications. Deze categorie producten blijft in dit boek buiten beschouwing.

Het bedrijf Oracle heeft zijn hoofdkwartier in Redwood Shores in Californië. Het is opgericht in 1977, en was in 1979 de eerste leverancier van een commercieel relationeel databasemanagementsysteem. Oracle is het op één na grootste softwarebedrijf ter wereld, met meer dan 40.000 werknemers. Het is verreweg de grootste leverancier van databasesoftware ter wereld. Daarnaast levert het diverse diensten, zoals consultancy, training, en support. Oracle heeft inmiddels vestigingen in meer dan 140 landen, verspreid over de hele wereld.

## SQLPlus en SQLDeveloper

De tools die het dichtst bij SQL blijven; bij uitstek geschikt voor interactieve ad-hoc databasebenadering. Dit zijn de tools waarmee we in dit boek voornamelijk zullen werken. SQLPlus is een eenvoudige, op 33 command-lines gebaseerde tool. SQLDeveloper biedt een menugestuurde grafische omgeving waarmee (onder andere) SQL ontwikkeld kan worden.

## **Forms en Reports**

Dit zijn de traditionele Oracle-tools waarmee client-server databaseapplicaties ontwikkeld kunnen worden.

## **JDeveloper en Apex**

Dit zijn de modernere Oracle-tools waarmee browser-based database applicaties ontwikkeld kunnen worden. Bij Jdeveloper vormt Java de programmeertaal, bij Apex vormt PL/SQL de programmeertaal.

1.9

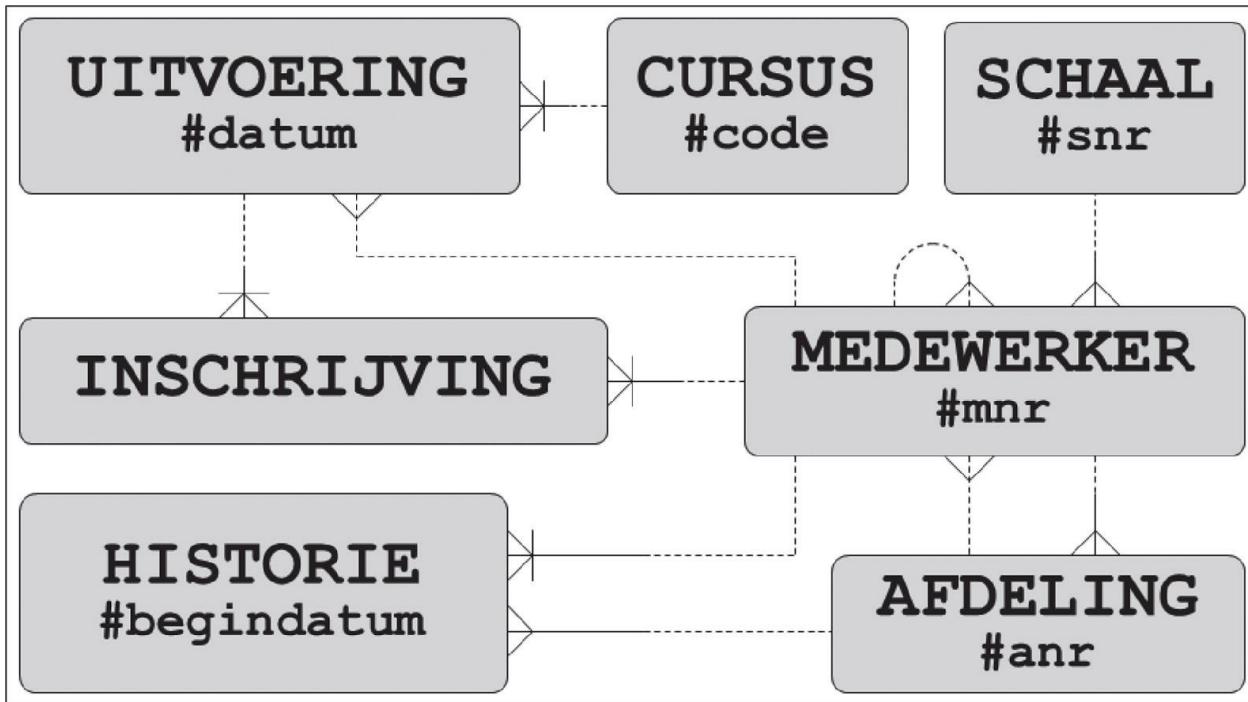
### De casus

In dit boek wordt gebruikgemaakt van zeven tabellen, die in deze paragraaf worden geïntroduceerd en toegelicht. In bijlage A is de casus ook gedocumenteerd; in deze bijlage zijn ook enkele verhelderende overzichten van de inhoud van de casus opgenomen.

Het spreekt voor zich dat inzicht in de structuur van de casus vereist is om de taal SQL met succes te kunnen loslaten op de inhoud ervan; de correctheid van SQL-commando's kan nooit worden gegarandeerd zonder deze kennis.

We beginnen met een ER-diagram van de casus ([zie figuur 1.2](#)). Dit is een afbeelding van een *logisch ontwerp*, wat betekent dat er nog op geen enkele wijze rekening is gehouden met de implementatieomgeving. Bij een *fysiek ontwerp* is dat wel het geval: dan is bijvoorbeeld al de keuze gemaakt voor implementatie in een Oracle-omgeving, en praten we al over tabellen.

Om dit soort ER-diagrammen correct te kunnen interpreteren is enige uitleg nodig. Er is sprake van zeven entiteiten, die worden weergegeven in een ‘soft box’. Om het diagram leesbaar te houden zijn vrijwel alle attributen achterwege gelaten; alleen de sleutelattributen zijn weergegeven.



**Figuur 1.2**

Tussen de entiteiten bestaan diverse relaties. De tien ‘kraaienpoten’ in het diagram geven ‘één-op-veel’-relaties weer. Elk van deze relaties is in twee richtingen te lezen. De ‘werkt voor’-relatie dienen we bijvoorbeeld als volgt te interpreteren:

Iedere medewerker werkt voor precies één afdeling.

Voor een afdeling kunnen nul, één of meer medewerkers werken.

**Merk op:** afdelingen zonder werknemers zijn toegestaan. Alle één-op-veelrelaties in onze casus hebben deze eigenschap, wat in dit soort diagrammen wordt aangegeven met een stippellijn. Tussen

medewerker en afdeling bestaan overigens twee relaties: een medewerker ‘werkt voor’ een afdeling, en ‘kan hoofd zijn van’ een afdeling. De relatie ‘heeft als chef’ is een voorbeeld van een *recursieve relatie*, ofte wel een

relatie van een entiteit met zichzelf.

Elke entiteit heeft een *unique identifier*, waarmee occurrences van die entiteit onderling onderscheiden kunnen worden. Dit kan een enkel attribuut zijn (bijvoorbeeld mnr voor de entiteit medewerker), een combinatie van attributen, al of niet gecombineerd met relaties. Een attribuut dat onderdeel is van een unique identifier wordt voorafgegaan door een hekje (#); relaties die onderdeel zijn van een unique identifier worden gemerkt met een dwarsstreepje. Zo bestaat de unique identifier 35

van de entiteit UITVOERING bijvoorbeeld uit de combinatie van het attribuut datum en de relatie naar de entiteit CURSUS, en de unique identifier van de entiteit INSCHRIJVING uit de twee relaties naar respectievelijk MEDEWERKER en UITVOERING.

De vertaalslag naar een tabelstructuur verloopt grofweg als volgt:

1 Iedere entiteit wordt een tabel.

2

Ieder attribuut wordt een kolom.

3

Iedere relatie wordt omgezet in een refererende sleutel (FK) (aan de kant van de kraaienpoot).

4

Iedere unique identifier wordt omgezet in een primaire sleutel (PK).

De volgende zeven tabellen zijn daarvan het resultaat:

## **MEDEWERKERS**

MNR

Nummer, dat voor iedere medewerker uniek is **PK**

**NAAM**

Achternaam, evt. voorafgegaan door  
voorvoegsels

**VOORL**

Voorletters (zonder interpunctie)

**FUNCTIE**

Taakomschrijving van de medewerker

**CHEF**

Het nummer van de chef van de medewerker **FK**

**GBDATUM**

Geboortedatum van de medewerker

**MAANDSAL**

Maandsalaris (exclusief toelage)

**COMM**

Onderdeel van het jaarsalaris dat alleen voor  
verkopers van toepassing is (commissie)

**AFD**

Nummer van de afdeling waaraan de

**FK**

medewerker verbonden is

**AFDELINGEN****ANR**

Uniek afdelingsnummer

**PK****NAAM**

Naam van de afdeling

**LOCATIE**

Plaats waar de afdeling gevestigd is

**HOOFD**

Medewerkersnummer van het hoofd van de

**FK**

afdeling

**SCHALEN****SNR**

Uniek nummer van een salarisschaal

**PK**

36

ONDERGRENS Laagste salaris dat tot de schaal behoort

BOVENGRENS Hoogste salaris dat tot de schaal behoortNetto

maandelijkse

**TOELAGE**

toelage op het salaris

**CURSUSSEN****CODE**

Code, die voor iedere cursus uniek is

**PK**

**OMSCHRIJVING** Omschrijving van de cursusinhoud

**TYPE**

Indicatie van het type cursus (waarbij we onderscheiden: ALG, BLD en DSG)

**LENGTE**

De cursuslengte, uitgedrukt in dagen

**UITVOERINGEN****CURSUS**

Code van de cursus in kwestie

**FK/PK****BEGINDATUM**

Datum waarop de cursus begint

**PK****DOCENT**

De persoon die de cursus geeft

**FK**

LOCATIE

Plaats waar de cursus plaatsvindt

**INSCHRIJVINGEN**

CURSIST

Het medewerkersnummer van de cursist **FK/PK**

CURSUS

De code van de cursus

**FK/PK**

BEGINDATUM

Datum waarop de cursus begint

**PK**

EVALUATIE

Beoordeling van de deelnemer (geheel  
getal op een schaal 1-5)

**HISTORIE**

MNR

Medewerker nummer

**FK/PK**

**BEGINJAAR**

Jaartal, in vier cijfers

**BEGINDATUM** Begindatum van het tijdsinterval

**PK**

**EINDDATUM**

Einddatum van het tijdsinterval

**AFD**

Afdeling waarvoor gedurende het tijdsinterval is **FK**  
gewerkt

**OPMERKINGEN** Ruimte voor vrije tekst

**MAANDSAL**

Maandsalaris gedurende het tijdsinterval

Aan de beschrijving van de tabel *medewerkers* valt verder niet zo veel 37

toe te voegen. Aandacht verdient de kolom COMM: deze kolom is alleen van toepassing voor verkopers, en bevat daarom structureel ontbrekende informatie (in het geval van niet-verkopers). Bovendien wordt deze commissie op jaarbasis betaald, terwijl salarissen maandelijks worden overgemaakt. Denk er verder aan dat het maandsalaris ook nog een netto toelage kent, die afhankelijk is van de salarisschaal.

De structuur van de tabel AFDELINGEN spreekt voor zich. Let op het tweetal relaties dat tussen deze tabel en de medewerkerstabel bestaat: een medewerker kan ‘verbonden zijn aan’ een afdeling, en ‘hoofd zijn van’ een afdeling.

De salaris *schalen* zijn niet overlappend, wat in de praktijk vaak wel het geval

is. Hiervoor is gekozen vanwege de eenvoud: op deze manier valt een bepaald salaris altijd in één schaal. Bovendien is de monetaire eenheid in het midden gelaten. De netto toelage wordt – net als het salaris – op maandbasis uitgekeerd.

Wat betreft de tabellen *cursussen* en *uitvoeringen*: deze namen zijn met opzet gekozen om het verschil tussen een *cursus* (generiek) en een *cursusuitvoering* (specifiek) zo duidelijk mogelijk aan te geven. In de wandeling wordt voor beide begrippen vaak de term ‘cursus’

gehanteerd.

We onderscheiden de volgende drie cursustypes:

### **ALG**

(algemeen)

Introductiecursussen

### **BLD**

(build)

Applicatiebouw

### **DSG**

(design)

Systeemanalyse en ontwerp

Dat betekent dat we alleen deze drie waarden toestaan voor de CURSUSTYPE-kolom; dit is een voorbeeld van een *attribuutconstraint*.

We hadden overigens in ons diagram ook een extra entiteit CURSUSTYPE kunnen modelleren; dan was deze kolom een refererende sleutel geworden naar een achtste tabel.

Er dienen procedures te worden vastgelegd die regelen hoe er met *historische gegevens* in een informatiesysteem moet worden omgegaan.

Dit is een belangrijk – in de praktijk beslist niet altijd even eenvoudig – onderdeel van het systeemontwerp. In onze casus is het vooral 38 interessant om wat dit betreft te kijken naar cursusuitvoeringen en inschrijvingen.

Als een geplande cursusuitvoering niet doorgaat, bijvoorbeeld vanwege gebrek aan inschrijvingen, blijft hij toch in de tabel staan (omwille van de statistiek). Zodoende kan het voorkomen dat docent en/of locatie niet zijn ingevuld; dat is natuurlijk pas relevant als een geplande cursus lijkt te gaan plaatsvinden.

Inschrijvingen worden in deze database opgevat als synoniem voor cursusdeelnames. Dat kunnen we bijvoorbeeld opmaken uit de kolom EVALUATIE van de INSCHRIJVINGEN-tabel, waarin de waardering van de deelnemer na afloop van de cursus wordt vastgelegd. Dit wordt uitgedrukt op een schaal van 1 tot 5, in betekenis oplopend van ‘slecht’

tot ‘uitstekend’. Als een inschrijving geannuleerd wordt, verwijderen we de bijbehorende rij uit de tabel. Als de begindatum van een zekere inschrijving in het verleden ligt, dan geeft dat dus per definitie aan dat de bewuste cursus ook daadwerkelijk is gevolgd.

In de HISTORIE-tabel worden gegevens vastgelegd met betrekking tot het arbeidsverleden van alle medewerkers, vanaf de datum van indiensttreding. Iedere wijziging van afdeling en/of maandsalaris wordt geregistreerd; de huidige afdeling en het huidige maandsalaris worden ook in deze tabel opgeslagen, waarbij het attribuut einddatum leeg wordt gelaten. Er is ook ruimte voor opmerkingen.

Van alle tabellen is behalve de structuur ook de inhoud opgenomen in appendix A.

## **Hoofdstuk 2**

### **Kennismaking met SQL, SQL\*Plus en SQL Developer**

In de eerste paragrafen van dit hoofdstuk wordt een globaal overzicht van de taal SQL gegeven, en daarmee een eerste indruk van de mogelijkheden van deze taal.

Vervolgens worden enkele basisbegrippen geïntroduceerd, die voortdurend een rol spelen bij de gedetailleerde behandeling van SQL-commando's. Het gaat daarbij om termen als: constante, variabele, expressie, conditie, functie, operator, operand, etc.

Ten slotte volgt een eerste kennismaking met de tools SQL\*Plus en SQL Developer, die we zullen gebruiken om de taal SQL te hanteren. Met gereedschap moet je leren omgaan, teneinde er optimaal plezier van te hebben.

In dit hoofdstuk komen voor het eerst praktische oefeningen aan de orde. Daartoe is toegang nodig tot een Oracle-database, en een schema waarin de zeven casustabellen zijn geïnstalleerd. Op de website van de uitgever ([www.academicservice.nl](http://www.academicservice.nl)) staan de scripts om dit schema op te zetten of zo nodig te verversen.

Uitgangspunt is steeds dat Oracle ‘in de lucht’ is; het starten en afsluiten van de database op een server is vaak een taak van de systeembeheerder, en kan per omgeving nogal verschillend zijn geïmplementeerd. Over het algemeen zal dit gebeuren met behulp van Oracle Enterprise Manager, een tool voor databasebeheer. We gaan daar verder niet op in.

#### **2.1**

##### **Overzicht SQL**

SQL (de afkorting staat voor Structured Query Language) is een taal die in principe op verschillende manieren kan worden toegepast: *interactief* en

*embedded.*

Interactief betekent dat men via een toetsenbord SQL-commando's ingeeft, waarvan het antwoord op het scherm verschijnt; onder 40

embedded SQL verstaat men het gebruik van SQL-commando's binnen een programma in een andere programmeertaal (zoals C, .Net of Java).

In dit boek beperken wij ons tot het interactief gebruik van SQL.

Hoewel SQL een *vraagtaal* wordt genoemd, zijn de mogelijkheden bepaald uitgebreider dan deze term doet vermoeden. De taal SQL wordt over het algemeen onderverdeeld in vier taalonderdelen:

1

datadefinitie (Data Definition Language, DDL);

2

datamanipulatie (Data Manipulation Language, DML);

3

raadpleging (Retrieval);

4

beveiliging.

### **2.1.1**

#### **Datadefinitie**

Met behulp van datadefinitie-commando's kan een databasestructuur worden gebouwd, gewijzigd, of afgebroken. Een databasestructuur kan bestaan uit allerlei objecten: tabellen, views, indexen, synoniemen, ...

Vrijwel alle datadefinitie-commando's van SQL beginnen met een van de

volgende drie sleutelwoorden:

**CREATE** Maak een nieuw databaseobject

**ALTER**

Wijzig iets aan de structuur van een bestaand databaseobject **DROP**

Verwijder een databaseobject

Met het commando CREATE VIEW kan bijvoorbeeld een view worden gedefinieerd; met ALTER TABLE kan een tabelstructuur worden veranderd (door bijvoorbeeld een extra kolom aan de tabel toe te voegen); met het commando DROP INDEX kan een index worden

verwijderd.

Een van de sterke kanten van een relationeel DBMS is dat een tabel met het commando ALTER TABLE van structuur veranderd kan worden, zónder dat bestaande databaseapplicaties als gevolg daarvan hoeven te worden aangepast. Dit verschijnsel wordt ook wel *logische gegevensafhankelijkheid* genoemd.

## 2.1.2

### Datamanipulatie

41

Zoals datadefinitie-commando's de *structuur* van de database onder handen nemen, zo bewerken datamanipulatie-commando's de *inhoud* van de database. Daartoe zijn in de basis drie SQL-commando's beschikbaar:

**INSERT**

Voeg nieuwe rijen aan een tabel toe

**UPDATE**

Wijzig bestaande rijen in een tabel

## **DELETE**

Verwijder bestaande rijen uit een tabel

In hoofdstuk 6 zullen we zien dat Oracle twee additionele datamanipulatiecommando's heeft:

## **MERGE**

Voegt conditioneel nieuwe rijen toe, of wijzigt rijen in een tabel

## **INSERT**

Voeg nieuwe rijen aan meerdere tabellen tegelijk toe

## **ALL**

Het INSERT-commando kan op twee manieren gebruikt worden: er wordt één rij toegevoegd aan de tabel door een serie kolomwaarden op te geven in de VALUES-clausule, óf we voegen (potentieel) meerdere rijen tegelijk toe met behulp van een selectie op bestaande databasegegevens (subquery).

Datamanipulatie-commando's worden altijd opgevat als onderdeel van een *transactie*. Dat heeft onder meer tot gevolg dat alle daarmee aangebrachte wijzigingen in de database een voorlopig karakter krijgen en alleen maar zichtbaar zijn binnen onze *sessie*. Sessies van andere interactieve gebruikers en/of databaseapplicaties zien onze aangebrachte wijzigingen niet totdat we de transactie bevestigen of annuleren.

Er zijn aparte SQL-commando's om een transactie te bevestigen of annuleren:

## **COMMIT**

Bevestig de wijzigingen van de huidige transactie

## **ROLLBACK**

Annuleer de wijzigingen van de huidige transactie

Let op de volgende verschillen tussen datadefinitie en datamanipulatie:

- 1 Met DELETE kunnen we een tabel leegmaken, met DROP verwijderen we een tabel.

2

Met UPDATE kunnen we de tabelinhoud wijzigen, met ALTER de structuur.

3

Datadefinitie is onherroepelijk (geen mogelijkheid tot annuleren), terwijl gevolgen van datamanipulatie altijd te herstellen zijn (met ROLLBACK).

In hoofdstuk 6 komen we nog iets uitgebreider op datamanipulatie terug. In hoofdstuk 7 komt nog het TRUNCATE-commando ter sprake, waarmee we op een efficiënte (maar onherroepelijke) wijze alle rijen van een tabel kunnen verwijderen.

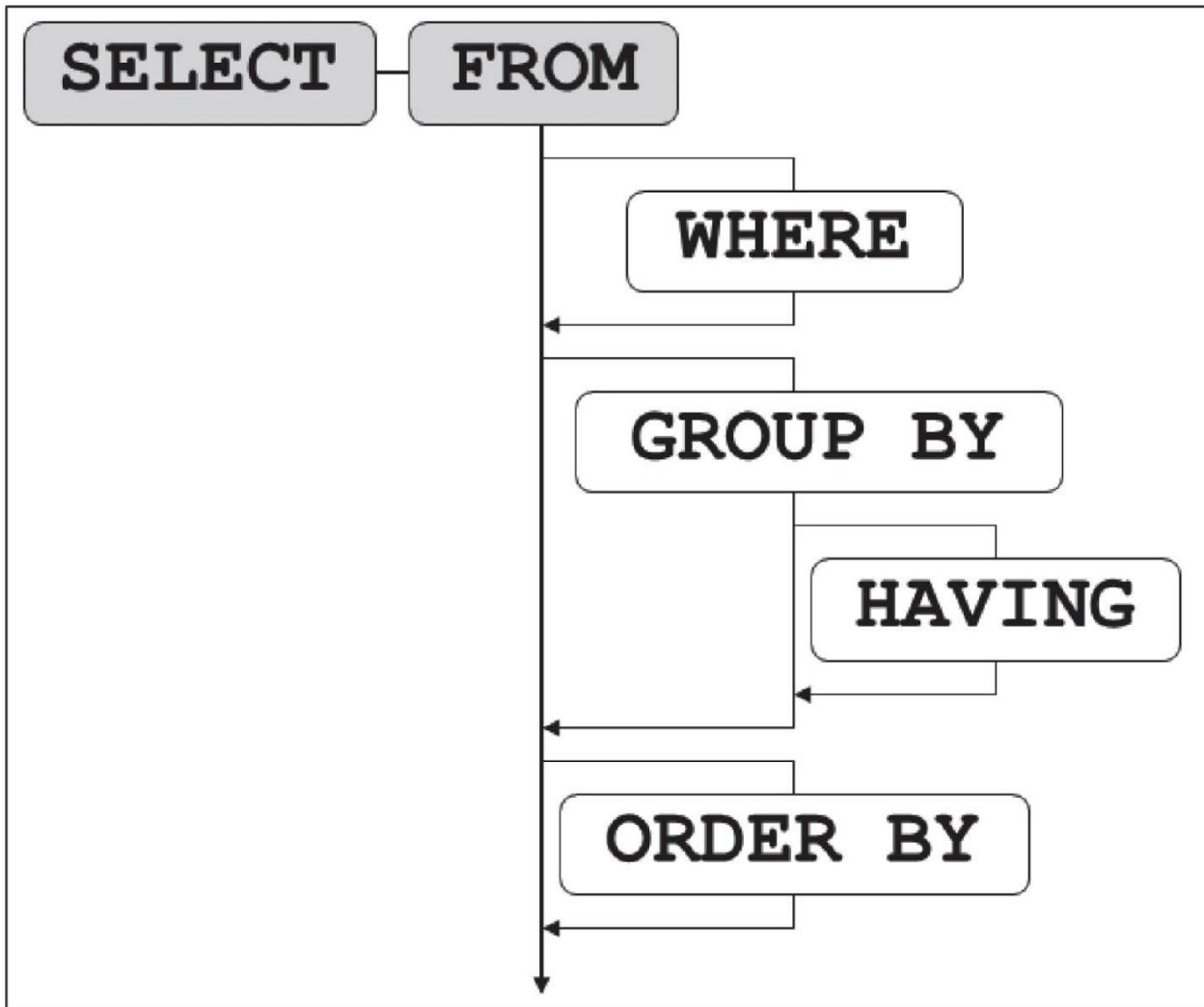
### 2.1.3

#### Raadpleging

Het enige SQL-commando om databasegegevens te raadplegen is SELECT. Dit commando stelt ons in staat om op allerlei manieren rijen uit één of meerdere tabellen te raadplegen. Een raadpleging die we middels een SELECT commando kunnen laten uitvoeren, noemen we vaak een *query*. Het resultaat van een query kunnen we beschouwen als een tabel; het is immers een verzameling van rijen.

Het SELECT-commando (van de ANSI/ISO SQL-standaard) heeft in totaal zes hoofdcomponenten, waarmee in principe alle SQL-operatoren ten behoeve van raadpleging gerealiseerd worden. Een schema van deze componenten ziet eruit als in figuur 2.1:





**Figuur 2.1**

De lijnen in dit schema geven als een soort ‘rangeerterrein’ de mogelijke commando-samenstellingen aan. De volgende regels zijn eruit af te lezen:

De volgorde van de componenten ligt altijd vast.

De SELECT- en FROM-componenten zijn verplicht.

De WHERE-component is naar keuze (optioneel).

De GROUP BY-component is optioneel.

HAVING komt nooit zonder GROUP BY voor.

De ORDER BY-component is optioneel.

Wat betreft de werking van deze zes componenten van het SELECT-commando geven we hier slechts een summiere aanduiding:

### **FROM**

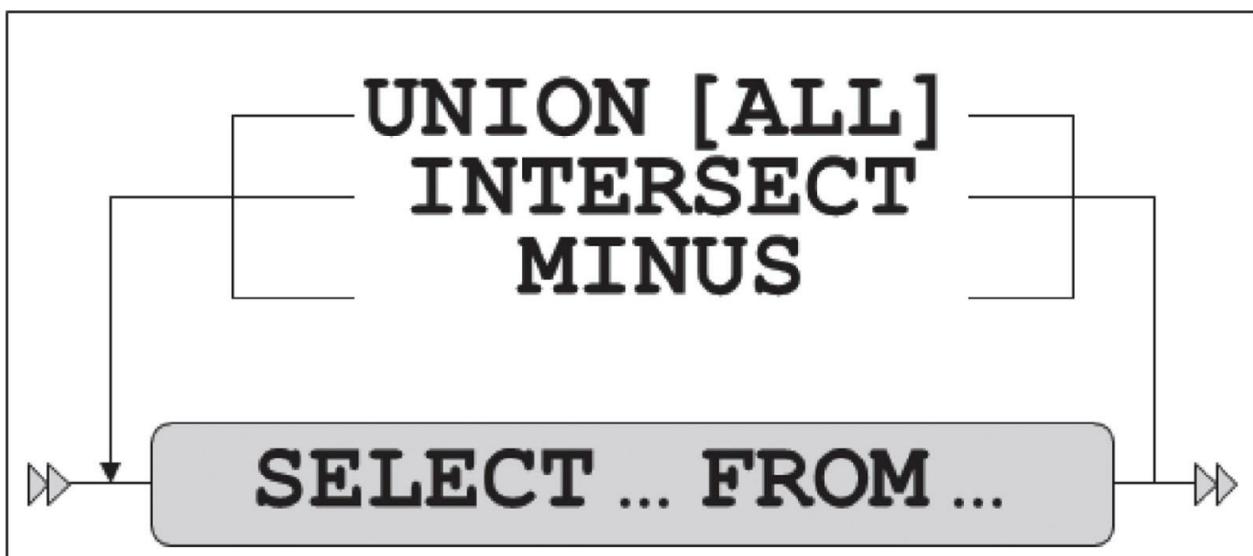
Welke tabel(len) moet(en) worden geraadpleegd.

### **WHERE**

Waaraan moeten de rijen voldoen.

**GROUP BY** Waarop moeten de rijen worden gegroepeerd.

44



### **HAVING**

Waaraan moeten de groepen van rijen voldoen.

## **SELECT**

Welke kolommen willen we in het resultaat zien.

**ORDER BY** In welke volgorde worden de rijen in het resultaat verlangd.

Als we nu terugdenken aan de relationele operatoren uit het vorige hoofdstuk, dan kunnen we bijvoorbeeld vaststellen:

De SELECT-component werkt als projectie-operator.

Met de FROM-component wordt het (Cartesisch) product gerealiseerd.

De restrictie wordt verzorgd door de WHERE-component.

Nu we het toch over relationele operatoren hebben: de vereniging, de doorsnede en het verschil zijn ook in SQL geïmplementeerd. We kunnen ze gebruiken om de resultaten van meerdere queries met elkaar te combineren tot één resultaattabel ([zie figuur 2.2](#)). We komen hier in

[hoofdstuk 8 nog op terug](#), in de paragraaf ‘Verzamelingsoperatoren’.

## **Figuur 2.2**

### **2.1.4**

## **Beveiliging**

SQL biedt allerlei commando's ten behoeve van beveiliging.

Uitgebreide bespreking van deze materie valt buiten het bestek van dit boek; in deze paragraaf wordt slechts een summier overzicht gegeven.

Allereerst wordt de toegang tot de database geregeld via een gebruikersautorisatie, onder andere met behulp van een wachtwoord. De belangrijkste commando's om dit te doen zijn:

## **CREATE USER**

Definieer een nieuwe gebruiker

## **ALTER USER**

Wijzig een bestaande gebruiker

## **DROP USER**

Verwijder een gebruiker

Vervolgens kunnen we met behulp van privileges de toegang tot de gegevens tot in de details regelen, evenals de handelingen die mogen worden verricht. De SQL-commando's om dit te doen zijn:

## **GRANT**

Verleen bepaalde privileges aan een gebruiker

## **REVOKE**

Ontneem bepaalde privileges van een gebruiker

Oracle onderscheidt twee soorten privileges:

### *Objectprivileges*

Het recht om een bepaald object op een bepaalde manier te benaderen, bijvoorbeeld het SELECT-, INSERT- en UPDATE-privilege op een specifieke tabel.

### *Systeemprivileges*

Het recht om bepaalde acties te ondernemen, zoals CREATE SESSION (het recht om te mogen aanloggen) en CREATE TABLE (het recht om een tabel te mogen maken). Er zijn ongeveer 200 verschillende Oracle-systeemprivileges.

Zie de Oracle-documentatie voor verdere details.

Ten slotte kunnen privileges worden gegroepeerd in ‘rollen’, waardoor een zeer flexibele beveiliging mogelijk is. Een typische gang van zaken is:

1

**CREATE ROLE** rolnaam;

2

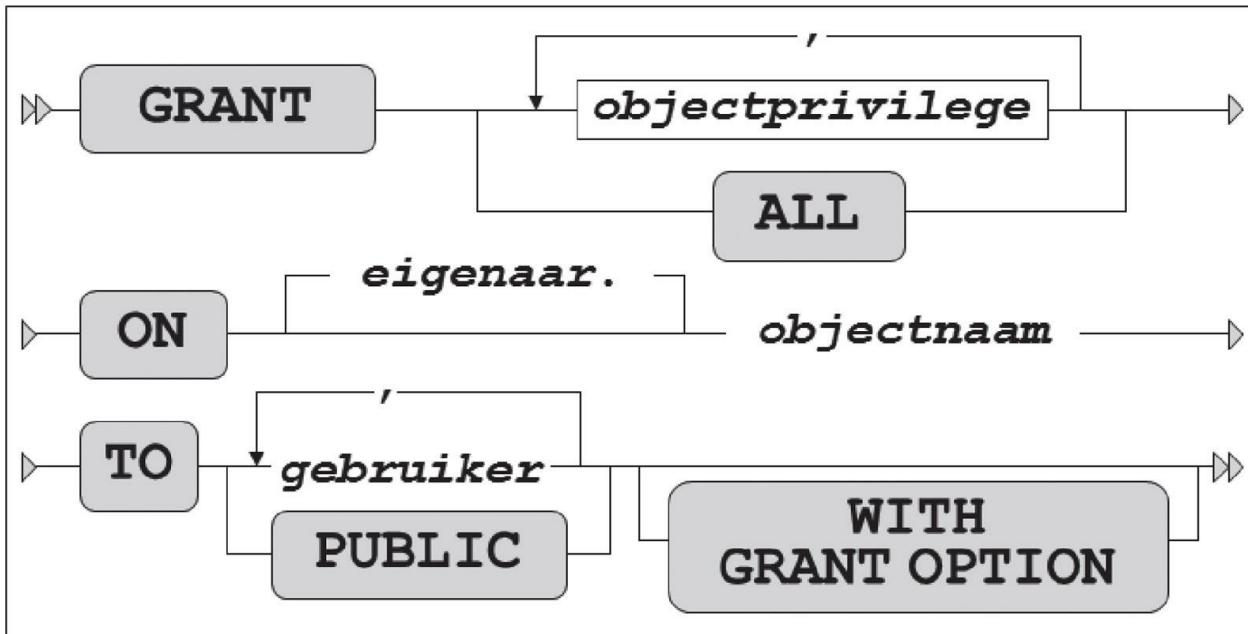
**GRANT** privileges **TO** rolnaam;

3

**GRANT** rolnaam **TO** gebruiker(s).

In de eerste stap wordt een nieuwe rol gedefinieerd. In de tweede stap vullen we de rol met een mix naar keuze van object- en

systeemprivileges. Ten slotte kan deze verzameling privileges in één commando aan gebruikers worden verleend. Rollen hebben een aantal prettige eigenschappen: ze zijn dynamisch (latere wijzigingen werken automatisch door), ze kunnen selectief worden ge(de)activeerd tijdens een sessie, en kunnen worden beveiligd met een wachtwoord. Het belangrijkste voordeel van rollen is beslist hun beheersbaarheid.



**Objectprivilege:** Geeft het recht op:

<b>SELECT</b>	Raadplegen van rijen in een tabel
<b>INSERT</b>	Invoeren van nieuwe rijen in een tabel
<b>UPDATE</b>	Wijziging van kolomwaarden van rijen in een tabel
<b>DELETE</b>	Verwijderen van rijen uit een tabel
<b>REFERENCES</b>	Verwijzing vanuit refererende sleutel naar unieke sleutel van een tabel
<b>EXECUTE</b>	Uitvoeren van functies of procedures
<b>ALTER</b>	Wijzigen van de structuur van een tabel
<b>INDEX</b>	Creëren van indexen op een tabel

Iedere tabel in de database heeft een eigenaar – degene die de tabel heeft gemaakt. De eigenaar heeft impliciet alle objectprivileges op zijn of haar tabel. In principe is het de eigenaar die bepaalt wat andere databasegebruikers met zijn of haar tabellen mogen doen. Daarom gaan we tot besluit van deze paragraaf over beveiliging nog iets verder in op objectprivileges. [Figuur 2.3](#) toont de syntax van het GRANT-commando, waarbij we systeemprivileges en rollen buiten beschouwing laten.

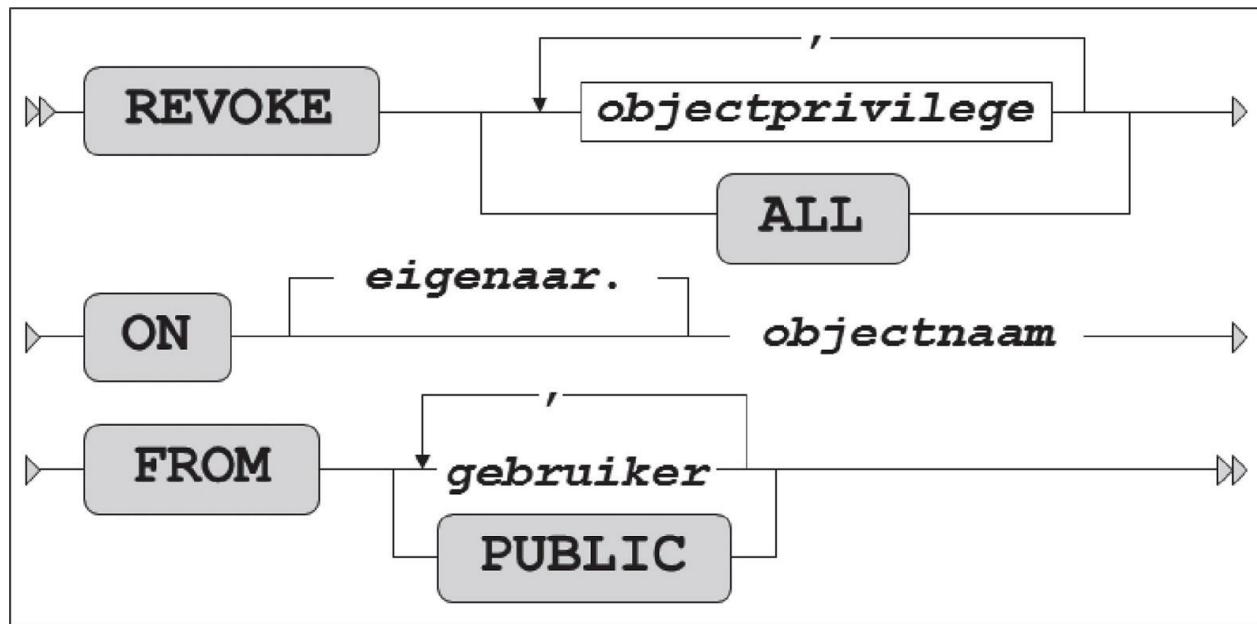
### Figuur 2.3

[In figuur 2.4 worden de belangrijkste Oracle-objectprivileges opgesomd, met](#)

een korte toelichting. Voor een vollediger overzicht: zie de Oracle-documentatie.

#### Figuur 2.4

47



Dan volgen hier nog een paar losse opmerkingen met betrekking tot het GRANT-commando:

1

Het recht om een tabel te verwijderen (DROP TABLE) kan door een eigenaar niet worden toegekend aan andere gebruikers. Overigens bestaat er wel een Oracle- systeem privilege DROP ANY TABLE.

2

Willen we alle objectprivileges voor een tabel ineens verlenen, dan kan dat met behulp van het sleutelwoord ALL (zie het syntaxdiagram [in figuur 2.3](#)).

3

We kunnen per GRANT-commando privileges verlenen aan één

gebruiker, aan enkele gebruikers, of aan alle gebruikers tegelijk. Het laatste gebeurt met behulp van de pseudo-gebruiker PUBLIC (zie het syntaxdiagram [in figuur 2.3](#)).

4

Het UPDATE-privilege kent desgewenst nog een verfijning: dit recht kan ook voor specifieke kolommen worden verleend, door

kolomnamen tussen haakjes te specificeren.

5

In principe is er geen onderscheid tussen tabellen en views. De privileges ALTER, INDEX en REFERENCES zijn echter betekenisloos op views.

6

De GRANT OPTION zorgt ervoor dat niet alleen bepaalde privileges worden verleend, maar daarbij ook het recht om deze privileges aan weer andere gebruikers door te geven.

De tegenhanger van GRANT is REVOKE. Het syntaxdiagram, dat verder voor zichzelf spreekt, ziet eruit als [in figuur 2.5](#).

## Figuur 2.5

48

- 
- 

Naast de twee in deze paragraaf genoemde standaard SQL-commando's (GRANT en REVOKE) worden door Oracle ook nog enkele andere commando's ondersteund, waarmee bijvoorbeeld invloed uitgeoefend kan worden op het locking-mechanisme van het DBMS, om auditing te plegen, en om gebruikersautorisatie te regelen. Bespreking van deze SQL-toevoegingen valt buiten het kader van dit boek.

## 2.2

### Enkele basisbegrippen

Een *constante* is ‘iets’ dat een vaste waarde heeft. We maken onderscheid tussen getallen (numerieke constanten) en tekst (alfanumerieke constanten). Een alfanumerieke constante wordt in het databasejargon ook wel een *string* genoemd.

Alfanumerieke constanten moeten in SQL altijd tussen enkele aanhalingstekens (single quotes) worden gezet.

Getallen zijn het eenvoudigst; ze worden in SQL altijd als zodanig herkend en opgevat. We kunnen desgewenst expliciet angeven dat de numerieke constanten moeten worden opgevat als drijvende-komma-

(floating point) -getallen door aan het einde een ‘F’ of een ‘D’ toe te voegen, respectievelijk om enkele of dubbele precisie aan te geven. Pas overigens op met het gebruik van decimale punten en komma’s in getallen, want de correcte interpretatie daarvan hangt af van een lokale instelling (NLS\_NUMERIC\_CHARACTERS).

Datums en tijdsduurindicaties vormen een probleem apart; ze worden meestal ingevoerd en weergegeven als een alfanumerieke constante.

Toch kunnen we in SQL alfanumerieke constanten die een datum voorstellen op een bijzondere manier angeven, door er een speciaal woord voor te zetten en ons vervolgens aan een strikte conventie te houden (zie de voorbeelden in [figuur 2.6](#)).

Bij het weergeven van datumconstanten hebben we drie keuzes: Als alfanumerieke constante weergeven en vertrouwen op impliciete conversie en interpretatie door Oracle, hetgeen mis kan gaan afhankelijk van de huidige NLS-instellingen (National Language Support).

Als alfanumerieke constante weergeven en er een TO\_DATE-conversiefunctie op toepassen waarin we explicet angeven hoe de 49

- <b>Numeriek:</b>	42
	8.75
- <b>Alfanumeriek:</b>	8.75F
	'Jansen'
	'ALG'
	'42'
- <b>Datums en intervallen:</b>	DATE '2004-02-09'
	TIMESTAMP '2004-02-09 11.42.59.00000'
	INTERVAL '2' SECOND
	INTERVAL '1-3' YEAR TO MONTH

datum moet worden geïnterpreteerd (zie [hoofdstuk 5](#)).

Als alfanumerieke constante weergeven, voorafgegaan door een van de woorden DATE, TIMESTAMP of INTERVAL. Als we INTERVAL

gebruiken moeten we na de alfanumerieke constante de grootheden aangeven, bijvoorbeeld DAY, MONTH of YEAR (zie de voorbeelden in

[figuur 2.6](#)).

Hier volgen enkele voorbeelden van SQL-constanten:

## Figuur 2.6

Let op het subtiele verschil tussen 42 en '42'! Het verschil tussen numerieke en alfanumerieke constanten komt met name tot uiting in hun bewerkingsmogelijkheden. Numerieke constanten kunnen bijvoorbeeld worden opgeteld, of met elkaar vermenigvuldigd. Met alfanumerieke constanten gaat dat niet; die kun je eigenlijk alleen maar aan elkaar plakken (concateneren).

De taal SQL is over het algemeen ongevoelig voor het verschil tussen *hoofd-* en *kleine letters*. Er is echter één belangrijke uitzondering, waar we terdege rekening mee moeten houden: deze ongevoelijheid geldt namelijk niet binnen strings. Dat betekent bijvoorbeeld dat 'Jansen'

niet gelijk is aan ‘JANSEN’. Dit wil nog wel eens de verklaring zijn van de melding ‘no rows selected’ in gevallen waarin we wel dégelijk resultaat verwachtten.

Een *variabele* is ‘iets’ dat verschillende waarden kan aannemen, of waarvan de waarde onbekend is. Een variabele heeft altijd een naam.

Wat SQL betreft zijn er twee soorten variabelen: *kolomnamen* en *systeemvariabelen*.

50

SYSDATE	De systeemdatum
CURRENT_DATE	De datum aan de applicatiekant
SYSTIMESTAMP	De systeemdatum en precieze tijd, met tijdzone-informatie
LOCALTIMESTAMP	Idem, aan de applicatiekant
USER	De naam waaronder we aangelogd zijn

#### **Rekenkundige operatoren**

- + Optellen
- Aftrekken
- \* Vermenigvuldigen
- / Delen

Kolomnamen spelen in SQL de rol van variabelen: de naam van een kolom blijft steeds hetzelfde, maar de waarde kan van rij tot rij variëren.

Systeemvariabelen hebben weinig tot niets met tabellen te maken, maar kunnen in SQL toch een belangrijke rol spelen. Ze worden ook wel *pseudokolommen* genoemd. [Zie figuur 2.7 voor een aantal](#) voorbeelden van zulke systeemvariabelen.

#### **Figuur 2.7**

Het verschil tussen datums (en tijdstippen) aan de *database*-kant en aan de *applicatie*-kant kan van belang zijn als we via een netwerkverbinding verbonden zijn met een database op een lokatie in een andere tijdszone.

Veel vergissingen in SQL ontstaan door het vergeten van aanhalingstekens. Bekijk het volgende SQL-fragment maar eens:

...WHERE PLAATS = UTRECHT...

PLAATS en UTRECHT zullen door Oracle beide worden opgevat als de naam van een variabele (kolomnaam), terwijl waarschijnlijk de bedoeling was:

...WHERE PLAATS = ‘UTRECHT’...

Een *operator* is een bewerking, terwijl een *operand* datgene is waarop de bewerking wordt losgelaten. We verdelen de SQL-operatoren in vier soorten, waarbij het type operand bepalend is. SQL kent vier rekenkundige operatoren:

51

<b>Alfanumerieke operatoren</b>	
	Concatenatie
<b>Vergelijgingsoperatoren</b>	
< Kleiner dan > Groter dan = Gelijk aan <= Kleiner dan of gelijk aan >= Groter dan of gelijk aan <> of != Ongelijk aan	

## Figuur 2.8

Rekenkundige operatoren kunnen alleen op getallen worden toegepast, op een uitzondering met betrekking tot datums na. Als je twee datums van elkaar aftrekt, krijg je als resultaat het verschil in dagen; bovendien kun je bij een datum met behulp van een getal een aantal dagen optellen.

Er is maar één alfanumerieke operator in SQL waarmee string-expressies aan

elkaar kunnen worden geplakt. Dit bescheiden aantal wordt ruimschoots gecompenseerd door de alfanumerieke functies, waarover meer [in hoofdstuk 5.](#)

## Figuur 2.9

De volgende categorie wordt gevormd door de bekende vergelijgingsoperatoren, waarmee we condities of voorwaarden kunnen formuleren:

## Figuur 2.10

Uitdrukkingen waarin deze operatoren voorkomen noemt men ook wel *voorwaarden, predikaten of condities*. Het zijn uitspraken die waar (TRUE) of onwaar (FALSE) zijn. Soms is het resultaat onbeslist (UNKNOWN), met name als er informatie ontbreekt. Hierop komen we terug in

[hoofdstuk 4, in paragraaf 4.8 ‘Null-waarden’.](#)

Er is ook nog een drietal operatoren dat op condities kan worden losgelaten: de logische operatoren. Bespreking voert op dit moment nog te ver, maar we vermelden ze vast:

52

### Logische operatoren

- |     |                                      |
|-----|--------------------------------------|
| AND | Logisch ‘en’                         |
| OR  | Logisch ‘of’ (de inclusieve variant) |
| NOT | Logische ontkenning                  |

<b>Expressie:</b>	<b>Datatype:</b>
3 + 4	Numeriek
naam  ', '  voorl	Alfanumeriek
plaats = 'Utrecht'	Logisch
12*sal > 20000 and comm >= 100	Logisch
gbdatum + interval '16' year	Datum
999	Numeriek

## ■ Figuur 2.11

Een *expressie* is een uitdrukking waarin variabelen, constanten en/of operatoren op een bepaalde manier met elkaar worden gecombineerd.

Net als constanten zijn expressies altijd van een bepaald datatype. Zie [figuur 2.12 voor een aantal](#) voorbeelden.

## ■ Figuur 2.12

Het laatste voorbeeld [in figuur 2.12](#) illustreert het feit dat de meest eenvoudige expressie wordt gevormd door een constante.

Als expressies gecompliceerd worden, kan het probleem van de *precedentie* optreden, ofte wel de vraag: welke operator heeft voorrang?

Daarvoor gelden in SQL bepaalde regels. Zo zullen rekenkundige operatoren bijvoorbeeld vóórgaan voor vergelijkingsoperatoren, die op hun beurt weer voorrang hebben op de logische operatoren. Het is raadzaam om in twijfelgevallen (ronde) haakjes te gebruiken; daarmee kan de volgorde worden afgedwongen, net als in de wiskunde.

Op het gebied van *functies* heeft Oracle veel aan de SQL-standaard toegevoegd. De uitdrukkingskracht van Oracle SQL is mede daardoor zo groot. Functies zijn te herkennen aan hun gedaante: ze hebben een naam, tussen haakjes gevolgd door één of meer argumenten, onderling gescheiden door komma's. Ze kunnen in expressies worden gebruikt, net als operatoren. We onderscheiden de volgende soorten functies: numerieke functies;

- 
- 
- 
- 
- 

AVG(MAANDSAL)	Het gemiddelde maandsalaris
SQRT(16)	De wortel uit 16
LENGTH(VOORL)	Het aantal voorletters
LOWER(NAAM)	De NAAM in kleine letters
SUBSTR(GBDATUM, 4, 3)	Drie karakters van GBDATUM, vanaf de vierde positie

alfanumerieke functies;

groepsfuncties;

datumfuncties;

conversiefuncties;

overige functies.

Als illustratie worden [in figuur 2.13 enige](#) voorbeelden gegeven: **Figuur 2.13**

Oracle biedt de mogelijkheid om zelf SQL-functies te definiëren, met behulp van de taal PL/SQL. [In hoofdstuk 5 zullen](#) we daarvan een voorbeeld zien.

In een database moeten voortdurend *namen* worden gegeven: aan tabellen, kolommen, views, indexen, synoniemen, etc. Over het algemeen beperken we ons daarbij tot het gebruik van letters, en eventueel het liggende streepje (de underscore:) ter verhoging van de leesbaarheid.

Houd er rekening mee dat er in dit geval géén onderscheid bestaat tussen hoofd- en kleine letters; intern worden alle namen van databaseobjecten naar hoofdletters vertaald en opgeslagen. We mogen ook cijfers gebruiken in namen, als de naam maar met een letter begint.

Namen mogen in Oracle niet langer zijn dan 30 karakters.

Uiteraard dienen databaseobjecten *verschillende* namen te hebben om ze uit elkaar te kunnen houden. Verschillende gebruikers mogen wél dezelfde namen gebruiken, omdat de combinatie naam/eigenaar een object binnen de database uniek maakt.

In SQL-commando's kunnen we eventueel *commentaar* opnemen, om nadere informatie of uitleg te verstrekken die niet tot de SQL-commando's zelf behoort, en wel op twee manieren: tussen /\* en \*/, of na twee mintekens. Bijvoorbeeld:

54

```
/* deze tekst zal als commentaar worden opgevat,  
   en dus door Oracle worden genegeerd */  
-- en dit trouwens ook
```

- 卷之三

Figuur 2.14

SQL heeft net als andere talen *geserveerde woorden*. Dat zijn woorden die – onder andere – niet als naam voor een databaseobject mogen worden gebruikt. Bijvoorbeeld: AND, CREATE, DROP, FROM, GRANT, HAVING, INDEX, INSERT, MODIFY, NOT, NULL, NUMBER, OR, ORDER, RENAME, REVOKE, SELECT, SYNONYM, SYSDATE, TABLE, UPDATE, USER, VALUES, VIEW, WHERE, ...

Zie de Oracle-documentatie (google “database object names and qualifiers”) voor verdere details wat naamgeving van database-objecten betreft. Een

volledig overzicht van gereserveerde woorden kan gevonden worden in een dictionary view (google “oracle reserved words”).

In deze paragraaf zijn de volgende begrippen behandeld:

constanten, variabelen;

operator en operand;

condities en expressies;

precedentie;

functies;

naamgeving databaseobjecten;

commentaar;

gereserveerde woorden.

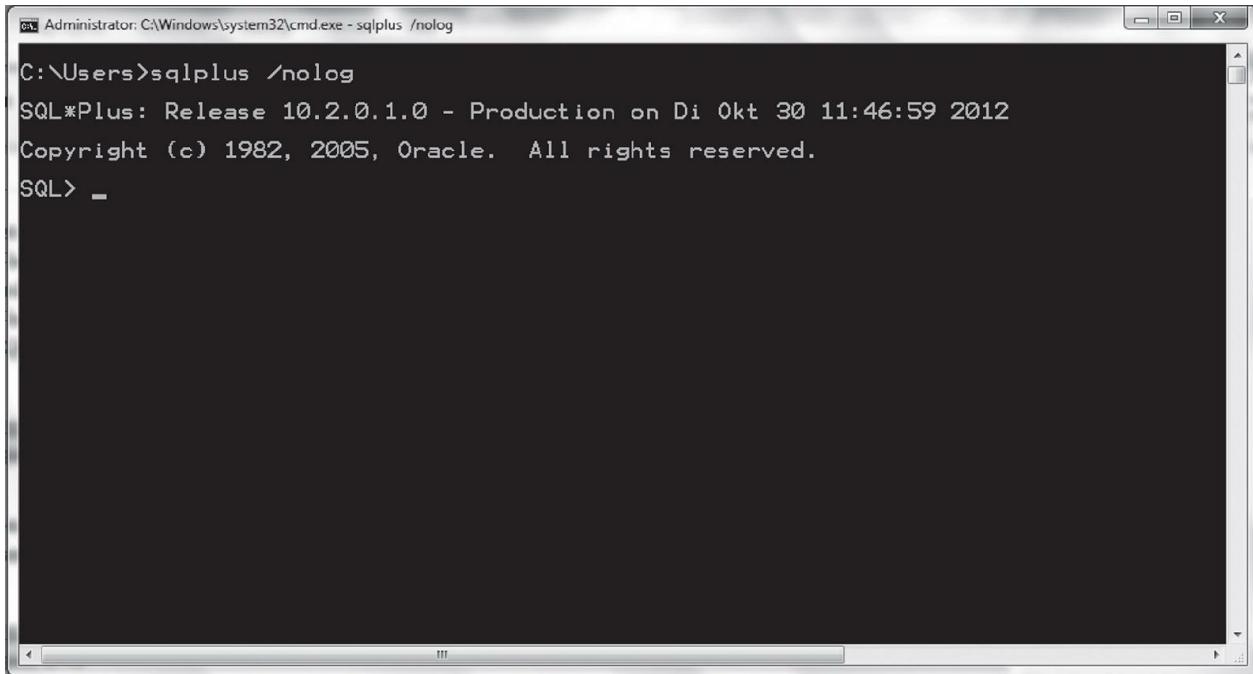
## 2.3

### Kennismaking met SQL\*Plus

SQL\*Plus is bij uitstek geschikt voor het uitvoeren van interactieve SQL commando's op een Oracle database. De kracht van SQL\*Plus ligt in de eenvoud van deze tool: je kunt er alleen maar interactieve SQL

mee uitvoeren. Het oogt daarom wel als een primitieve tool: het is een regelgeoriënteerde omgeving. Je biedt je commando aan, en krijgt daarna direct de output terug, waarna je een volgend commando kunt aanbieden. In de paragraaf hierna zullen we ook kennis maken met SQL

Developer. Deze tool is veel uitgebreider en biedt een grafische omgeving waarbinnen we ook interactieve SQL-commando's kunnen 55



A screenshot of a Windows command prompt window titled 'Administrator: C:\Windows\system32\cmd.exe - sqlplus /nolog'. The window displays the SQL\*Plus welcome message:

```
C:\Users>sqlplus /nolog
SQL*Plus: Release 10.2.0.1.0 - Production on Di Okt 30 11:46:59 2012
Copyright (c) 1982, 2005, Oracle. All rights reserved.
SQL> _
```

uitvoeren.

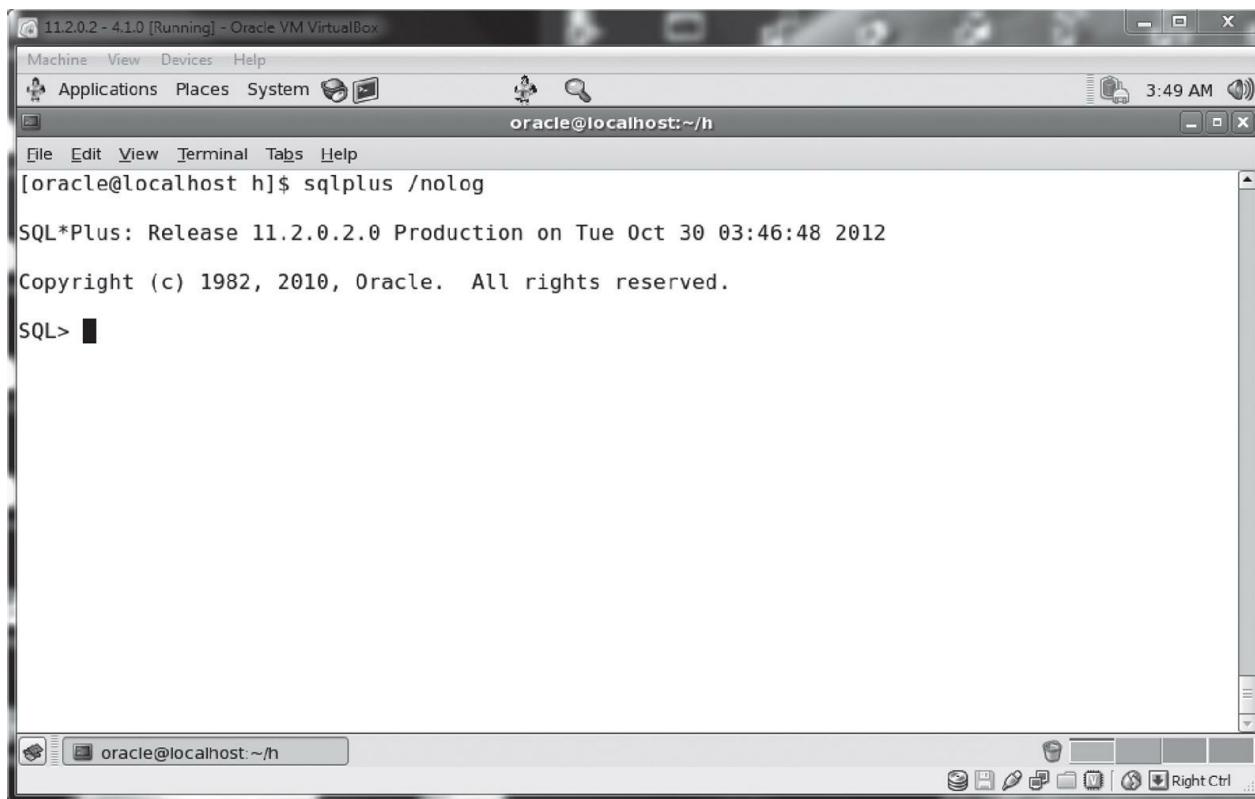
Als Oracle ‘in de lucht’ is, dan kan in een MS-Windows omgeving SQL\*Plus worden gestart door op een command-prompt (bijvoorbeeld in cmd.exe) het ‘sqlplus /nolog’ commando te geven.

Als het goed is meldt SQL\*Plus zich ([zie figuur 2.15](#)) en wordt de ‘SQL>’ prompt getoond.

### Figuur 2.15

Als we niet met Windows werken, maar met Linux, dan kunnen we in een shell-window (bijvoorbeeld een bash-shell) het ‘sqlplus /nolog’

commando geven ([zie figuur 2.16](#)).



The screenshot shows a terminal window titled 'oracle@localhost:~/h' running within an Oracle VM VirtualBox environment. The window has a standard Linux-style interface with a menu bar (File, Edit, View, Terminal, Tabs, Help) and a toolbar with icons for Applications, Places, System, and search. The main area displays the output of the SQL\*Plus command 'sqlplus /nolog'. The output includes the release information 'SQL\*Plus: Release 11.2.0.2.0 Production on Tue Oct 30 03:46:48 2012' and the copyright notice 'Copyright (c) 1982, 2010, Oracle. All rights reserved.' Below this, the prompt 'SQL>' is visible. The status bar at the bottom shows the connection details 'oracle@localhost:~/h' and various system icons.

**Figuur 2.16**

We hebben nu de SQL\*Plus-tool gestart, maar kunnen pas zinvol met SQL gaan werken nadat we inloggen aan een Oracle database. Dit doen we middels het ‘CONNECT’ commando: als we in staan zijn om een geldige gebruikersnaam/wachtwoord-combinatie te verstrekken,

[verschijnt op het scherm de melding dat we ‘Connected’ zijn. Zie figuur 2.17.](#)

```
[oracle@localhost h]$ sqlplus /nolog
SQL*Plus: Release 11.2.0.2.0 Production on Tue Oct 30 03:58:40 2012
Copyright (c) 1982, 2010, Oracle. All rights reserved.

SQL> connect boek/boek
Connected.
SQL>
```

**Figuur 2.17**

Met het commando EXIT of QUIT (gevolgd door de [Enter]-toets) kunnen we SQL\*Plus verlaten.

### 2.3.1

#### De SQL-buffer

We zijn nu in principe in staat om SQL-commando's in te geven en uit te laten voeren. SQL\*Plus 'begrijpt' niet alleen SQL, maar kent ook een groot aantal eigen SQL\*Plus-commando's. Deze SQL\*Plus-commando's moeten we duidelijk onderscheiden van SQL-

commando's, omdat ze door SQL\*Plus zelf worden uitgevoerd, en niet worden aangeboden aan de Oracle-database.

Een centraal begrip binnen SQL\*Plus is de *SQL-buffer*. Hierin wordt steeds het laatst ingegeven SQL-commando bewaard. Dat is handig als we bijvoorbeeld vergissingen maken; we kunnen de fout in de buffer herstellen

en het commando opnieuw laten uitvoeren.

SQL\*Plus-commando's komen niet in de buffer terecht, en laten de bufferinhoud dus intact. Ze worden over het algemeen onmiddellijk uitgevoerd nadat de [Enter]-toets is ingedrukt.

58

Als we een SQL-commando ingeven, wordt de buffer overschreven en

[zijn we het vorige SQL-commando kwijt. Binnenkort \(in paragraaf 2.3.4\) zullen we zien hoe we toch op eenvoudige wijze meerdere SQL-](#)

commando's kunnen bewaren voor hergebruik.

SQL-commando's beslaan meestal meerdere regels, die door SQL\*Plus voor ons tijdens het invoeren van regelnummers worden voorzien.

Willen we een SQL-commando afsluiten en onmiddellijk laten uitvoeren, dan moeten we de laatste regel afsluiten met een puntkomma (;). Deze puntkomma is geen onderdeel van het SQL-commando zelf, en zal ook niet in de buffer terechtkomen.

Als we de puntkomma vergeten (wat in het begin vast wel eens zal gebeuren), dan kunnen we dat op de volgende lege regel alsnog herstellen; als we echter op de lege regel nogmaals [Enter] ingeven, dan wordt het SQL-commando niet uitgevoerd, maar is het alleen maar in de buffer terechtgekomen. Het kan dan met behulp van een SQL\*Plus-

[commando alsnog worden uitgevoerd, zoals we zullen zien in paragraaf 2.3.3.](#)

We gaan ons in de volgende twee paragrafen bezighouden met de mogelijkheden om de inhoud van de SQL-buffer te wijzigen,

bijvoorbeeld om fouten te herstellen of een bestaand commando uit te breiden.

Daartoe geven we eerst een willekeurig SQL-commando in, afgesloten met een puntkomma:

```
SQL> select *
```

```
2 from medewerkers;
```

We krijgen nu alle gegevens te zien van de medewerkerstabel. De asterisk (\*) betekent ‘laat alle kolommen van de tabel zien’. Dit SQL-commando staat nu in de SQL-buffer, zoals blijkt als we het SQL\*Plus-commando LIST ingeven. Merk op dat de puntkomma niet nodig is, en dat SQL\*Plus-commando’s mogen worden afgekort – dit in

tegenstelling tot SQL-commando’s.

```

SQL> select *
2  from medewerkers;

      MNR NAAM        VOORL FUNCTIE      CHEF GBDATUM    MAANDSAL   COMM     AFD
----- -----  -----
7369  SMIT          N    TRAINER       7902 17-DEC-65      800      20
7499  ALDERS        JAM  VERKOPER      7698 20-FEB-61     1600      300     30
7521  DE WAARD      TF   VERKOPER      7698 22-FEB-62     1250      500     30
7566  JANSEN        JM   MANAGER      7839 02-APR-67     2975
7654  MARTENS       P    VERKOPER      7698 28-SEP-56     1250     1400     30
7698  BLAAK          R    MANAGER      7839 01-NOV-63     2850
7782  CLERCKX        AB   MANAGER      7839 09-JUN-65     2450
7788  SCHOTTEN       SCJ  TRAINER      7566 26-NOV-59     3000
7839  DE KONING      CC   DIRECTEUR    17-NOV-52     5000
7844  DEN DRAAIER    JJ   VERKOPER      7698 28-SEP-68     1500      0      30
7876  ADAMS          AA   TRAINER      7788 30-DEC-66     1100
7900  JANSEN          R   BOEKHOUDER  7698 03-DEC-69      800      30
7902  SPIJKER        MG   TRAINER      7566 13-FEB-59     3000
7934  MOLENAAR       TJA  BOEKHOUDER  7782 23-JAN-62     1300
14 rows selected.

SQL> L
1  select *
2* from medewerkers
SQL>

```

**Figuur 2.18**

### 2.3.2

#### Het gebruik van een externe editor

We gaan nu de inhoud van de buffer wijzigen (editen). Dat kan op twee manieren: met behulp van een externe editor, of met de editor van SQL\*Plus zelf. Het voordeel van de SQL\*Plus-editor is dat hij altijd beschikbaar is, en onafhankelijk is van het platform waarop wordt gewerkt. Daar staat tegenover dat de SQL\*Plus-editor niet zo gebruiksvriendelijk is als bijvoorbeeld Notepad onder Windows, of vi onder Linux. In deze paragraaf bespreken we eerst hoe met een externe editor kan worden gewerkt.

Het definiëren van de externe editor – of het opvragen welke editor het is – is mogelijk met behulp van het SQL\*Plus-commando **DEFINE**: **SQL> define**

**\_editor=Notepad**

SQL> **define \_editor**

DEFINE \_EDITOR = “Notepad” (CHAR)

SQL>

60

Merk op dat de variabele \_EDITOR heet; let op de underscore aan het begin van de naam.

De *externe editor* kan worden aangeroepen om de inhoud van de SQL-buffer aan te passen. SQL\*Plus biedt daartoe het SQL\*Plus-commando EDIT. Dat zal overigens alleen maar lukken als er een SQL-commando in de buffer staat; een lege buffer zal resulteren in de foutmelding

‘nothing to save’. Ook is het goed om te weten dat deze handelwijze een subprocess start dat steeds dient te worden verlaten om weer in SQL\*Plus verder te kunnen. We kunnen onder Windows/Linux ook een aparte Notepad/vi-sessie starten, zodat we eenvoudig tussen twee windows kunnen switchen. Dan moeten we wel steeds de inhoud van de buffer eerst naar disk schrijven.

### 2.3.3

#### **De SQL\*Plus-editor**

We beginnen weer met ons eerste SQL-commando in de SQL-buffer: SQL>  
select \*

2 from medewerkers;

Het is de bedoeling dat we de opdrachten in deze paragraaf letterlijk uitvoeren, ook al menen we fouten te zien (die zijn opzettelijk opgenomen). Om te beginnen is het belangrijk om te weten dat de editor van SQL\*Plus regelgeoriënteerd is – dat wil zeggen dat er steeds slechts één regel actueel is waarop kan worden gewijzigd. Deze regel wordt op het scherm gemarkeerd

met een asterisk (\*), en is normaal gesproken de laatst ingegeven regel; in ons voorbeeld dus de tweede regel.

Willen we nu de eerste regel wijzigen, dan moeten we die eerst activeren met het commando L1 (zie [figuur 2.19](#)). We gaan het sterretje vervangen door een tweetal kolomnamen. C is een afkorting van het SQL\*Plus-commando CHANGE. Op de actuele regel wordt gezocht naar het éérste voorkomen van een ‘\*’; dat karakter wordt veranderd in

‘naam, gebdatum’.

In plaats van schuine strepen (/) kunnen we ook een willekeurig ander karakter als scheidingsteken (separator) in het commando CHANGE gebruiken, en een spatie tussen de C en de eerste separator is niet nodig.

61

```
SQL> L1
1* select *

SQL> C/*/naam, gebdatum/
1* select naam, gebdatum

SQL> R
1 select naam, gebdatum
2* from medewerkers

select naam, gebdatum
*
ERROR at line 1:
ORA-00904: "GEBDATUM": invalid identifier

SQL> C/e//
1* slect naam, gebdatum
```

```
SQL> R
 1 select naam, gebdatum
 2* from medewerkers

select naam, gebdatum
*
ERROR at line 1:
ORA-00904: "GEBDATUM": invalid identifier

SQL> C/e//  

 1* slect naam, gebdatum
```

## Figuur 2.19

[We gaan nu de inhoud van de buffer opnieuw laten uitvoeren \(zie figuur 2.20\); dat doen we met het commando RUN \(afgekort met R\). Daarbij](#)

blijkt dat we zojuist iets fout hebben gedaan. Bekijk de foutmelding nauwkeurig; let op de indicatie van de regel waarop de fout is geconstateerd, en de plaats binnen die regel (wordt door het sterretje aangegeven). Let ook op de vergissing die we daarna maken bij een eerste poging tot correctie.

## Figuur 2.20

We hebben nu op regel 1 de eerstvoorkomende ‘e’ verwijderd, in plaats van de bedoelde ‘e’ in ‘gebdatum’. Dit is de manier waarop het commando CHANGE altijd werkt. We moeten dus opletten dat we niet te globale opdrachten geven; beter was in dit geval bijvoorbeeld geweest:

‘c/ge/g’.

```

SQL> L
  1 select naam, gbdatum
  2* from medewerkers
SQL> L1
  1* select naam, gbdatum
SQL> A , afd
  1* select naam, gbdatum, afd
SQL> L
  1 select naam, gbdatum, afd
  2* from medewerkers
SQL>

```

```

  1 select naam, gbdatum, afd
  2* from medewerkers
SQL> I
  3 where afd = 30;

```

NAAM	GBDATUM	AFD
ALDERS	20-FEB-61	30
DE WAARD	22-FEB-62	30
MARTENS	28-SEP-56	30
BLAAK	01-NOV-63	30
DEN DRAAIER	28-SEP-68	30
JANSEN	03-DEC-69	30

```
SQL>
```

We kunnen ook tekst aan het eind van een bestaande regel toevoegen ([zie figuur 2.21](#)). Denk eraan om eerst de gewenste regel te selecteren.

## Figuur 2.21

A staat voor APPEND. Let er wel op dat er normaal gesproken geen spatie wordt tussengevoegd. Dat is hier ook niet nodig; is dat wel het geval, dan moeten we een tweede spatie achter de A van APPEND meegeven.

Een of meer regels toevoegen kan ook, en wel met het commando INPUT

([zie figuur 2.22](#)). Toevoegen gebeurt in principe onder de actuele regel; is dat de laatste regel van de buffer, dan komt de nieuwe regel dus onderaan. Let op de regelnummering; afsluiten gaat op dezelfde manier als met het ingeven van commando's.

## Figuur 2.22

Voor alle duidelijkheid: de I staat voor INPUT en niet voor INSERT, want dat is een SQL-commando (om rijen aan een tabel toe te voegen). In dit 63

```
SQL> L
 1 select naam, gbdatum, afd
 2 from medewerkers
 3* where afd = 30
SQL> DEL
SQL> L
 1 select naam, gbdatum, afd
 2* from medewerkers
SQL>
```

geval hebben we de puntkomma gebruikt om het gewijzigde SQL-commando direct uit te voeren; gaat het om het tussenvoegen van een regel dan moeten we tweemaal [Enter] ingeven om de input mode te verlaten, waarna we het commando in de buffer kunnen uitvoeren met /

of RUN.

Willen we een of meer regels verwijderen, dan kan dat met het commando DEL ([zie figuur 2.23](#)).

## Figuur 2.23

DEL is overigens *niet* een afkorting van DELETE, want dat is een SQL-commando (om rijen van een tabel te verwijderen). Het SQL\*Plus-commando DEL verwijdert de actuele regel uit de buffer. We kunnen eventueel als argument een regelnummer meegeven, of een begin- en eindregelnummer om een reeks regels tegelijk te verwijderen.

Nog een handigheidje: we hebben al gezien dat we een regel actueel kunnen maken door het regelnummer in te geven (zonder de L van LIST). We kunnen een bestaande regel ook overschrijven zonder hem eerst te verwijderen, namelijk door de gewenste nieuwe regel in te tikken, voorafgegaan door zijn regelnummer ([zie figuur 2.24 en 2.25](#)).

## 64

```
SQL> L
 1 select code, omschrijving
 2 from cursussen
 3* where type = 'DSG'
SQL> 2
 2* from cursussen
SQL> 42
SP2-0226: Invalid line number
SQL> 1 select *
SQL> L
 1 select *
 2 from cursussen
 3* where type = 'DSG'
SQL>
```

```
 1 select *
 2 from cursussen
 3* where type = 'DSG'
SQL> 8 order by code
SQL> L
 1 select *
 2 from cursussen
 3 where type = 'DSG'
 4* order by code
SQL>
```

```
 1 select *
 2 from cursussen
 3 where type = 'DSG'
 4* order by code
SQL>
SQL> 0 /* dit is slechts commentaar */
SQL> L
 1 /* dit is slechts commentaar */
 2 select *
 3 from cursussen
 4 where type = 'DSG'
 5* order by code
SQL>
```

Figuur 2.24

## Figuur 2.25

Als we regelnummer 0 gebruiken, zijn we in staat om een eerste regel toe te voegen vóór de huidige eerste regel in de buffer (zie [figuur 2.26](#)).

## Figuur 2.26

65

```
SQL> select anr, omschrijving
  2  from afdelingen
  3 where locatie = 'SCHIERMONNIKOOG';
select anr, omschrijving
*
ERROR at line 1:
ORA-00904: "OMSCHRIJVING": invalid identifier
```

```
SQL> L*
 1* select anr, omschrijving
SQL> C/O.../naam
 1* select anr, naam
SQL> 3
 3* where locatie = 'SCHIERMONNIKOOG'
SQL> C/S...G/UTRECHT
 3* where locatie = 'UTRECHT'
SQL>
```

Bij het werken met CHANGE komt het gebruik van drie puntjes (*ellipsis*) soms van pas. De volgende voorbeelden (zie [figuur 2.27 en 2.28](#))

illustreren de werking hiervan. Let op: we maken eerst weer opzettelijk een fout.

## Figuur 2.27

Normaal gesproken is de laatst ingegeven regel altijd de actuele regel van de SQL-buffer. Treedt er echter een foutsituatie op, dan is dat de regel waarop de fout is geconstateerd; daardoor kan direct een correctie worden aangebracht zonder eerst de bewuste regel te hoeven selecteren (zie [figuur 2.28](#)).

Het sterretje in L\* betekent namelijk: laat de actuele regel zien.

### Figuur 2.28

Hiermee zijn alle editor-commando's van SQL\*Plus aan de orde geweest. Het is een simpele editor; het loont desondanks beslist de moeite om te leren ermee om te gaan. Dit komt ons vooral van pas als we met Oracle op een machine moeten werken die ons verder vreemd is; de SQL\*Plus-editor is namelijk identiek op alle machines waarop Oracle draait.

[Figuur 2.29 geeft](#) een overzicht van alle tot nu behandelde SQL\*Plus-66

<b>LIST</b>	Laat de hele buffer zien
<b>LIST n of n</b>	Maak regel <i>n</i> actueel
<b>CHANGE/oud/nieuw/</b>	Vervang eerste <i>oud</i> door <i>nieuw</i> op de actuele regel
<b>APPEND tekst</b>	Voeg <i>tekst</i> toe aan de actuele regel
<b>INPUT</b>	Voeg regel(s) toe onder de actuele regel
<b>DEL</b>	Verwijder actuele regel
<b>RUN of /</b>	Voer inhoud van de buffer uit
<b>EDIT</b>	Start een externe editor op de buffer
<b>DEFINE_EDITOR</b>	Definieer keuze van externe editor

commando's:

### Figuur 2.29

#### 2.3.4

#### Commando's bewaren

Zoals gezegd wordt de SQL-buffer steeds overschreven. Willen we de inhoud van de buffer bewaren, dan kan dat met behulp van het SQL\*Plus-commando SAVE. Op de achtergrond wordt dan een bestand gecreëerd, waarin de inhoud van de buffer wordt opgeslagen. Blijkt daarbij een bestand met de opgegeven naam al te bestaan, dan kunnen we met de opties APPEND of REPLACE aangeven wat onze bedoeling is.

De optie APPEND is met name handig als we al onze commando's in één

groot bestand willen opsparen – bijvoorbeeld om dat bestand later te kunnen afdrukken op een printer.

We tikken nu de volgende commando's in:

```

SQL> save BLA
Created file BLA.sql
SQL> select * from afdelingen;

ANR NAAM          LOCATIE      HOOFD
---- -----
 10 HOOFDKANTOOR    LEIDEN        7782
 20 OPLEIDINGEN    DE MEERN      7566
 30 VERKOOP         UTRECHT       7698
 40 PERSONEELSZAKEN GRONINGEN    7839

SQL> save BLI
Created file BLI.sql

SQL> select * from cursussen;

CODE OMSCHRIJVING          TYP LENGTE
----- -----
S02 Introductiecursus SQL    ALG     4
OAG Oracle voor applicatiegebruikers ALG     1
JAV Java voor Oracle-ontwikkelaars BLD     4
PLS Introductie PL/SQL        BLD     1
XML XML voor Oracle-ontwikkelaars BLD     2
ERM Datamodellering met ERM   DSG     3
PMT Procesmodelleringstechnieken DSG     1
RSO Relationale systeemontwerp DSG     2
PRO Prototyping              DSG     5
GEN Systeemgeneratie         DSG     4

10 rows selected.

SQL> save BLA
SP2-0540: File "BLA.sql" already exists.
Use "SAVE filename[.ext] REPLACE".

SQL> save BLA replace
Created file BLA.sql

SQL>

```

## Figuur 2.30

Let op de foutmelding bij de tweede SAVE BLA-poging; REPLACE of APPEND is noodzakelijk indien een bestand al blijkt te bestaan. We hebben nu dus twee bestanden gecreëerd, die overigens de

standaardextensie .SQL krijgen.

Dit soort bestanden kan met GET weer worden teruggehaald in de buffer, om ze bijvoorbeeld te wijzigen. Willen we de inhoud van het bestand ophalen en direct uitvoeren, dan kan dat met START ('get and run').

68

```
SQL> GET BLA
1* select * from cursussen
SQL> START BLI

ANR NAAM          LOCATIE      HOOFD
-----
10 HOOFDKANTOOR   LEIDEN       7782
20 OPLEIDINGEN    DE MEERN    7566
30 VERKOOP        UTRECHT     7698
40 PERSONEELSZAKEN GRONINGEN 7839

SQL>
```

```

SQL> L
 1* select * from afdelingen
SQL> @BLA

CODE OMSCHRIJVING          TYP LENGTE
----- -----
S02  Introductiecursus SQL    ALG   4
OAG  Oracle voor applicatiegebruikers ALG   1
JAV  Java voor Oracle-ontwikkelaars BLD   4
PLS  Introductie PL/SQL       BLD   1
XML  XML voor Oracle-ontwikkelaars BLD   2
ERM  Datamodellering met ERM    DSG   3
PMT  Procesmodelleringstechnieken DSG   1
RSO  Relationale systeemontwerp  DSG   2
PRO  Prototyping               DSG   5
GEN  Systeemgeneratie         DSG   4

10 rows selected.

SQL>

```

De commando's SAVE, GET en START kunnen in principe overweg met volledige bestandsspecificaties (met directorypad, naam en extensie) en gaan bij afwezigheid van padnamen uit van de huidige directory, en bij afwezigheid van een extensie uit van .SQL.

### Figuur 2.31

Zoals uit het volgende voorbeeld blijkt ([zie figuur 2.32](#)) mag in plaats van START als commando ook @ worden gebruikt.

### Figuur 2.32

#### 2.3.5

### **SQL\*Plus-instellingen**

```
SQL> set pagesize 22
SQL> set pause "Geef Enter... "
SQL> set pause on
SQL> run
 1* select * from cursussen
Geef Enter...
```

```
SQL> show pages
pagesize 22
SQL> show pause
PAUSE is ON and set to "Geef Enter... "
SQL> set pause off
SQL> show pause
PAUSE is OFF
SQL>
```

Het gedrag van SQL\*Plus als omgeving kan op talloze manieren worden beïnvloed met behulp van variabelen of instellingen (settings).

We komen daar nog op terug [in hoofdstuk 11](#), maar hier volgen vast een paar voorbeelden:

### Figuur 2.33

Hiermee wordt bereikt dat SQL\*Plus de uitvoer naar het beeldscherm doseert per pagina, in dit geval steeds 22 regels. Dit is vooral handig als er zoveel rijen in het resultaat voorkomen dat ze niet op een scherm passen.

Bestaande settings kunnen worden opgevraagd met het commando SHOW, en kunnen uiteraard ook worden uitgezet ([zie figuur 2.34](#)):

### Figuur 2.34

Middels ‘show all’ kunnen alle huidige settings bekijken worden.

Er is naast het SQL\*Plus-commando SET ook nog een SQL-commando om de omgeving aan te passen: ALTER SESSION. Hiermee kunnen bijvoorbeeld enkele NLS (National Language Support) -parameters worden geregeld, zoals:

## NLS\_DATE\_FORMAT

Standaard weergaveformaat voor datums

## NLS\_TIME\_FORMAT

Standaard weergaveformaat voor tijdstippen

70

```
SQL> alter session
  2  set    nls_date_format='dd-mm-yyyy'
  3      nls_language=Dutch
  4      nls_currency='Eur';
```

```
Session altered.
```

```
SQL>
```

## NLS\_LANGUAGE

De taal waarin SQL\*Plus boodschappen

geeft

NLS\_NUMERIC\_CHARACTERS De betekenis van de punt en de komma in getallen

## NLS\_CURRENCY

Het symbool om geldbedragen weer te

geven

De belangrijkste van deze parameters is wellicht NLS\_DATE\_FORMAT, omdat het manipuleren met datums nog wel eens verwarring en problemen oplevert.

## Figuur 2.35

Instellingen die we tijdens een sessie veranderen, blijven van kracht totdat we SQL\*Plus verlaten; bij het opnieuw starten van SQL\* Plus krijgen we de standaardinstellingen weer terug. We zouden dan bepaalde instellingen (steeds) opnieuw moeten ingeven. Willen we dat niet, dan kunnen we de bijbehorende commando's opnemen in een bestand met de naam login.sql. De commando's in dit bestand worden bij het starten van SQL\*Plus automatisch uitgevoerd.

login.sql is een voorbeeld van een *SQL\*Plus-script*; op dit soort bestanden komen we nog uitgebreid terug [in hoofdstuk 11](#).

Als een resultaat niet op een schermregel past, kan het handig zijn om met het commando COLUMN een of meer kolommen een beetje te versmallen. Normaal gesproken worden kolommen op het scherm even breed afgebeeld als ze zijn gedefinieerd in de database.

71

```
SQL> select * from cursussen
  2  where type = 'ALG';

CODE OMSCHRIJVING                      TYP    LENGTE
-----
S02  Introductiecursus SQL                ALG      4
OAG  Oracle voor applicatiegebruikers    ALG      1

SQL> COL omschrijving FORMAT a32
SQL> /

CODE OMSCHRIJVING                      TYP    LENGTE
-----
S02  Introductiecursus SQL                ALG      4
OAG  Oracle voor applicatiegebruikers    ALG      1

SQL>
```

```

SQL> select * from schalen
2 where snr > 3;

      SNR ONDERGRENNS BOVENGRENNS TOELAGE
----- ----- -----
        4      2001      3000      200
        5      3001      9999      500

SQL> COL toelage FOR 9999.99
SQL> /

      SNR ONDERGRENNS BOVENGRENNS TOELAGE
----- ----- -----
        4      2001      3000    200.00
        5      3001      9999    500.00

SQL>

```

## Figuur 2.36

In figuur 2.36 is te zien hoe alfanumerieke kolommen smaller op het scherm kunnen worden afgebeeld, met behulp van de optie FORMAT van het commando COLUMN. Alle SQL\*Plus-commando's (en eventuele onderdelen daarvan) mogen worden afgekort, zolang de afkorting uniek is; het COLUMN-commando kan dus worden afgekort tot COL, en FORMAT kan worden afgekort tot FOR (zie figuur 2.37). De breedte van numerieke kolommen kan op een vergelijkbare manier worden beïnvloed, zoals uit figuur 2.37 blijkt:

## Figuur 2.37

72

```

SQL> store set <bestandsnaam>[.sql] [replace|append]

```

```
SQL> spool BLA.TXT [create|replace|append]
SQL> select * from medewerkers;
...
SQL> select * from afdelingen;
...
SQL> spool off
```

Met het STORE SET-commando ([zie figuur 2.38](#)) kunnen ten slotte de actuele SQL\*Plus-settings in een bestand (een SQL\*Plus-script) worden opgeslagen:

### Figuur 2.38

Voor de duidelijkheid: de blokhaken [in figuur 2.38 \(zoals in \[.sql\]\)](#) geven commandoonderdelen aan die optioneel zijn, waarbij de verticale streep () eventuele keuzemogelijkheden (zoals bijvoorbeeld in

**[replace|append]**) van elkaar scheidt. De combinatie vet-cursief (zoals in <**bestandsnaam**>) geeft aan dat dit fragment niet letterlijk moet worden ingegeven, maar moet worden vervangen door een

bestandsnaam naar keuze.

Als we de SQL\*Plus-settings met behulp van het STORE SET-commando in een script hebben bewaard, dan kunnen we deze settings te allen tijde herstellen met behulp van het START- (of @) -commando.

### 2.3.6

#### Nog een paar nuttige SQL\*Plus-commando's

Het resultaat van een volledige SQL\*Plus-sessie kan worden vastgelegd in een bestand met behulp van het commando SPOOL ([zie figuur 2.39](#)):

### Figuur 2.39

Het bestand BLA.TXT, in dezelfde directory waarin het commando SAVE

zijn bestanden bewaart, zal nu een volledige kopie bevatten van alle schermuitvoer. We kunnen eventueel achter de bestandsnaam een van de opties CREATE, REPLACE, of APPEND specificeren om het gewenste gedrag van het SPOOL-commando preciezer aan te geven.

73

SQL> descr medewerkers	Name	Null?	Type
	MNR	NOT NULL	NUMBER(4)
	NAAM	NOT NULL	VARCHAR2(12)
	VOORL	NOT NULL	VARCHAR2(5)
	FUNCTIE		VARCHAR2(10)
	CIEF		NUMBER(4)
	GBDATUM	NOT NULL	DATE
	MAANDSAL	NOT NULL	NUMBER(6,2)
	COMM		NUMBER(6,2)
	AFD		NUMBER(2)

Bij het formuleren van SQL-commando's is het soms handig om even een overzicht te krijgen van de structuur van een tabel, om de kolomnamen en de datatypes te zien. Dan is het SQL\*Plus-commando DESCRIBE plezierig ([zie figuur 2.40](#)).

### Figuur 2.40

Het commando HOST stelt ons in staat om commando's uit te voeren op het onderliggende besturingssysteem; onder Windows wordt

bijvoorbeeld een DOS-command window gestart, onder Linux zal een shell command window gestart worden.

Afhankelijk van het operating system kunnen we dan met EXIT, LOGOUT of een vergelijkbaar commando om het betreffende window af te sluiten, weer terugkeren naar SQL\*Plus.

Nog een tweetal SQL\*Plus-commando's tot slot. Met CLEAR BUFFER

kan de SQL-buffer worden leeggemaakt – hetgeen niet zo vaak nodig is, omdat hij toch steeds wordt overschreven door een volgend commando; met CLEAR SCREEN wordt het SQL\*Plus-window leeggemaakt.

Overzicht van alle SQL\*Plus-commando's die tot nu toe behandeld zijn, naast de SQL\*Plus-editorcommando's die al eerder zijn opgesomd: **SAVE**

Bewaar de bufferinhoud in een bestand

## **GET**

Haal een bestand terug naar de buffer

## **START of @**

Voer de inhoud van een bestand uit

## **SPOOL**

Bewaar schermuitvoer in een bestand

## **SET**

Verander een SQL\*Plus-setting

74

## **SHOW**

Vraag SQL\*Plus-settings op

## **COLUMN ...**

Verander de kolomweergave op het scherm

## **FORMAT**

## **STORE SET**

Bewaar de SQL\*Plus-settings in een bestand

## **DESCRIBE**

Geef een beschrijving van een tabel

## **HOST**

Start een operating system sessie

## **CLEAR**

Maak de SQL-buffer leeg

## **BUFFER**

## **CLEAR**

Maak het scherm leeg

## **SCREEN**

We hebben ook nog een SQL-commando geïntroduceerd:

## **ALTER**

Wijzig (bijvoorbeeld) allerlei NLS-instellingen

## **SESSION**

Als we nu al nieuwsgierig zijn naar meer mogelijkheden van SQL\*Plus, dan kunnen we de Oracle-documentatie raadplegen, of bijlage B van dit boek (beschikbaar via de website).

## 2.4

### Kennismaking met SQL Developer

We geven in deze paragraaf een korte introductie van de SQL

Developer-tool. We kunnen deze tool gratis downloaden vanaf de Oracle Technology Network site (google “download sql developer”).

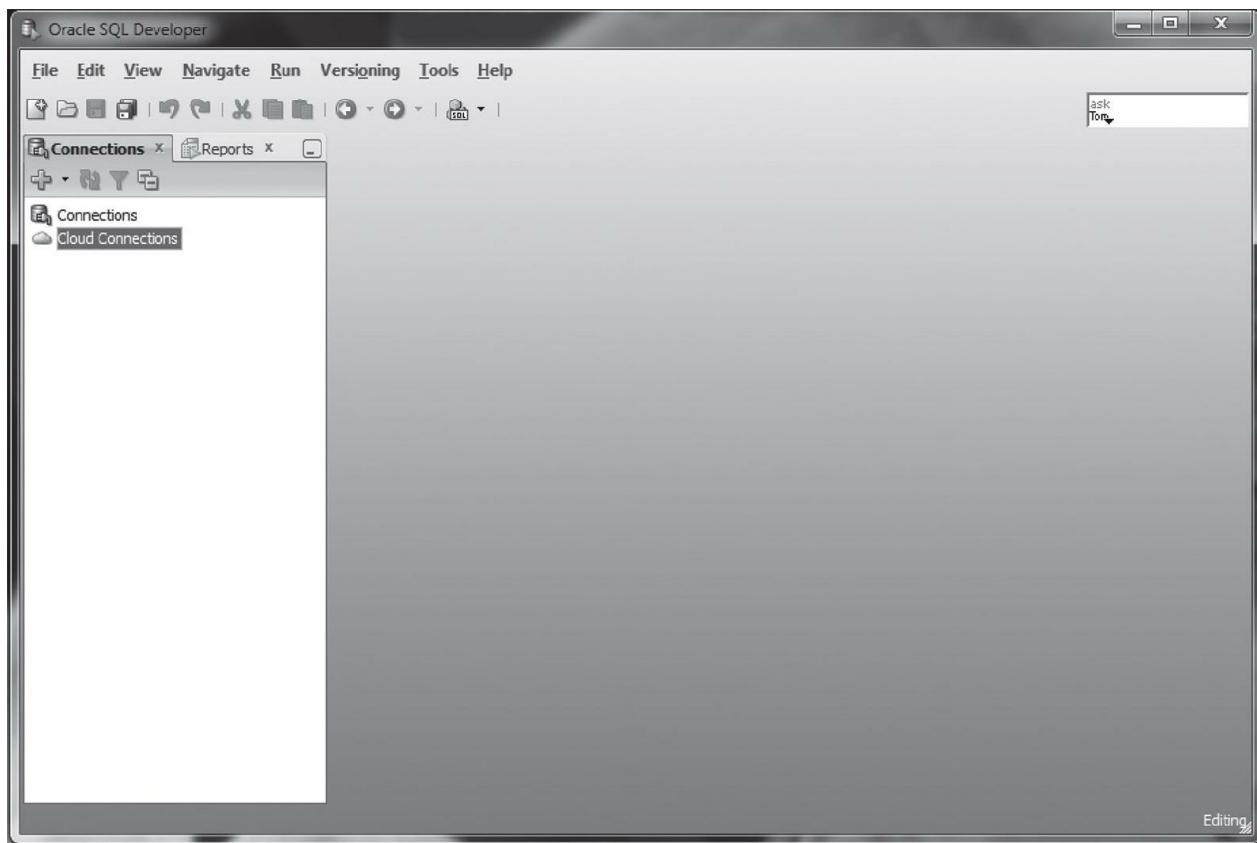
SQL Developer is een veelzijdige tool. Het kunnen uitvoeren van interactieve bevragingen is slechts één van de vele functies die SQL

Developer biedt. In deze paragraaf zullen we ons ook beperken tot slechts die functie.

Omdat SQL Developer een grafische tool is, in tegenstelling tot SQL\*Plus, dat regelgeoriënteerd is, zullen we zien dat SQL Developer eenvoudiger te bedienen en intuïtief te leren is. Nadat we SQL

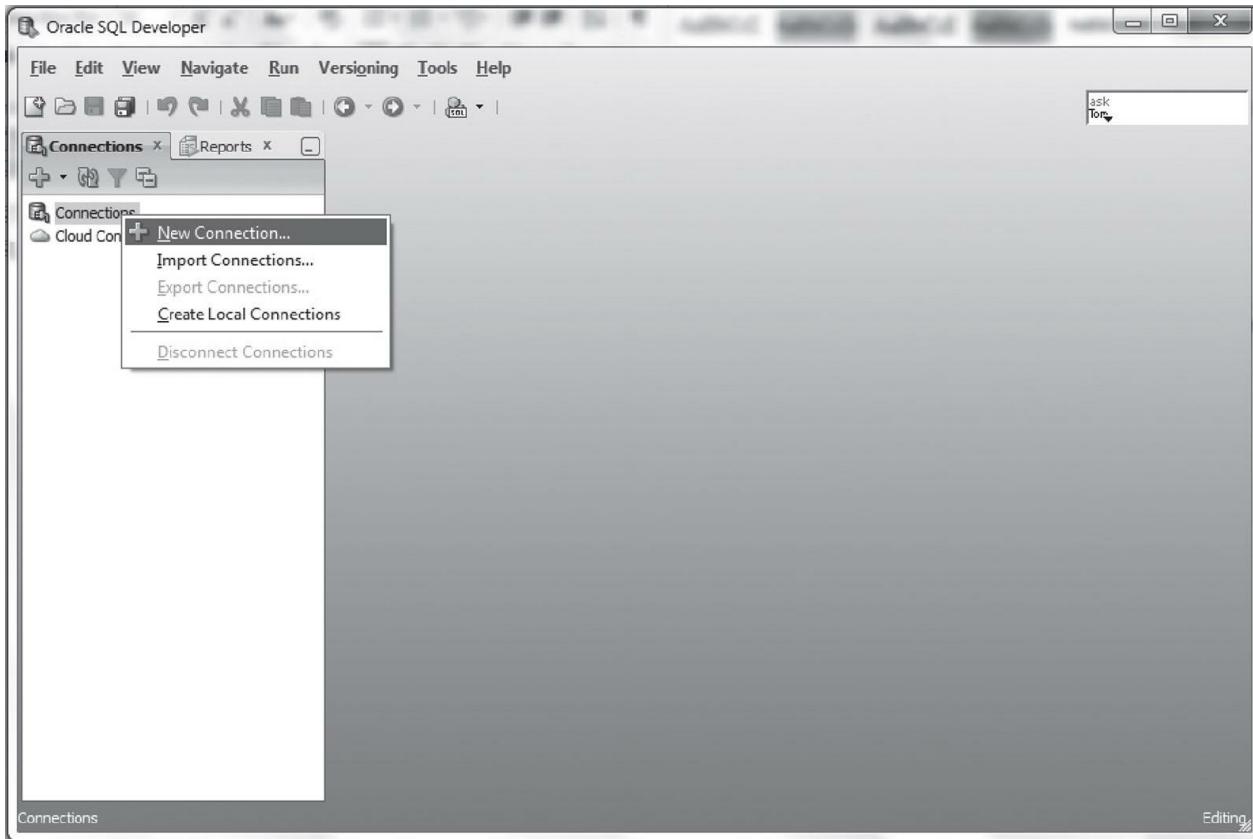
Developer hebben geïnstalleerd, kunnen we deze tool starten door eenvoudig het icoontje te dubbelklikken. Zie [figuur 2.41 voor het](#)

opstartscherms van SQL Developer.



**Figuur 2.41**

Net zoals bij SQL\*Plus zullen we ook hier eerst moeten inloggen aan een Oracle database, alvorens we met SQL kunnen gaan werken. Ga hiervoor in het Connections tab, op Connections staan, klik op de rechtermuisknop en kies vervolgens 'New Connection'. Zie [figuur 2.42](#).

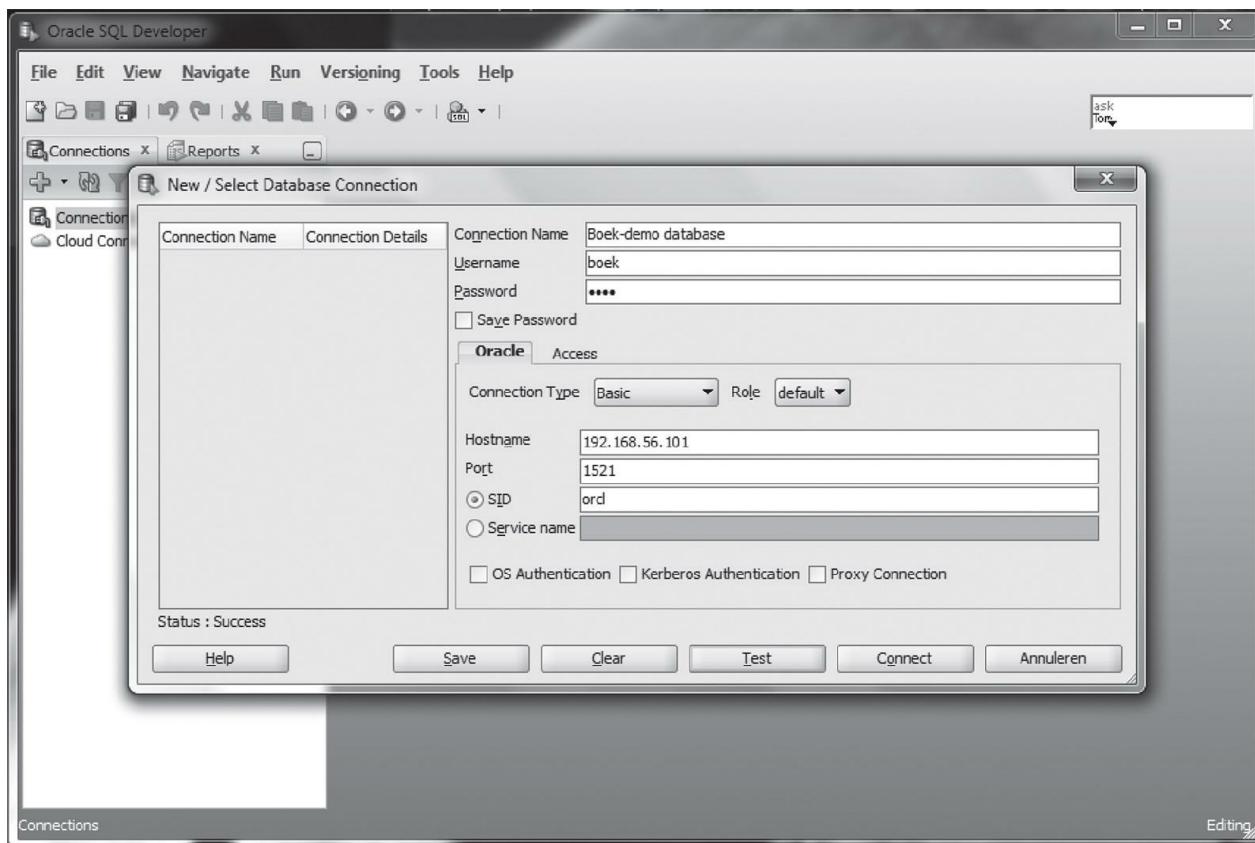


**Figuur 2.42**

We kunnen nu de connection-gegevens invullen. Achtereenvolgens vullen we in: Connection Name (een willekeurige naam voor deze connection), Username (gebruikersnaam) en Password (wachtwoord).

Daarna moeten we nog specificeren waar SQL Developer de Oracle-database kan vinden. Dit gaat middels een Hostname, een Port en een SID. Deze laatste drie gegevens kunnen verstrekken worden door de persoon die de Oracle-database heeft geïnstalleerd. Door op de Test-knop te drukken kunnen we verifiëren of onze connection werkt: er

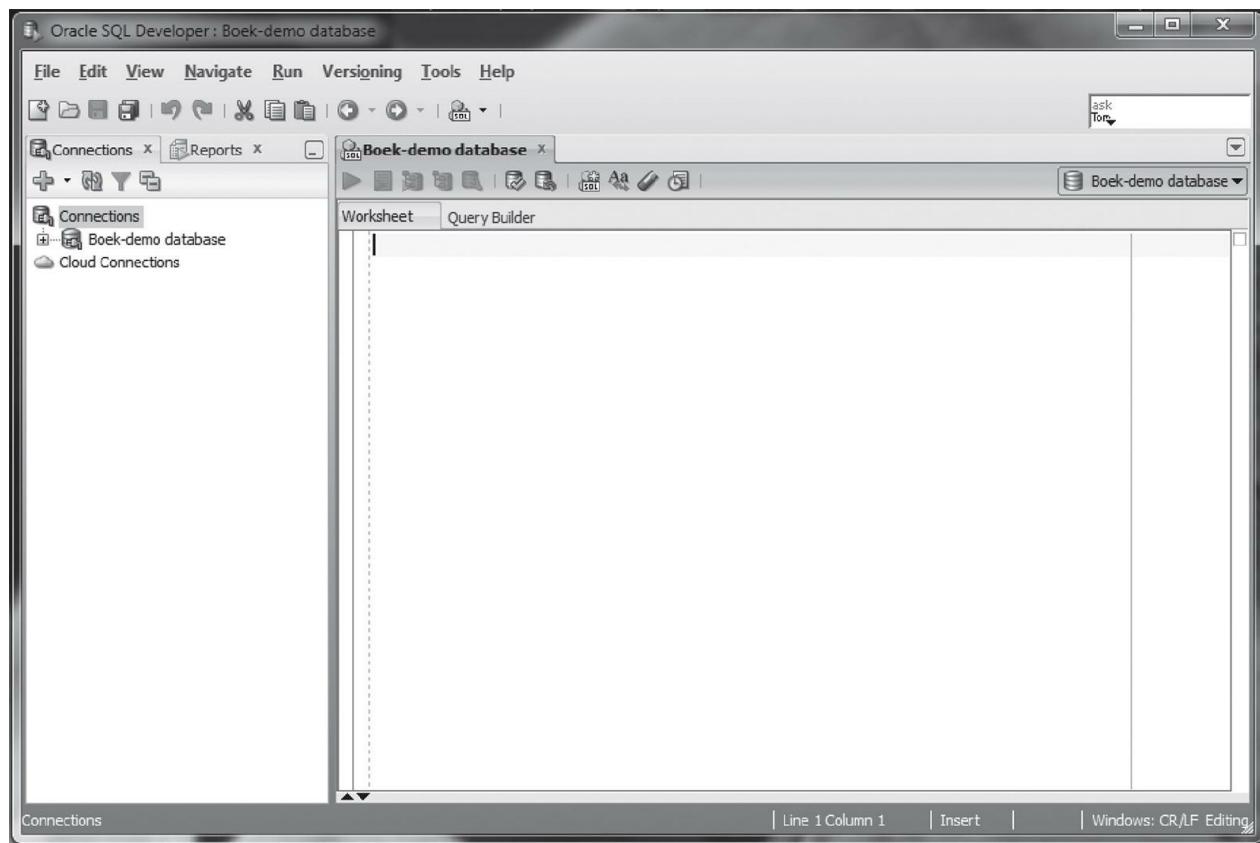
[verschijnt dan ‘Success’ linksonder, net boven de Help-knop. Zie figuur 2.43.](#)



Figuur 2.43

Als we nu op de Connect-knop drukken, loggen we in op de database, en zal de (SQL) Worksheet tab zich openen. [Zie figuur 2.44.](#)

Het Worksheet-gebied vormt feitelijk een tekst-editorgebied, zoals Notepad, waarin we onze SQL-commando's kunnen typen en aanpassen zoals we dat gewend zijn in een editor. SQL-commando's mogen op meerdere regels staan. We dienen ze af te sluiten met een puntkomma (;). Nadat we ze ingetypt hebben zijn er twee manieren om ze uit te voeren: door na de puntkomma een Control-Enter te geven of door op het meest linker icoontje (groene 'run'-driehoek) boven in het Worksheet-tab te drukken. Let op: voor beide manieren dient de cursor ergens in één van de regels van het SQL-commando te staan. Nadat het commando is uitgevoerd verschijnt onderin het 'Query Result'-tab, met daarin het resultaat van ons SQL-commando. [Zie figuur 2.45.](#)



The screenshot shows the Oracle SQL Developer interface. In the top-left corner, it says "Oracle SQL Developer : Boek-demo database". The menu bar includes File, Edit, View, Navigate, Run, Versioning, Tools, and Help. On the left, there's a Connections sidebar with "Boek-demo database" selected. The main area has a "Boek-demo database" tab open. The "Worksheet" tab is active, displaying the following SQL code:

```
select *  
from afdelingen;
```

Below the worksheet, the "Query Result" tab is active, showing the output of the query:

	ANR	NAAM	LOCATIE	HOOFD
1	10	HOOFDKANTOOR	LEIDEN	7782
2	20	OPLEIDINGEN	DE MEERN	7566
3	30	VERKOOP	UTRECHT	7698
4	40	PERSONEELSZAKEN	GRONINGEN	7839

At the bottom of the interface, status information includes "Line 2 Column 17", "Insert", "Modified", and "Windows: CR/LF Editing".

Figuur 2.44

Figuur 2.45

In de (SQL) Worksheet kunnen we meerdere SQL-commando's tegelijk 79

The screenshot shows the Oracle SQL Developer interface. In the top-left corner, there's a toolbar with various icons. Below it is a 'Connections' panel containing a connection to 'Boek-demo database'. The main workspace has a 'Worksheet' tab where two SQL statements are entered:

```
select *
from afdelingen;

select * from medewerkers;
```

Below the worksheet, the 'Query Result' tab is open, displaying the output of the second query. The results are presented in a table with the following columns:

	MNR	NAAM	VOORL	FUNCTIE	CHEF	GBDATUM	MAANDSAL	COMM	AFD
1	7369	SMIT	N	TRAINER	7902	17-12-65	800	(null)	20
2	7499	ALDERS	JAM	VERKOPER	7698	20-02-61	1600	300	30
3	7521	DE WAARD	TF	VERKOPER	7698	22-02-62	1250	500	30
4	7566	JANSEN	JM	MANAGER	7839	02-04-67	2975	(null)	20
5	7654	MARTENS	P	VERKOPER	7698	28-09-56	1250	1400	30
6	7698	BLAAK	R	MANAGER	7839	01-11-63	2850	(null)	30
7	7782	CLERCKX	AB	MANAGER	7839	09-06-65	2450	(null)	10
8	7788	SCHOTTEREN	SCJ	TRAINER	7566	26-11-59	3000	(null)	20

aanwezig hebben. Door steeds met de cursor ergens in het commando te gaan staan dat we willen uitvoeren en dan bijvoorbeeld weer op Control-Enter te drukken, kunnen we dat commando nogmaals laten uitvoeren. Zie [figuur 2.46 waarin](#) we de inhoud van de medewerkerstabel opgevraagd hebben.

## Figuur 2.46

We zien nu dat het Query Result tab, waar eerst nog het resultaat van ons eerste SQL commando in stond, ververst is met de inhoud ons tweede SQL commando. Als we het resultaat van een eerder SQL

commando niet verloren willen laten gaan, dan kunnen we het betreffende Query Result tab ‘vastpinnen’: dit doen we door op het rode pinnetje te drukken linksboven in het Query Result tab. Als we hierna weer een ander SQL commando uitvoeren, dan zal voor de output van

[dat commando een nieuw Query Result tab geopend worden. Zie figuur](#)

[2.47.](#)

Het is ook mogelijk om alle SQL-commando's die we in de Worksheet hebben staan, van boven naar onder, na elkaar uit te laten voeren en de output daarvan te tonen. Dit gaat middels de F5-toets, of door op het tweede icoontje linksboven in de Worksheet (het kleinere groene 80

The screenshot shows the Oracle SQL Developer interface. In the top-left corner, it says "Oracle SQL Developer : Boek-demo database". The menu bar includes File, Edit, View, Navigate, Run, Versioning, Tools, and Help. Below the menu is a toolbar with various icons. On the left, there's a Connections panel showing "Boek-demo database" and "Cloud Connections". The main workspace has a tab titled "Boek-demo database" which is currently active. It contains a "Worksheet" tab where the following SQL code is written:

```
select *  
from afdelingen;  
  
select * from medewerkers;  
  
select *  
from cursussen;
```

Below the worksheet is a "Query Result" tab showing the output of the last query. The output is a table with the following data:

	CODE	OMSCHRIJVING	TYPE	LENGTE
1	S02	Introductiecursus SQL	ALG	4
2	OAG	Oracle voor applicatiegebruikers	ALG	1
3	JAV	Java voor Oracle ontwikkelaars	BLD	4
4	PLS	Introductie PL/SQL	BLD	1
5	XML	XML voor Oracle ontwikkelaars	BLD	2
6	ERM	Datamodellering met ERM	DSG	3
7	PMT	Procesmodelleringstechnieken	DSG	1
8	RSO	Relationeel systeemontwerp	DSG	2

driehoekje, Run Script) te drukken. Er zal zich nu een ‘Script Output’-tab openen met daarin achtereenvolgens de resultaten van de SQL-commando’s. Zie [figuur 2.48](#).

**Figuur 2.47**

The screenshot shows the Oracle SQL Developer interface. In the top-left corner, the title bar reads 'Oracle SQL Developer : Boek-demo database'. Below it is a menu bar with File, Edit, View, Navigate, Run, Versioning, Tools, and Help. To the right is a toolbar with various icons. On the left, there's a 'Connections' sidebar with 'Boek-demo database' selected. The main area has tabs for 'Worksheet' and 'Query Builder', with 'Worksheet' currently active. The worksheet contains the following SQL code:

```
select *  
from afdelingen;  
  
select * from medewerkers;  
  
select *  
from cursussen;
```

Below the worksheet is a 'Query Result' tab showing the output of the third query:

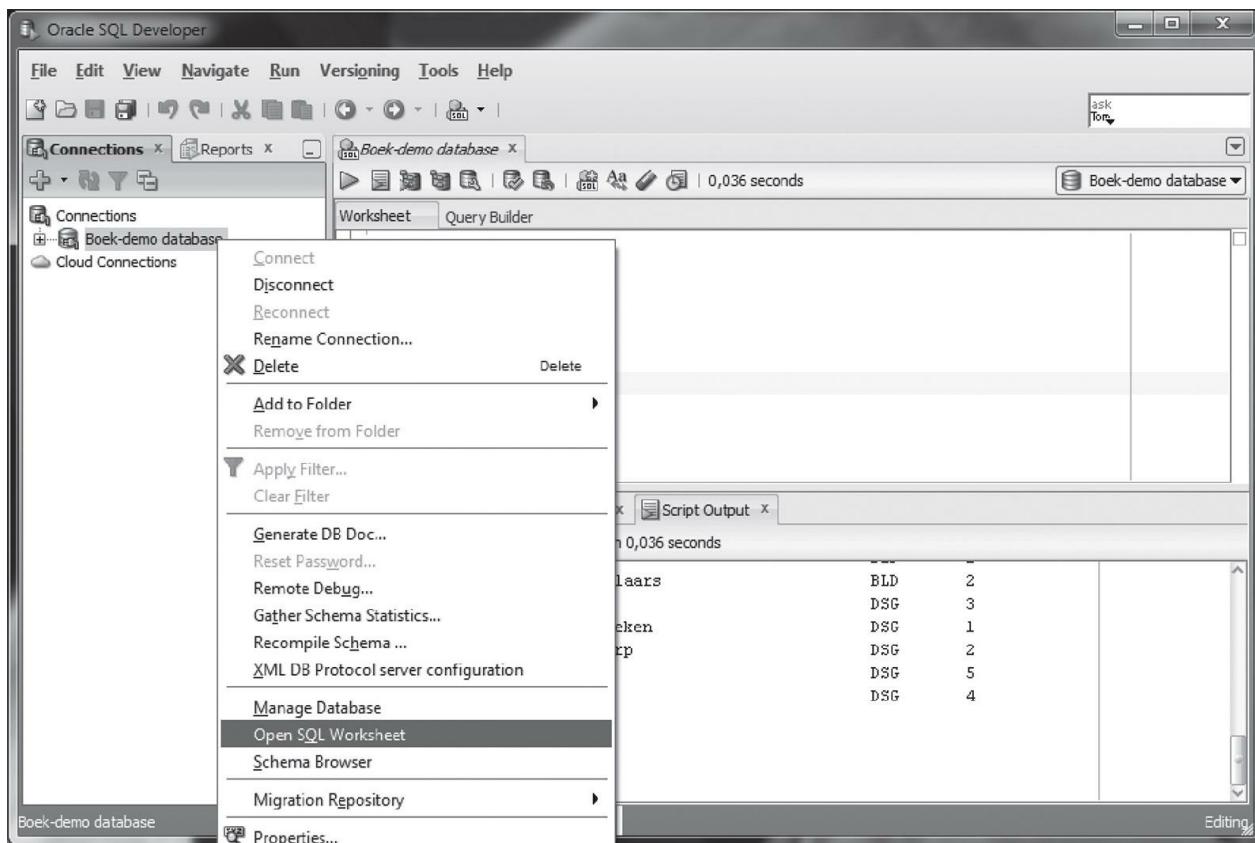
XML	XMI voor Oracle ontwikkelaars	BLD	2
ERM	Datamodelleering met ERM	DSG	3
PNT	Procesmodelleringstechnieken	DSG	1
RSD	Relationeel systeemontwerp	DSG	2
PRO	Prototyping	DSG	5
GEN	Systeemgeneratie	DSG	4

Below the table, it says '10 rows selected'.

**Figuur 2.48**

Door in dit tab naar boven te scrollen zien we de inhoud van achtereenvolgens de afdelingen-, medewerkers- en cursusentabel.

We kunnen ook meerdere Worksheets tegelijk openen. Ga hiervoor naar de Connections-tab, vervolgens naar onze connectie en klik op de rechtermuisknop. Selecteer nu 'Open SQL Worksheet'. Zie [figuur 2.49](#).



**Figuur 2.49**

Er zal nu een tweede Worksheet geopend worden, waarin we ook SQL-commando's kunnen gaan uitvoeren.

Worksheets, Query Result-tabs en Script Output-tabs zijn eenvoudig te sluiten door boven in het tabblad op het kruisje te klikken.

Voor nu weten we genoeg van SQL\*Plus en SQL Developer om in het volgende hoofdstuk met SQL zelf aan de slag te gaan. [In hoofdstuk 11](#)

besteden we opnieuw aandacht aan SQL\*Plus en SQL Developer.

## Hoofdstuk 3

### Datadefinitie – deel I

In dit (korte) hoofdstuk wordt een begin gemaakt met de bespreking van het datadefinitiegedeelte van SQL. Omdat we zo snel mogelijk aan het raadplegen van SQL willen beginnen, komt in dit hoofdstuk alleen het definiëren van eenvoudige tabellen –

en wat daar rechtstreeks mee samenhangt – aan de orde. Dit hoofdstuk is voornamelijk theoretisch van aard; er komen geen opgaven in voor, en er worden slechts een paar voorbeelden gegeven.

In de eerste paragraaf worden databaseschema's en -gebruikers geïntroduceerd; in een Oracle-database maken tabellen altijd onderdeel uit van een schema, en een schema heeft over het algemeen een gebruiker als eigenaar.

De meestgebruikte Oracle-datatypes worden [in paragraaf 3.3](#) behandeld. Als illustratie worden vervolgens in de vierde paragraaf de create table-commando's van de casustabellen gegeven, zonder ons druk te maken om constraints; daar komen we [in hoofdstuk 7 op terug](#).

Overigens is het maken van tabellen in een Oracle-database niet vanzelfsprekend toegestaan; daarvoor zijn enkele privileges nodig, die door de databasebeheerder kunnen worden verleend.

In de vijfde en laatste paragraaf van dit hoofdstuk komt de Oracle-dictionary aan bod. We zullen de globale indeling bespreken, enkele voorbeelden van dictionary-tabellen zien, en wat rondscharrelen in een paar dictionary-tabellen.

[In hoofdstuk 7 komen](#) we zoals gezegd nog terug op het datadefinitiegedeelte van SQL; in dat hoofdstuk worden bijvoorbeeld indexen, synoniemen en constraints behandeld.

## Schema's en gebruikers

Voordat we aan het maken en vullen van tabellen kunnen beginnen, moeten we het eerst even hebben over de manier waarop gegevens in een Oracle-database worden opgeslagen. Zoals we al in het vorige 84

- 
- 

hoofdstuk hebben gezien, kunnen we pas met de database aan de slag nadat we een *gebruikersnaam* en *wachtwoord* hebben verstrekt. We moeten ons dus identificeren als een bepaalde databasegebruiker.

In een Oracle-database bestaat er (over het algemeen) een één-op-één-relatie tussen gebruikers en gelijknamige schema's. We kunnen bijvoorbeeld spreken over de gebruiker BOEK en over het schema BOEK.

Om in een paar woorden het verschil tussen deze twee termen aan te geven:

Een gebruiker heeft een wachtwoord, en bepaalde privileges.

Een schema is een logische verzameling databaseobjecten, waarvan een gebruiker de eigenaar is.

In de praktijk worden beide termen vaak als synoniemen van elkaar gebruikt. Maar feitelijk heeft een databasegebruiker pas een (gelijknamig) schema, zodra die gebruiker eigen databaseobjecten heeft aangemaakt.

In SQL\*Plus kunnen we bijvoorbeeld met behulp van het CONNECT-commando een verbinding maken naar een ander schema, mits we een geldige combinatie van een naam en wachtwoord kunnen verstrekken van de bijbehorende databasegebruiker. Met ALTER SESSION SET

CURRENT\_SCHEMA kunnen we een ander schema “bezoeken” (zonder daarbij de identiteit en privileges van een andere gebruiker aan te nemen). Meer hierover volgt later.

In dit boek gaan we uit van de aanwezigheid van een gebruiker BOEK, met wachtwoord BOEK, en een gelijknamig schema BOEK met (op zijn minst) de

zeven casustabellen die in het eerste hoofdstuk zijn geïntroduceerd. Op de website staan alle scripts die we nodig hebben om dit schema te maken.

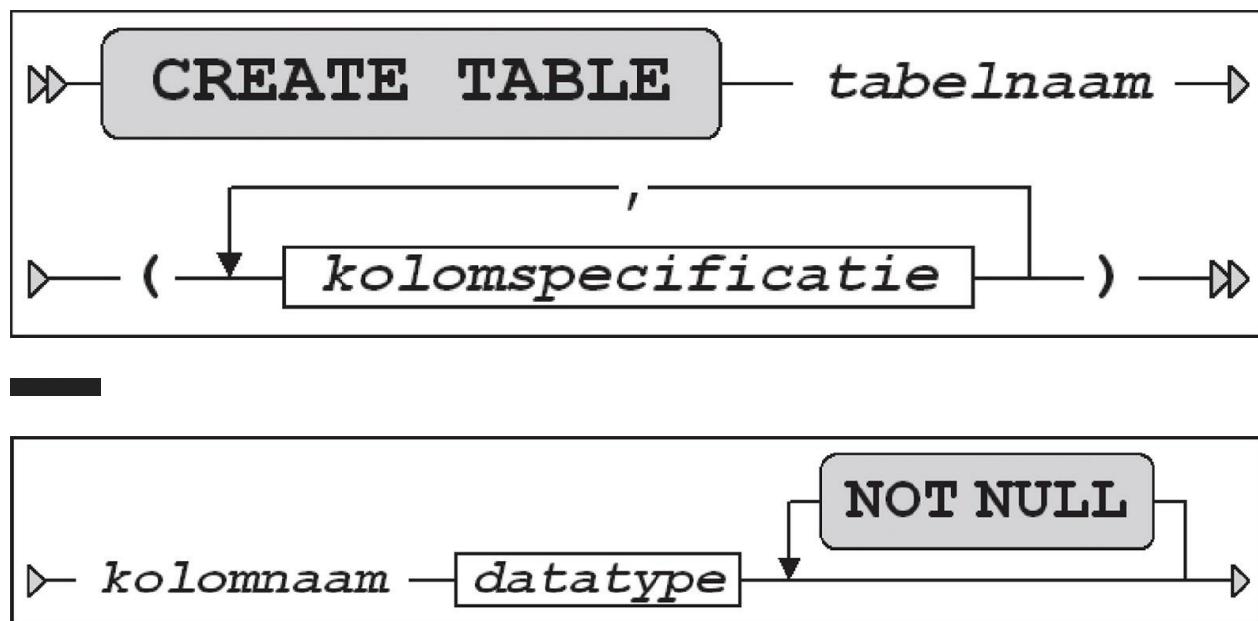
## 3.2

### Tabellen maken

Het SQL-commando om tabellen te maken is CREATE TABLE. Om een tabel te kunnen maken moet voor die tabel een naam worden gekozen en vervolgens moeten alle kolommen worden gespecificeerd. De

kolomspecificatie wordt tussen haakjes gezet, onderling gescheiden door een komma. Het syntaxdiagram ziet eruit als in [figuur 3.1](#).

85



Figuur 3.1

**Let op:** Dit is niet de volledige syntax van het commando CREATE TABLE; daar komen we in [hoofdstuk 7 nog op terug](#). Kijk voor de aardigheid maar eens hoeveel pagina's het CREATE TABLE-commando beslaat in de Oracle-documentatie (google 'oracle create table').

Een kolomspecificatie bestaat uit een aantal onderdelen ([zie figuur 3.2](#)).

### Figuur 3.2

Iedere kolomspecificatie begint met een kolomnaam, gevolgd door het datatype – waarover in de volgende paragraaf meer. Daaraan kan NOT

NULL worden toegevoegd. Hiermee wordt bereikt dat iedere toekomstige rij van de tabel in de bewuste kolom een waarde zal moeten hebben; het is een verplichte kolom.

De toevoeging NOT NULL is een voorbeeld van een constraint. In het commando CREATE TABLE kunnen veel meer constraints worden

gespecificeerd, zoals UNIQUE, CHECK, PRIMARY KEY en FOREIGN KEY. In

[hoofdstuk 7](#) komen deze mogelijkheden aan de orde.

## 3.3

### Datatypes

Oracle ondersteunt vele datatypes, zoals uit de documentatie kan worden opgemaakt. Sommige datatypes lijken op elkaar, of zijn zelfs synoniem; ze worden ondersteund vanwege de compatibiliteit van 86

Oracle met andere databasemanagement-systemen. Andere datatypes zijn zó specifiek van aard dat ze voor ons op dit moment nog niet van belang zijn. We zullen ons hier beperken tot de meestgebruikte datatypes.

Globaal gezien zijn er drie soorten kolomgegevens: getallen (numeriek), tekst (alfanumeriek) en datums. De voornaamste bijbehorende Oracle-datatypes zijn respectievelijk NUMBER, VARCHAR[2] en DATE.

Enkele voorbeelden van het datatype NUMBER:

**NUMBER(4)**

Een geheel getal van maximaal vier cijfers

### **NUMBER(6,2)**

Een getal van maximaal zes cijfers, waarvan  
maximaal twee achter de decimale punt

### **NUMBER(7,-3)**

Een duizendvoud van maximaal zeven cijfers

### **NUMBER**

Hetzelfde als NUMBER(38,\*)

### **NUMBER(\*,5)**

Hetzelfde als NUMBER(38,5)

Er zijn diverse alfanumerieke datatypes. Afhankelijk van de versie van Oracle zijn er enkele verschillen, die het gevolg zijn van de ontwikkeling van de SQL-standaard. Sinds Oracle7 zijn VARCHAR en VARCHAR2 bijvoorbeeld al identiek, maar dat zou in een toekomstige versie [kunnen veranderen. In hoofdstuk 7 komen](#) we daar nog uitgebreider op terug. We volstaan hier weer met enkele voorbeelden: **VARCHAR2(25)**

Alfanumeriek, *variabele* lengte, maximaal 25

karakters

### **CHAR(4)**

Alfanumeriek, *vaste* lengte van vier karakters

Datatype:

Maximale breedte:

## NUMBER

38

CHAR

2000

## **VARCHAR[2]**

4000

CLOB

8 TB

Als de maximale kolombreedte van VARCHAR2 voor een alfanumerieke kolum niet voldoende is, kunnen we het CLOB (Character Large OBject)

-datatype gebruiken. Overigens is de aangegeven maximale breedte (8

TB) niet helemaal correct; afhankelijk van een aantal

87

configuratieparameters kunnen we in kolommen van het CLOB-datatype nog veel meer dan 8 TB aan gegevens opslaan. Zie de Oracle-documentatie voor details (google “oracle clob max size”).

Intern worden datums door Oracle op een zódanige manier opgeslagen dat waarden zijn toegestaan van het jaar 4712 voor Christus tot en met het jaar 9999 na Christus. Datums worden standaard weergegeven in een formaat waarin alleen de dag, de maand en de laatste twee cijfers van het jaartal voorkomen; met behulp van functies kunnen datums echter op vele manieren worden gerepresenteerd. In ieder geval bevat een datum intern ook een tijdsindicatie (uren, minuten en seconden); daarop moeten we bedacht zijn bij het met elkaar vergelijken van ogenschijnlijk gelijke datums.

Oracle ondersteunt naast DATE ook nog de datatypes TIMESTAMP (met of zonder TIME ZONE) en INTERVAL om tijdgerelateerde informatie in een kolom op te slaan; zie [hoofdstuk 7 voor meer details](#).

In dit leerboek zullen we ons zo veel mogelijk beperken tot het gebruik van de standaard-datatypes NUMBER, VARCHAR2 en DATE.

### 3.4

#### De casustabellen

In deze paragraaf worden ter illustratie de commando's gegeven waarmee de casustabellen kunnen worden gecreëerd. Aangezien deze structuur uit zeven tabellen bestaat, gaat het om even zoveel CREATE

TABLE-commando's ([zie de figuren 3.3 tot en met 3.9](#)).

**Let op:** Bewaking van constraints laten we hier nog even buiten beschouwing; dit zijn dus *niet* de volledige commando's om de casustabellen te creëren. [Zie hoofdstuk 7 voor de volledige commando's.](#)

88

```
SQL> create table MEDEWERKERS
  2  ( mnr          number(4)    not null
  3 , naam         varchar2(12) not null
  4 , voirl        varchar2(5)  not null
  5 , functie      varchar2(10)
  6 , chef         number(4)
  7 , gbdatum      date        not null
  8 , maandsal    number(6,2)  not null
  9 , comm         number(6,2)
 10 , afd          number(2)   not null );
```

```
SQL> create table AFDELINGEN
  2  ( anr          number(2)    not null
  3 , naam        varchar2(20) not null
  4 , locatie     varchar2(20) not null
  5 , hoofd       number(4)           );
```

```
SQL> create table SCHALEN
  2  ( snr          number(2)    not null
  3 , ondergrens   number(6,2)  not null
  4 , bovengrens   number(6,2)  not null
  5 , toelage      number(6,2)           );
```

```
SQL> create table CURSUSSEN
  2  ( code         varchar2(4)  not null
  3 , omschrijving varchar2(50) not null
  4 , type         char(3)     not null
  5 , lengte       number(2)   not null );
```

Figuur 3.3

Figuur 3.4

Figuur 3.5

Figuur 3.6

89

```
SQL> create table UITVOERINGEN
  2  ( cursus      varchar2(4)  not null
  3 , begindatum   date       not null
  4 , docent       number(4)
  5 , locatie      varchar2(20)           );
```

```
SQL> create table INSCHRIJVINGEN
  2  ( cursist      number(4)    not null
  3 , cursus       varchar2(4)   not null
  4 , begindatum   date        not null
  5 , evaluatie    number(1)           );
```

```
SQL> create table HISTORIE
  2  ( mnr          number(4)    not null
  3 , beginjaar    number(4)    not null
  4 , begindatum   date        not null
  5 , einddatum    date        not null
  6 , afd          number(2)    not null
  7 , opmerkingen  varchar2(60)
  8 , maandsal    number(6,2)  not null );
```

Figuur 3.7

Figuur 3.8

Figuur 3.9

3.5

## De datadictionary

Als we bijvoorbeeld willen weten welke tabellen in de database aanwezig zijn, welke kolommen ze hebben, en of er al dan niet indexen op zijn gedefinieerd, welke privileges we hebben, en meer van dat soort vragen, dan kunnen we daartoe de datadictionary raadplegen. Dat hebben we overigens impliciet al meer dan eens gedaan; het SQL\*Plus-commando DESCRIBE raadpleegt de datadictionary.

De datadictionary is eigenlijk de interne administratie van Oracle; in de datadictionary worden gegevens over de gegevens ('meta-gegevens') opgeslagen. Deze gegevens worden door Oracle voortdurend dynamisch bijgehouden.

In een relationeel systeem als Oracle worden de datadictionary-90

gegevens op precies dezelfde manier opgeslagen als de ‘gewone’ gegevens: in tabellen dus. Dit is conform de vierde regel van Codd (zie [paragraaf 1.7](#)).

Het grote voordeel hiervan is dat ook datadictionary-gegevens met SQL kunnen worden geraadpleegd, net als ‘gewone’ gegevens. Als we de taal SQL beheersen, hoeven we dus alleen nog maar te weten hoe de datadictionary-tabellen heten, en welke kolommen ze hebben.

Toegang tot de datadictionary is natuurlijk ook een potentieel veiligheidsprobleem; vandaar dat er privileges en rollen bestaan om de toegang tot de datadictionary te reguleren. Er is bijvoorbeeld een rol SELECT\_CATALOG\_ROLE die alle benodigde rechten bevat om de datadictionary-gegevens te raadplegen. Let op wat er in [figuur 3.10](#) gebeurt:

Overigens, we hebben het steeds over datadictionary- *tabellen*, maar eigenlijk zijn het allemaal *views* die we benaderen; zie [hoofdstuk 10](#) voor details over views.

We kunnen de documentatie raadplegen voor een overzicht van de Oracle-datatadictionary, maar gelukkig bevat de Oracle-datatadictionary zelf een tabel waarin staat welke datadictionary-tabellen er zijn en welke informatie erin aan te treffen valt (zie [figuur 3.11](#)). Deze tabel heet DICTIONARY, waarvoor DICT een synoniem is.

```

SQL> describe dba_sys_privs
ERROR:
ORA-04043: object "SYS"."DBA_SYS_PRIVS" does not exist

SQL> connect / as sysdba
Connected.

SQL> grant select_catalog_role to boek;
Grant succeeded.

SQL> connect boek/boek
Connected.

SQL> desc dba_sys_privs
      Name          Null?    Type
----- -----
GRANTEE           NOT NULL VARCHAR2(30)
PRIVILEGE         NOT NULL VARCHAR2(40)
ADMIN_OPTION      VARCHAR2(3)

SQL>

```

```

SQL> col COLUMN_NAME format a30
SQL> col COMMENTS      format a40 word
SQL>
SQL> select * from dict;

TABLE_NAME          COMMENTS
-----
USER_RESOURCE_LIMITS Display resource limit of the user
USER_PASSWORD_LIMITS Display password limits of the user
USER_CATALOG          Tables, Views, Synonyms and Sequences
                      owned by the user
ALL_CATALOG          All tables, views, synonyms, sequences
                      accessible to the user
USER_CLUSTERS        Descriptions of user's own clusters
ALL_CLUSTERS         Description of clusters accessible to
                      the user
...

```

## Figuur 3.10

Uiteraard geven we in figuur 3.11 niet de hele uitvoer; die omvat meer dan

600 rijen...

### Figuur 3.11

Wat bij de datadictionary-tabelnamen opvalt, is een indeling in groepen.

92

Het idee daarachter is dat we soms uitsluitend geïnteresseerd zijn in onze eigen gegevens, terwijl we op een ander moment misschien willen weten wiens tabellen we allemaal mogen benaderen.

Databasebeheerders willen tenslotte vaak een totaaloverzicht van de database.

USER\_...

Informatie over eigen objecten

ALL\_...

Informatie over benaderbare objecten

DBA\_...

Alleen toegankelijk voor databasebeheerders

V\$...

De ‘dynamic performance views’

De ‘dynamic performance views’ vormen een bijzondere categorie; het zijn eigenlijk helemaal geen tabellen (of op tabellen gebaseerde views) maar het zijn objecten die als een tabel kunnen worden geraadpleegd, waarbij de informatie rechtstreeks aan het interne werkgeheugen wordt ontleend. Ze zijn vooral van belang voor databasebeheerders.

De namen geven over het algemeen een duidelijke indicatie van de inhoud van de tabellen; daar staat tegenover dat de namen nogal lang kunnen worden. Daarom zijn voor enkele veelgebruikte tabellen synoniemen gedefinieerd,

zoals CAT, OBJ, IND, TABS en COLS. Vooral CAT

is erg nuttig; zie [figuur 3.13 voor een voorbeeld](#).

Willen we (om een query te kunnen formuleren) van een specifieke datadictionary-tabel weten welke kolommen hij heeft, dan kunnen we net als bij gewone tabellen het SQL\*Plus-commando DESCRIBE

gebruiken (zie bijvoorbeeld [figuur 3.10](#)); we kunnen ook de datadictionary-tabel DICT\_COLUMNS raadplegen (zie [figuur 3.12](#)).

93

```
SQL> describe DICT_COLUMNS
Name           Null? Type
-----
TABLE_NAME          VARCHAR2(30)
COLUMN_NAME         VARCHAR2(30)
COMMENTS           VARCHAR2(4000)

SQL> select column_name, comments
  2  from dict_columns
  3  where table_name = 'ALL_USERS';

COLUMN_NAME           COMMENTS
-----
USERNAME             Name of the user
USER_ID              ID number of the user
CREATED              User creation date

SQL>
```

```

SQL> select * from cat;

TABLE_NAME          TABLE_TYPE
-----
MEDEWERKERS          TABLE
AFDELINGEN          TABLE
SCHALEN              TABLE
CURSUSSEN            TABLE
UITVOERINGEN         TABLE
INSCHRIJVINGEN      TABLE
HISTORIE             TABLE

7 rows selected.

SQL>

```

**Figuur 3.12**

**Figuur 3.13**

In hoofdstuk 2 hebben we het al gehad over diverse instellingen met betrekking tot het weergeven van datums en getallen, zoals bijvoorbeeld NLS\_DATE\_FORMAT. Zie figuur 3.14 voor de datadictionary view die al onze NLS sessie-instellingen weergeeft:

94

```

SQL> select * from nls_session_parameters;

PARAMETER          VALUE
-----
NLS_LANGUAGE        AMERICAN
NLS_TERRITORY       AMERICA
NLS_CURRENCY         $
NLS_ISO_CURRENCY    AMERICA
NLS_NUMERIC_CHARACTERS   ,
NLS_CALENDAR         GREGORIAN
NLS_DATE_FORMAT      DD-MON-YYYY
NLS_DATE_LANGUAGE    AMERICAN
...

```

**Figuur 3.14**

NLS staat overigens voor ‘National Language Support’. De NLS-voorzieningen in Oracle zijn uitgebreid gedocumenteerd in de *Globalization Support Guide*.

Tot slot van dit hoofdstuk volgt hier een selectie uit de tabellen van de datadictionary.

## DICTIONARY

Beschrijving van de datadictionary

### DICT\_COLUMNS

Datadictionary-kolombeschrijvingen

### ALL\_USERS

Informatie over alle databasegebruikers

### ALL\_INDEXES

Alle indexen

### ALL\_SEQUENCES

Alle sequences

*Toegankelijk voor de gebruiker*

### ALL\_OBJECTS

Alle objecten

### ALL\_SYNONYMS

Alle synoniemen

### ALL\_TABLES

Alle tabellen

ALL\_VIEWS

Alle views

USER\_INDEXES

Indexen

USER\_SEQUENCES

Sequences

USER\_OBJECTS

Objecten

*Eigendom van de gebruiker*

*zelf*

USER\_SYNONYMS

Synoniemen

USER\_TABLES

Tabellen

USER\_TAB\_COLUMNS

Kolommen

95

USER\_VIEWS

Views

**USER\_RECYCLEBIN**

Verwijderde

objecten

CAT

Synoniem voor **USER\_CATALOG**

COLS

Synoniem voor **USER\_TAB\_COLUMNS**

DICT

Synoniem voor **DICTIONARY**

DUAL

Dummy-tabel, met één rij en één kolom

IND

Synoniem voor **USER\_INDEXES**

OBJ

Synoniem voor **USER\_OBJECTS**

SYN

Synoniem voor **USER\_SYNONYMS**

TABS

Synoniem voor **USER\_TABLES**

## **Hoofdstuk 4**

### **Raadpleging – de basis**

In dit hoofdstuk gaan we met SQL de voorbeelddatabase raadplegen; dat doen we met het SELECT-commando.

Raadplegingen worden ook wel queries genoemd.

Van de zes hoofdcomponenten van het SELECT-commando komen er in dit hoofdstuk vier aan de orde; de andere twee stellen we uit tot [hoofdstuk 8](#).

Met de SELECT-component geven we aan welke kolommen we in de resultaattabel wensen te zien, en welke koppen erboven komen te staan.

Wat de FROM-component betreft beperken we ons in dit eerste hoofdstuk over raadpleging tot het benaderen van één tabel tegelijk.

Met de WHERE-component kunnen we condities formuleren waaraan de rijen die we zoeken moeten voldoen. Daarbij staan ons diverse operatoren ter beschikking, zoals BETWEEN, LIKE, NOT, AND en OR.

Met de ORDER BY-component wordt de rijvolgorde in de resultaattabel geregeld. *Null-waarden* en de bijbehorende driewaardige logica (voorwaarden kunnen in SQL als resultaat

‘waar’, ‘onwaar’ of ‘onbekend’ opleveren) worden in een aparte paragraaf behandeld.

Queries kunnen als component van een ander SQL-commando voorkomen. In dit hoofdstuk maken we een begin met de

bespreking van dergelijke *subqueries*; in hoofdstuk 9 komen we er nog op terug.

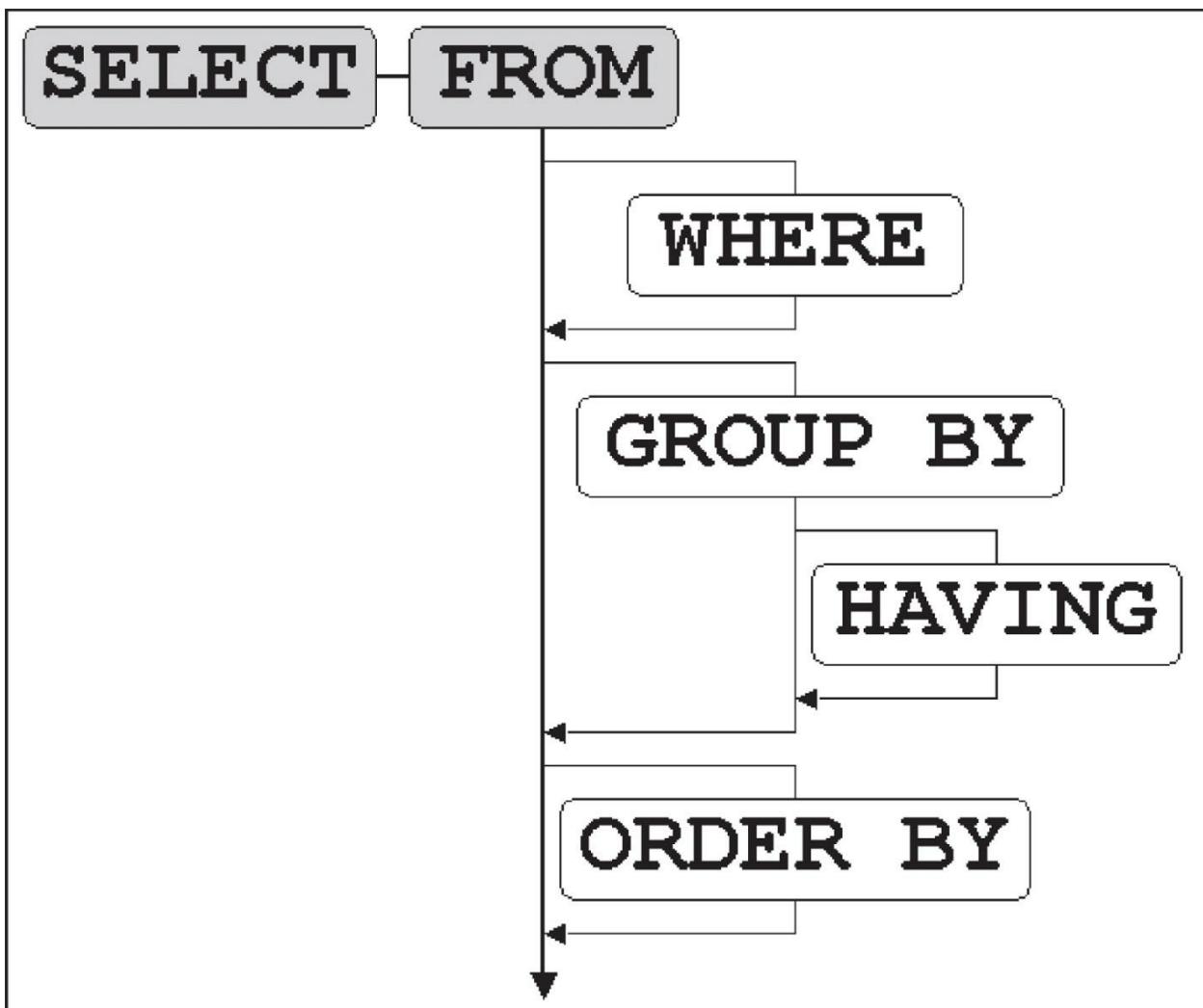
Het hoofdstuk wordt afgesloten met opgaven.

#### 4.1

Overzicht van de SELECT-componenten

Op deze pagina staat een korte samenvatting van hetgeen al eerder is behandeld:

97



**Figuur 4.1**

[Figuur 4.1](#) is een herhaling van figuur 2.1, en illustreert de volgende vier hoofdregels van het SELECT-statement:

De volgorde van de componenten ligt vast.

SELECT en FROM zijn verplicht.

**WHERE**, GROUP BY en ORDER BY zijn optioneel.

HAVING komt niet voor zonder GROUP BY.

## **SELECT**

Welke kolom(men) krijgt de resultaattabel.

## **FROM**

Welke tabel(len) worden geraadpleegd.

## **WHERE**

Waaraan moeten de rijen voldoen.

## **GROUP BY**

Waarop moet worden gegroepeerd.

## **HAVING**

Waaraan moeten de groepen rijen voldoen.

## **ORDER BY**

In welke volgorde wensen we het resultaat.

Deze zes componenten dienen (volgens de ANSI/ISO-standaard) in de volgende volgorde te worden verwerkt: FROM, WHERE, GROUP BY, HAVING, 98

```

SQL> select * from afdelingen;

ANR NAAM          LOCATIE      HOOFD
----- -----      -----
 10 HOOFDKANTOOR    LEIDEN       7782
 20 OPLEIDINGEN    DE MEERN     7566
 30 VERKOOP         UTRECHT      7698
 40 PERSONEELSZAKEN GRONINGEN   7839

SQL>

```

**SELECT, ORDER BY.** Dus eerst worden de rijen van de genoemde tabel(len) in de FROM-clausule opgevraagd, op die rijen worden dan de condities van de WHERE-clausule toegepast, dan worden optioneel de rijen gegroepeerd conform de GROUP BY-clausule, eventueel worden op deze gegroepeerde rijen de condities van de HAVING-clausule toegepast, dan worden de expressies van de SELECT-clausule toegepast om de rijen van de resultaattabel te bepalen, die ten slotte gesorteerd worden conform de ORDER BY-clausule.

## 4.2

### De SELECT-component

Laten we beginnen met een eenvoudig commando, zoals [in figuur 4.2.](#)

#### Figuur 4.2

De asterisk (\*) betekent: laat alle kolommen van de tabel afdelingen zien. Als we gegevens raadplegen, dan spreken we meestal over queries.

Anders geformuleerd: een SELECT-commando wordt ook wel een *query* genoemd.

```
SQL> select naam, voorl, functie, maandsal  
2   from medewerkers  
3  where afd = 30;
```

NAAM	VOORL	FUNCTIE	MAANDSAL
ALDERS	JAM	VERKOPER	1600
DE WAARD	TF	VERKOPER	1250
MARTENS	P	VERKOPER	1250
BLAAK	R	MANAGER	2850
DEN DRAAIER	JJ	VERKOPER	1500
JANSEN	R	BOEKHOUDER	800

```
SQL>
```

### Figuur 4.3

Het resultaat van deze query spreekt voor zich. We zullen van deze query de *syntax* (de taalconstructieregels) eens nauwgezet bekijken. In principe zijn we daar erg vrij in; we kunnen een commando volledig op één regel achter elkaar ingeven, een commando spreiden over diverse regels, en extra spaties of tabs gebruiken.

Het is verstandig om zoveel mogelijk een standaardlay-out aan te houden; dat verhoogt de leesbaarheid en de onderhoudbaarheid. Met de term *witruimte* bedoelen we vanaf nu: een of meer spaties en/of tabs en/of een regelovergang.

In de SELECT-component is witruimte verplicht na het woord SELECT; de kolommen zijn onderling gescheiden door een komma, waarbij witruimte niet nodig is maar wel is toegestaan. Verder is witruimte verplicht na FROM en WHERE. Rond het isgelijkteken is de witruimte niet noodzakelijk.

We kunnen een kolomkop van de resultaattabel veranderen op de manier die is weergegeven in [figuur 4.4](#).

```
SQL> select naam, voorl, maandsal AS salaris  
2   from medewerkers  
3  where afd = 30;
```

NAAM	VOORL	SALARIS
ALDERS	JAM	1600
DE WAARD	TF	1250
MARTENS	P	1250
BLAAK	R	2850
DEN DRAAIER	JJ	1500
JANSEN	R	800

```
SQL>
```

#### Figuur 4.4

Het AS-woord geeft ons de mogelijkheid een eigen naam te introduceren voor de betreffende tabel/kolomnaam. Het AS-woord is overigens optioneel: de tabelkolom naam en eigen naam mogen ook door witruimte gescheiden zijn. Dit commando had er dus ook als volgt mogen uitzien:

```
SQL>
```

```
select naam, voorl, maandsal salaris
```

```
2
```

```
from medewerkers
```

```
3
```

```
where afd = 30;
```

Oracle noemt dit soort eigen namen (kolomtitels) *kolom-aliassen*.

Opmerking: we hadden hetzelfde resultaat op het scherm kunnen bereiken door niet de query zelf aan te passen, maar in plaats daarvan de optie HEADING van het SQL\*Plus-commando COLUMN te gebruiken.

Daarover meer in [hoofdstuk 11](#).

Het kan gebeuren dat een resultaattabel dubbele rijen bevat. Dit soort dubbele rijen kunnen we desgewenst elimineren met behulp van de toevoeging DISTINCT (zie [figuur 4.5](#)).

101

```
SQL> select DISTINCT functie, afd  
  2  from medewerkers;
```

FUNCTIE	AFD
BOEKHOUDER	10
BOEKHOUDER	30
DIRECTEUR	10
MANAGER	10
MANAGER	20
MANAGER	30
TRAINER	20
VERKOPER	30

```
SQL>
```

```
SQL> select snr, bovengrens - ondergrens  
  2  from schalen;
```

SNR	BOVENGRENS-ONDERGRENS
1	500
2	199
3	599
4	999
5	6998

```
SQL>
```

## Figuur 4.5

Deze query zou zonder de toevoeging DISTINCT veertien rijen opleveren, omdat we veertien medewerkers hebben.

In plaats van kolomnamen mogen ook berekeningen of bewerkingen op kolomnamen worden geselecteerd. We gaan een aantal voorbeelden bekijken waarin we dit soort expressies van kolomnamen zullen selecteren.

In figuur 4.6 berekenen we de schaalbreedte van de salarisschalen, door het verschil van bovengrens en ondergrens te selecteren.

In figuur 4.7 voegen we naam en voorletters samen in één kolom, en berekenen het jaarsalaris door het maandsalaris met 12 te vermenigvuldigen.

### Figuur 4.6

102

```
SQL> select voorl||' '||naam as naam
  2 ,      12 * maandsal    as jaarsal
  3 from   medewerkers
  4 where  afd = 10;
```

NAAM	JAARSAL
AB CLERCKX	29400
CC DE KONING	60000
TJA MOLENAAR	15600

```
SQL>
```

```
SQL> select 3 + 4 from afdelingen;
```

3+4
7
7
7
7

```
SQL>
```

### Figuur 4.7

## Figuur 4.8

NB: alleen in [figuur 4.7 hebben we zelf](#) nieuwe kolom-aliasen geïntroduceerd voor de expressies die geselecteerd worden. In gevallen waarin we dat niet doen (4.6 en 4.8) zien we dat Oracle zelf een kolomnaam heeft bepaald die afhangt van de expressie die we gebruikt hebben.

Het resultaat in [figuur 4.8 komt](#) misschien wat vreemd over, maar is bij nadere beschouwing toch consequent: de expressie  $3+4$  wordt voor iedere rij van de tabel uitgewerkt, en omdat een WHERE-component ontbreekt wordt de expressie voor alle vier rijen uitgewerkt. Omdat er bovendien geen variabelen in de expressie  $3+4$  voorkomen, is het resultaat van de expressie (7) voor elke rij natuurlijk hetzelfde.

Dit soort queries kun je beter op een dummytabel loslaten: een tabel met één rij en één kolom. Zo'n tabel zou je zelf kunnen maken, maar er is al een centrale dummytabel in de datadictionary van Oracle aanwezig, met de naam DUAL.

103

```
SQL> select 123 * 456 from dual;

123*456
-----
56088

SQL> select sysdate from dual;

SYSDATE
-----
31-OCT-2012

SQL>
```

```
SQL> select naam, (sysdate-gbdatum)/365  
2  from medewerkers  
3  where mnr = 7839;
```

NAAM	(SYSDATE-GBDATUM) / 365
DE KONING	59.9949034

```
SQL>
```

In figuur 4.9 zien we twee toepassingsvoorbeelden van de tabel DUAL.

Merk op dat de inhoud van deze tabel volstrekt onbelangrijk is; alleen het feit dat de tabel één rij heeft is van belang.

### Figuur 4.9

### Figuur 4.10

In de laatste twee voorbeelden wordt gebruikgemaakt van de systeemdatum, die kan worden aangeroepen met SYSDATE. Het wordt ook wel een pseudokolom genoemd. Natuurlijk is de uitvoer van queries waarin SYSDATE wordt gebruikt afhankelijk van het moment waarop ze worden uitgevoerd, dus het resultaat ziet er bij u ongetwijfeld anders uit dan in figuur 4.9 en 4.10.

We moeten bedacht zijn op eventuele null-waarden in expressies. Als één van de variabelen in een expressie een null-waarde aanneemt, is het resultaat van de expressie als geheel onbepaald. Op deze problematiek komen we nog uitgebreid terug; als voorproefje daarvan kijken we naar het resultaat van de query in figuur 4.11. Daar zien we dat het totale jaarsalaris (inclusief commissie) van twee van de vier medewerkers 104

```
SQL> select naam, 12 * maandsal + comm  
2   from medewerkers  
3  where mnr < 7600;
```

NAAM	12 *MAANDSAL+COMM
SMIT	
ALDERS	19500
DE WAARD	15500
JANSEN	

```
SQL>
```

onbekend is, omdat de commissie van die medewerkers een null-waarde bevat.

### Figuur 4.11

## 4.3

### De WHERE-component

Met de WHERE-component kunnen we een *voorwaarde* of *conditie* specificeren waaraan de rijen moeten voldoen om in het resultaat te komen. We maken onderscheid tussen *enkelvoudige* en *samengestelde* condities.

Enkelvoudige condities zijn expressies waarin een vergelijkingsoperator van SQL voorkomt:

< kleiner dan

$\geq$

kleiner dan of gelijk aan

> groter dan

$\geq$

groter dan of gelijk aan

= gelijk aan

<>

(of !=) ongelijk aan

Expressies waarin dit soort operatoren voorkomen, vormen een uitspraak die waar (TRUE) of onwaar (FALSE) kan zijn. Tenminste, zo is het in de wiskundige logica geregeld. We zullen in een van de volgende paragrafen zien dat null-waarden dit in SQL iets gecompliceerder maken, maar op dit moment maken we ons daar nog niet druk om.

De figuren 4.12 en 4.13 geven twee voorbeelden van een WHERE-component met enkelvoudige condities. In [paragraaf 4.5 van dit hoofdstuk](#) zullen we nader ingaan op samengestelde condities, die we kunnen bouwen met behulp van de logische operatoren AND, OR en NOT.

105

```
SQL> select naam, voorl, maandsal  
  2  from medewerkers  
  3  where maandsal >= 3000;
```

NAAM	VOORL	MAANDSAL
SCHOTTEN	SCJ	3000
DE KONING	CC	5000
SPIJKER	MG	3000

```
SQL>
```

```
SQL> select naam, locatie  
  2  from afdelingen  
  3  where locatie <> 'UTRECHT';
```

NAAM	LOCATIE
HOOFDKANTOOR	LEIDEN
OPLEIDINGEN	DE MEERN
PERSONEELSZAKEN	GRONINGEN

```
SQL>
```

Figuur 4.12

Figuur 4.13

4.4

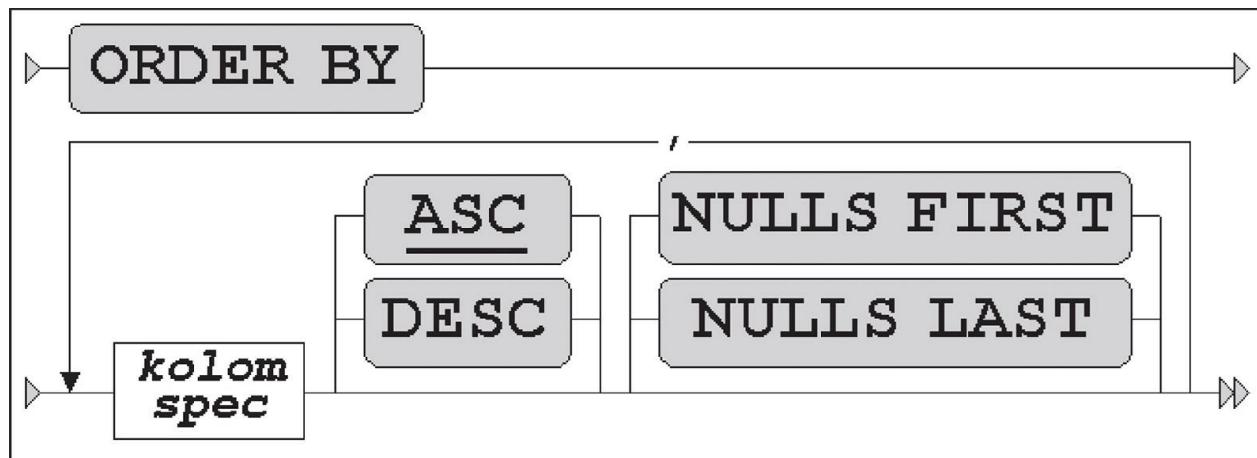
## De ORDER BY-component

Het resultaat van een query is een tabel: een verzameling rijen dus. De volgorde van deze rijen is het gevolg van de strategie die de SQL optimizer heeft gekozen om de query op te lossen. Daardoor is deze volgorde lastig voorspelbaar, en in ieder geval niet onder alle omstandigheden constant.

Als we de rijen *gegarandeerd* in een bepaalde volgorde in het resultaat willen zien, dan moeten we de ORDER BY-component gebruiken (zie

[figuur 4.14](#)).

106



Figuur 4.14

We kunnen meerdere sorteerspecificaties opgeven, onderling gescheiden door komma's. Iedere sorteerspecificatie bestaat uit een kolomspecificatie, eventueel gevuld door de toevoeging DESC

(descending) als we aflopend willen sorteren. Zonder deze toevoeging wordt

ASC (ascending) gesorteerd.

De kolomspecificatie kan bestaan uit een enkele kolom of uit een kolomexpressie. Om kolommen aan te duiden kunnen we reguliere kolomnamen gebruiken, kolom-aliassen die we in de SELECT-component hebben gedefinieerd (vooral handig bij complexe expressies in de SELECT-component), óf uit het kolomnummer binnen de resultaattabel.

Let wel: dit kolomnummer heeft niets te maken met de volgorde van de kolommen in de database, maar wordt bepaald door de SELECT-component van de query.

Gebruik van kolomnummers als kolomspecificatie in de ORDER BY-component is niet aan te raden; sorteren op kolom-aliassen verhoogt de leesbaarheid, en maakt de ORDER BY-component onafhankelijker van de SELECT-component.

107

```
SQL> select afd, naam, voorl, maandsal
  2  from medewerkers
  3  where maandsal < 1500
  4  order by afd, naam;
```

AFD	NAAM	VOORL	MAANDSAL
10	MOLENAAR	TJA	1300
20	ADAMS	AA	1100
20	SMIT	N	800
30	DE WAARD	TF	1250
30	JANSEN	R	800
30	MARTENS	P	1250

```
SQL>
```

```
SQL> select naam, 12*maandsal+comm as jaarsal  
2   from medewerkers  
3  where functie = 'VERKOPER'  
4  order by jaarsal desc;
```

NAAM	JAARSAL
ALDERS	19500
DEN DRAAIER	18000
MARTENS	16400
DE WAARD	15500

```
SQL>
```

## Figuur 4.15

## Figuur 4.16

Wat sorteren betreft zorgen null-waarden voor een probleem. We kunnen ze in theorie op vier manieren behandelen:

1

Altijd als eerste waarden (ongeacht de sorteerrichting).

2

Altijd als laatste waarden (ongeacht de sorteerrichting).

3

Als laagste waarden (low-values).

4

Als hoogste waarden (high-values).

[Figuur 4.14](#) laat zien dat we per kolomexpressie expliciet kunnen angeven hoe we de null-waarden behandeld willen zien. Laten we eens kijken wat het standaardgedrag is, met behulp van het voorbeeld in

## figuur 4.17.

108

```
SQL> select evaluatie  
2   from inschrijvingen  
3  where cursist = 7788  
4  order by evaluatie;
```

EVALUATIE

```
-----  
4  
5
```

```
SQL>
```

```
SQL> select evaluatie  
2   from inschrijvingen  
3  where cursist = 7788  
4  order by evaluatie DESC;
```

EVALUATIE

```
-----  
5  
4
```

```
SQL>
```

- 
- 

## **Figuur 4.17**

De null-waarde is hier wat lastig te zien, maar het is de derde rij.

Sorteren we nu aflopend, dan is het resultaat zoals [figuur 4.18](#).

## **Figuur 4.18**

Hieruit blijkt dat Oracle null-waarden als ‘high-values’ behandelt. Met andere woorden:

NULLS LAST is de standaard voor ASC.

NULLS FIRST is de standaard voor DESC.

## 4.5

### AND, OR, NOT

Met behulp van de logische operatoren AND en OR zijn enkelvoudige condities met elkaar te combineren tot een samengestelde conditie. Als AND wordt gebruikt, geven we aan dat iedere rij aan beide gestelde voorwaarden dient te voldoen; als OR wordt gebruikt, hoeft slechts één van beide voorwaarden vervuld te zijn.

109

```
SQL> select code, type, lengte
  2  from cursussen
  3  where type = 'BLD'
  4 or    lengte = 2;
```

CODE	TYP	LENGTE
JAV	BLD	4
PLS	BLD	1
XML	BLD	2
RSO	DSG	2

```
SQL>
```

Dat klinkt eenvoudig. Nu worden de woorden ‘en’ en ‘of’ in de natuurlijke taal nogal slordig gebruikt. Uit de context is de bedoeling dan meestal wel te achterhalen. In een formele taal als SQL liggen de betekenissen van AND en OR natuurlijk volledig vast, en is er geen ruimte voor verschillende interpretaties. Bij een vertaling van een vraag in het Nederlands naar SQL kunnen (zullen?) we ons nog wel eens vergissen.

Er wordt vaak onderscheid gemaakt tussen *inclusief* en *exclusief* gebruik van of: moet de ene voorwaarde de andere uitsluiten, of niet? In de natuurlijke taal is het onderscheid vrijwel altijd impliciet: als we bijvoorbeeld met iemand een

afspraak proberen te maken voor donderdag of vrijdag, bedoelen we dat natuurlijk exclusief. Hoe zit dat nu in SQL?

Laten we ter illustratie naar het volgende voorbeeld kijken (zie figuur

4.19). Uit dit voorbeeld blijkt dat de OR-operator van SQL inclusief

werkt; anders had de derde rij niet in het resultaat mogen voorkomen.

De XML-cursus is immers zowel een bouwcursus (dus de eerste voorwaarde is vervuld) als een cursus met een lengte van twee dagen (dus aan de tweede voorwaarde is ook voldaan).

### **Figuur 4.19**

Er kan zich een probleem voordoen als AND en OR beide in één samengestelde conditie worden gebruikt. We proberen dat met behulp van een query op de DUAL-tabel uit (zie figuur 4.20).

110

```
SQL> select 'is waar ' as conditie
  2  from  dual
  3 where 1=1 or 1=0
  4   and  0=1;
```

```
CONDITIE
-----
is waar
```

```
SQL>
```

```
SQL> select 'is waar ' as conditie
  2  from  dual
  3  where  (1=1 or 1=0) and 0=1;

no rows selected

SQL> select 'is waar ' as conditie
  2  from  dual
  3  where  1=1 or (1=0 and 0=1);

CONDITIE
-----
is waar

SQL>
```

## Figuur 4.20

Het is duidelijk dat de eerste expressie ( $1=1$ ) waar is, en dat de andere twee onwaar zijn. Maar hoe wordt nu het waarheidsgehalte van de samengestelde conditie als geheel bepaald?

Dat hangt af van de *precedentie* (voorrangsregeling) die wordt toegepast. Lezen we van links naar rechts,  $1=1$  OR ..., dan lijkt de conclusie TRUE logisch; andersom, ... AND  $0=1$ , ligt FALSE meer voor de hand. Is de OR-operator belangrijker dan de AND-operator, of juist andersom?

Het beste advies onder deze omstandigheden is altijd: laat het er niet op aankomen, maar gebruik haakjes om de precedentie af te dwingen. In

[figuur 4.21 proberen we](#) twee verschillende haakjesvarianten uit.

## Figuur 4.21

```
SQL> select naam, functie, afd
  2  from medewerkers
  3  where NOT afd > 10;
```

NAAM	FUNCTIE	AFD
CLERCKX	MANAGER	10
DE KONING	DIRECTEUR	10
MOLENAAR	BOEKHOUDER	10

```
SQL>
```

```
SQL> select naam, functie, afd
  2  from medewerkers
  3  where afd <= 10;
```

NAAM	FUNCTIE	AFD
CLERCKX	MANAGER	10
DE KONING	DIRECTEUR	10
MOLENAAR	BOEKHOUDER	10

```
SQL>
```

Laat u in dit soort gevallen niet van de wijs brengen door de eventuele fraaie lay-out van een query: tabs, spaties en regelovergangen verhogen weliswaar de leesbaarheid, maar voor SQL heeft alle witruimte in een commando dezelfde betekenis.

De operator NOT kan in principe op iedere conditie worden losgelaten, waardoor de conditie wordt ontkend. Zie bijvoorbeeld [figuur 4.22](#).

## Figuur 4.22

Dat kan in dit eenvoudige geval natuurlijk ook anders, door de ‘>’ te veranderen in ‘<=’ ([zie figuur 4.23](#)). De NOT-operator wordt eigenlijk pas interessant bij samengestelde condities, waarin tevens AND en OR

worden gebruikt. We hebben dan bijvoorbeeld meer greep op

correctheid.

### Figuur 4.23

Let erop dat de NOT-operator altijd vóór de conditie wordt gezet. De 112

```
SQL> select naam, functie, afd
  2  from medewerkers
  3  where afd NOT > 10;
where afd NOT > 10
      *
ERROR at line 3:
ORA-00920: invalid relational operator

SQL>
```

```
SQL> select code, type, lengte
  2  from cursussen
  3  where (type = 'BLD' or lengte = 2)
  4  and not (type = 'BLD' and lengte = 2);

CODE TYP LENGTE
---- - - -----
JAV BLD      4
PLS BLD      1
RSO DSG      2

SQL>
```

constructie van [figuur 4.24](#) is daarom syntactisch onjuist.

### Figuur 4.24

Er zijn overigens uitzonderingen op deze regel; in de volgende paragraaf zullen we zien dat een drietal SQL-operatoren een eigen ingebouwde ontkenningsmogelijkheid kent (BETWEEN, IN, LIKE).

Vooral als er sprake is van ontkenningen in een samengestelde conditie is het toepassen van haakjes altijd aan te raden, om daarmee de reikwijdte van de NOT-operator precies aan te geven ([zie figuur 4.25](#)).

In dit voorbeeld bouwen we overigens de ‘exclusieve of’, door op de vierde regel uit te sluiten dat beide voorwaarden tegelijkertijd vervuld zijn; vandaar dat de XML-cursus nu ontbreekt (vergelijk het resultaat met [figuur 4.19](#)).

### Figuur 4.25

We kunnen in SQL ook haakjes wegwerken: de volgende twee queries zijn equivalent. Let op: de ontkenning komt in de twee condities terecht, 113

```
SQL> select naam, voorl, maandsal  
  2  from medewerkers  
  3  where maandsal between 1300 and 1600;
```

NAAM	VOORL	MAANDSAL
ALDERS	JAM	1600
DEN DRAAIER	JJ	1500
MOLENAAR	TJA	1300

```
SQL>
```

en de AND verandert in OR.

```
select * from medewerkers
```

**NOT** (naam = ‘BLAAK’ **AND** voorl =

where

‘R’)

```
select * from medewerkers
```

naam <> ‘BLAAK’ **OR** voorl <>

where

‘R’

We passen hier een wet van De Morgan toe (zie

[http://nl.wikipedia.org/wiki/Wetten\\_van\\_De\\_Morgan\).](http://nl.wikipedia.org/wiki/Wetten_van_De_Morgan).)

4.6

BETWEEN, IN, LIKE

De operator BETWEEN maakt bepaalde formuleringen in SQL wat eenvoudiger en beter leesbaar (zie [figuur 4.26](#)).

#### Figuur 4.26

Uit dit voorbeeld blijkt, dat de BETWEEN-operator de beide grenswaarden meeneemt.

Deze operator kent bovendien een ‘eigen’ ontkenning, die ingebouwd mag worden. De volgende drie formuleringen zijn gelijkwaardig: 1

where maandsal **NOT** between 1000 and 2000

2

where **NOT** maandsal between 1000 and 2000

3

where maandsal < 1000 **OR** maandsal > 2000

De tweede operator die in deze paragraaf aan de orde komt is de IN-operator; hiermee kunnen we zoeken op een gegeven reeks waarden (zie

[figuur 4.27](#)).

114

```
SQL> select mnr, naam, voorl  
2   from medewerkers  
3  where mnr in (7499,7566,7788);
```

MNR	NAAM	VOORL
7499	ALDERS	JAM
7566	JANSEN	JM
7788	SCHOTTEN	SCJ

```
SQL>
```

```
SQL> select * from inschrijvingen  
2  where evaluatie NOT IN (3,4,5);
```

CURSIST	CURS	BEGINDATU	EVALUATIE
7876	S02	12-APR-99	2
7499	JAV	13-DEC-99	2

```
SQL>
```

## Figuur 4.27

Net als BETWEEN beschikt ook de IN-operator over een eigen ontkenningsmogelijkheid.

De volgende query ([zie figuur 4.28](#)) geeft ons de inschrijvingen die niet als evaluatie een 3, 4 of 5 hebben.

## Figuur 4.28

Voor de volgende vier verschillende formuleringen geldt (ga maar na!) dat ze gelijkwaardig zijn:

1

where evaluatie **NOT** in (3,4,5)

2

where **NOT** evaluatie in (3,4,5)

3

where **NOT** (evaluatie=3 **OR** evaluatie=4 **OR** [evaluatie=5](#))

4

where evaluatie<>3 **AND** evaluatie<>4 **AND** [evaluatie<>5](#)

Een voor de hand liggende eis bij toepassing van de IN-operator [or is dat](#) de waarden tussen haakjes van hetzelfde (relevante) datatype [moeten](#) zijn.

De derde operator in deze paragraaf is LIKE. Deze operat[or wordt altijd](#) toegepast in combinatie met een zoekpatroon. We zijn bij [voorbeeld op](#) zoek naar alle cursussen die iets met SQL te maken hebben ([zie figuur](#)

115

```
SQL> select * from cursussen
  2  where omschrijving LIKE '%SQL%';

CODE OMSCHRIJVING          TYP    LENGTE
-----  -----
S02  Introductiecursus SQL      ALG      4
PLS  Introductie PL/SQL        BLD      1

SQL>
```

```
SQL> select mnr, voorl, naam  
2   from medewerkers  
3  where naam like '_A%';
```

MNR	VOORL	NAAM
7566	JM	JANSEN
7654	P	MARTENS
7900	R	JANSEN

```
-----  
7566 JM JANSEN  
7654 P MARTENS  
7900 R JANSEN
```

```
SQL>
```

4.29).

### Figuur 4.29

In combinatie met de LIKE-operator hebben twee karakters in het zoekpatroon een bijzondere betekenis; ze worden ook wel *wildcards* genoemd. Het zijn het procentteken ([zie figuur 4.29](#)) en het liggende streepje (de underscore), met respectievelijk de volgende betekenissen:

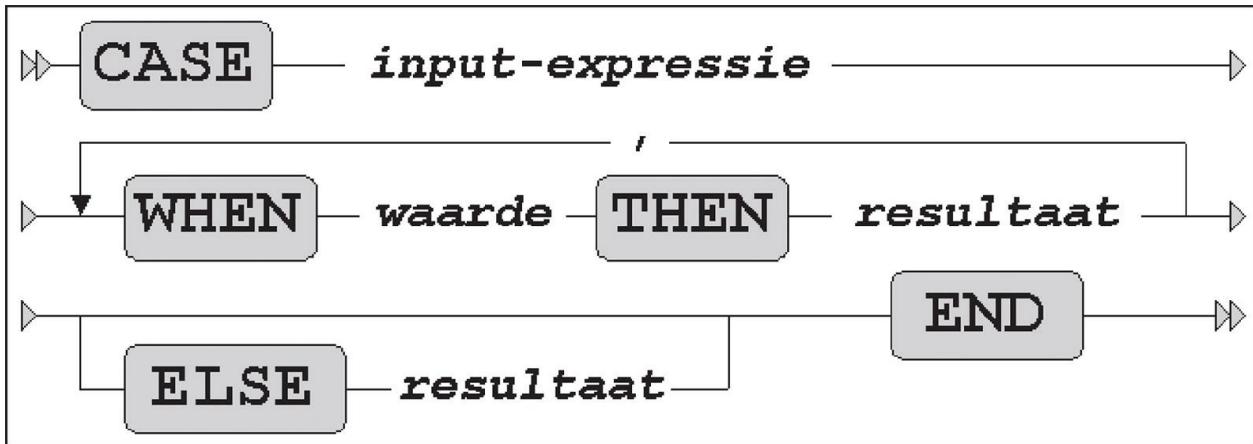
% nul, één of meer willekeurige karakters

—

precies één willekeurig karakter

Willen we deze karakters in een zoekpatroon tijdelijk van hun bijzondere betekenis ontdoen – bijvoorbeeld om te kunnen zoeken naar underscores – dan kan dat met behulp van de ESCAPE-optie van de LIKE-operator; zie de documentatie voor details. Geven deze twee wildcards onvoldoende mogelijkheden dan kunnen we gebruikmaken van de REGEXP\_LIKE-functie; [zie hoofdstuk 5, reguliere expressies](#).

De volgende query ([zie figuur 4.30](#)) levert alle medewerkers die als tweede karakter in hun naam een hoofdletter A hebben.



**Figuur 4.30**

Ook in dit geval – net als eerder bij de operatoren BETWEEN en IN – mag de ontkenning worden ingebouwd (WHERE ... NOT LIKE ...).

Tot slot beschouwen we nog twee vreemde gevallen:

1

select \* from medewerkers where naam like 'BLAAK'

2

select \* from medewerkers where naam = 'BL%'

Beide queries zullen geen foutmelding opleveren. Echter, in het eerste geval kan men beter de operator '=' gebruiken; in het tweede geval heeft het procentteken géén speciale betekenis, zodat de kans op 'no records selected' groot is.

4.7

## CASE-expressies

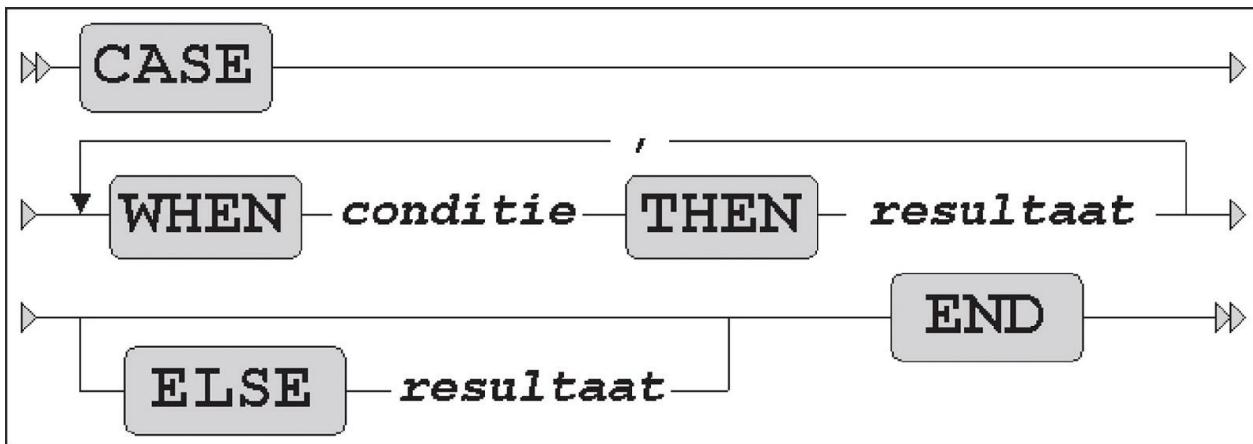
Met behulp van CASE-expressies kunnen vrij gecompliceerde

procedurele problemen worden aangepakt. Er zijn twee soorten CASE-expressies, in het Engels aangeduid als 'simple' ([zie figuur 4.31](#)) en

'searched' (zie [figuur 4.32](#)).

Figuur 4.31

117



- 
- 
- 

Figuur 4.32

Het verschil is als volgt: [in figuur 4.31 geven we een input-expressie die vervolgens met de waarden in de WHEN ... THEN-lus worden vergeleken.](#)

Dit is een simpele CASE-expression; de vergelijkingsoperator is steeds het gelijkheidsteken, de linker operand is steeds de input-expression, en de rechter operand is de waarde uit de WHEN-component. De kracht van de CASE-syntax [in figuur 4.32](#) is dat we in de WHEN ... THEN-lus volledige condities specificeren, waarbij we alle logische operatoren kunnen gebruiken. Met andere woorden: in iedere WHEN-component kunnen we een compleet nieuwe conditie specificeren.

Het resultaat van CASE-expressions komt als volgt tot stand: Oracle onderzoekt de WHEN-expressies in de volgorde zoals ze zijn gespecificeerd, en retourneert het THEN-resultaat van de *eerste* conditie die waar is. Let op: er wordt niet verder gezocht. De volgorde van de WHEN-expressies is dus belangrijk.

Levert geen van de WHEN-expressies een resultaat, dan wordt de ELSE-expressie geretourneerd.

In het geval dat er geen ELSE-expressie is gespecificeerd, wordt een null-waarde geretourneerd.

Het spreekt voor zich dat we consequent met datatypes moeten omgaan.

In figuur 4.31 moeten de input-expressies en de WHEN-waarden van hetzelfde datatype zijn, en in beide figuren (4.31 en 4.32) moeten de THEN-resultaten ook hetzelfde datatype hebben.

Van beide CASE-expressiemogelijkheden volgen voorbeelden, in figuur

4.33 en figuur 4.34. De resultaten van beide queries spreken voor zich.

```

SQL> select cursist, begindatum
  2 ,      case evaluatie
  3           when 1 then 'slecht'
  4           when 2 then 'matig'
  5           when 3 then 'ok'
  6           when 4 then 'goed'
  7           when 5 then 'zeer goed'
  8           else 'niet ingevuld'
  9       end
10  from  inschrijvingen
11  where cursus = 'S02';

CURSIST BEGIN DATUM CASE EVALUATIE
----- -----
7499 12-APR-99 goed
7934 12-APR-99 zeer goed
7698 12-APR-99 goed
7876 12-APR-99 matig
7788 04-OCT-99 niet ingevuld
7839 04-OCT-99 ok
7902 04-OCT-99 goed
7902 13-DEC-99 niet ingevuld
7698 13-DEC-99 niet ingevuld

SQL>

```

### Figuur 4.33

Merk overigens op dat Smit slechts 10% opslag krijgt, ondanks de vierde regel van de query in [figuur 4.34](#). Dat komt doordat hij trainer is, waardoor de tweede regel al resultaat oplevert; de overige WHEN-expressies worden dus niet meer bekeken.

CASE-expressies zijn erg krachtig en flexibel, maar soms ook nogal lang.

Vandaar dat er diverse Oracle-functies bestaan die je zou kunnen opvatten als afkortingen van CASE-expressies, zoals COALESCE en NULLIF

(beide functies zijn onderdeel van de ISO-standaard), NVL, NVL2 en DECODE. Deze functies komen in het volgende hoofdstuk aan de orde.

```

SQL> select naam, functie
  2 ,      case when functie = 'TRAINER' then ' 10%'
  3          when functie = 'MANAGER' then ' 20%'
  4          when naam      = 'SMIT'    then ' 30%'
  5          else ' 0%'
  6      end as opslag
  7 from medewerkers
  8 order by opslag desc, naam;

NAAM        FUNCTIE      OPSLAG
-----  -----
BLAAK        MANAGER     20%
CLERCKX      MANAGER     20%
JANSEN       MANAGER     20%
ADAMS        TRAINER     10%
SCHOTTEN     TRAINER     10%
SMIT         TRAINER     10%
SPIJKER      TRAINER     10%
ALDERS       VERKOPER    0%
DE KONING    DIRECTEUR   0%
DE WAARD     VERKOPER    0%
DEN DRAAIER   VERKOPER    0%
JANSEN       BOEKHOUDER  0%
MARTENS      VERKOPER    0%
MOLENAAR     BOEKHOUDER  0%

SQL>

```

**Figuur 4.34**

## 4.8

### NULL-waarden

Als een kolom voor een bepaalde rij geen waarde bevat, dan noemt men dat een *null-waarde*. Een null-waarde geeft aan dat er ergens informatie ontbreekt. Dit ontbreken van informatie kan diverse oorzaken hebben.

Een attribuut kan ‘niet van toepassing’ zijn; denk bijvoorbeeld aan de commissie voor niet-verkopers. Een waarde kan ook ‘onbekend’ zijn; bij het vullen van de database ontbraken toevallig bepaalde gegevens.

Een waarde kan ook ‘onbepaald’ zijn: men weet niet of het bewuste attribuut niet van toepassing of onbekend is.

Het is eigenlijk jammer dat we de *reden* waarom informatie ontbreekt niet kunnen weergeven. Als bijvoorbeeld een evaluatiewaarde in de tabel inschrijvingen ontbreekt, kan dat zijn omdat de cursus nog niet heeft plaatsgevonden, de cursist geen mening had, de cursist het formulier (nog) niet heeft ingeleverd, of dat de binnengekomen formulieren nog niet zijn verwerkt...

120

```
SQL> set null "<NULL>"  
SQL>
```

```
SQL> column evaluatie NULL "onbekend!"  
SQL> select * from inschrijvingen  
  2 where cursist = 7566;  
  
CURSIST CURS BEGINDATU EVALUATIE  
----- ----- -----  
 7566 JAV 01-FEB-00      3  
 7566 PLS 11-SEP-00 onbekend!  
SQL>
```

Men dient een null-waarde niet te verwarringen met het getal 0. Dat is namelijk wel dégelijk een waarde. De Nederlandse taal helpt in dit geval niet echt (null en nul); in het Engels (null en zero) is het verschil duidelijker. Om deze reden wordt null in het Nederlands vaak uitgesproken als ‘nil’. Ook een spatie is geen null-waarde; zelfs de lege string (‘’) is formeel iets anders dan een null-waarde, hoewel de lege string in bepaalde Oracle SQL-commando’s (nog) wel als null-waarde wordt opgevat ([zie hoofdstuk 6 over datamanipulatie](#)). Null-waarden kunnen in SQL het beste worden weergegeven met behulp van het gereserveerde woord NULL.

Op het beeldscherm wordt een null-waarde meestal door ‘niets’ weergegeven; zie bijvoorbeeld [de figuren 4.17 en 4.18. Daar kunnen we](#) in SQL\*Plus op twee manieren invloed op uitoefenen. Allereerst kan dat

globaal met behulp van een environment setting.

### Figuur 4.35

Hiermee instrueren we SQL\*Plus, om voor elke kolomwaarde NULL die in het resultaat van een query voorkomt, de tekst '<NULL>' op het beeldscherm te tonen.

De weergave van null-waarden kan bovendien specifiek op kolomniveau worden geregeld met behulp van het SQL\*Plus-commando COLUMN, zoals uit [figuur 4.36](#) blijkt.

### Figuur 4.36

121

```
SQL> select mnr, naam, comm
  2  from medewerkers
  3  where comm > 400;

      MNR NAAM          COMM
----- -----
    7521 DE WAARD       500
    7654 MARTENS      1400

SQL> select mnr, naam, comm
  2  from medewerkers
  3  where comm <= 400;

      MNR NAAM          COMM
----- -----
    7499 ALDERS        300
    7844 DEN DRAAIER      0

SQL>
```

Laten we nu eens de resultaten van de twee queries van [figuur 4.37](#) met elkaar vergelijken.

## **Figuur 4.37**

We zouden bij de overgang van de eerste naar de tweede query misschien verwachten dat alle andere medewerkers in het resultaat verschijnen; we hebben immers de conditie omgekeerd. Dat blijkt echter niet het geval.

Als een conditie in SQL wordt uitgewerkt, is het resultaat daarvan waar (TRUE), onwaar (FALSE), of onbekend (UNKNOWN). Men zegt ook wel: SQL is gebaseerd op een *driewaardige logica*.

Alleen de rijen waarvoor de conditie ‘waar’ is, zullen in het resultaat terechtkomen. Dat ligt ook nogal voor de hand. In ons voorbeeld zijn er echter enkele rijen die als resultaat ‘onbekend’ opleveren, ongeacht de ontkenning. Deze rijen zullen dus in beide gevallen niet in het resultaat verschijnen.

De volgende stelling, die in het Engels het aardigst klinkt, is hiervan een gevolg:

**in SQL, NOT is not ‘not’**

```
SQL> select mnr, naam, comm
  2  from medewerkers
  3  where comm <= 400
  4  or    comm is null;
```

MNR	NAAM	COMM
7369	SMIT	<NULL>
7499	ALDERS	300
7566	JANSEN	<NULL>
7698	BLAAK	<NULL>
7782	CLERCKX	<NULL>
7788	SCHOTTEN	<NULL>
7839	DE KONING	<NULL>
7844	DEN DRAAIER	0
7876	ADAMS	<NULL>
7900	JANSEN	<NULL>
7902	SPIJKER	<NULL>
7934	MOLENAAR	<NULL>

SQL>

```
SQL> select naam, functie, maandsal, comm
  2  from medewerkers
  3  where comm is not null;
```

NAAM	FUNCTIE	MAANDSAL	COMM
ALDERS	VERKOPER	1600	300
DE WAARD	VERKOPER	1250	500
MARTENS	VERKOPER	1250	1400
DEN DRAAIER	VERKOPER	1500	0

SQL>

Als we op zoek zijn naar alle medewerkers behalve de gelukkigen met een commissie groter dan 400, dan moeten we gebruikmaken van de SQL-operator IS NULL, die we in het volgende voorbeeld introduceren ([zie figuur 4.38](#)).

### Figuur 4.38

Overigens beschikt Oracle SQL over een tweetal functies die specifiek bedoeld zijn om flexibel met null-waarden om te kunnen gaan (NVL en

NVL2); ze worden in het volgende hoofdstuk behandeld.

Ook de IS NULL-operator heeft – net als BETWEEN, IN en LIKE – een eigen ontkenningsmogelijkheid (zie [figuur 4.39](#)).

123

```
SQL> select naam, voorl  
2   from medewerkers  
3  where comm = comm;
```

NAAM	VOORL
ALDERS	JAM
DE WAARD	TF
MARTENS	P
DEN DRAAIER	JJ

```
SQL>
```

### Figuur 4.39

Merk op dat de IS NULL-operator altijd waar (TRUE) of onwaar (FALSE) oplevert; het resultaat van deze operator is nooit onbekend (UNKNOWN).

De IS NULL-operator heeft maar één operand: de kolomnaam die ervóór staat. De operator ‘=’ heeft echter een linker en een rechter operand. Let op het subtile verschil tussen de volgende twee queries:

```
select * from inschrijvingen where evaluatie IS null
```

```
select * from inschrijvingen where evaluatie = null
```

De ogenschijnlijk onschuldige verandering heeft gevolgen voor het resultaat. We krijgen geen foutmelding, want syntactisch is alles in orde.

De vergelijking van twee null-waarden door de operator ‘=’ levert echter als resultaat altijd onbekend op. Anders gezegd: je kunt niet zomaar stellen dat een null-waarde gelijk is aan een andere null-waarde.

Dus:

$\text{NULL} = \text{NULL}$

is onbekend

$\text{NULL IS NULL}$

is waar

Als we dit tot ons hebben laten doordringen, dan weten we ook waarom

[de query van figuur 4.40](#) niet gewoon alle rijen van de medewerkerstabel oplevert.

#### Figuur 4.40

In de logica noemen we een expressie die altijd waar oplevert een *tautologie*. Uit het voorbeeld blijkt dat sommige triviale waarheden uit 124

```
SQL> select * from inschrijvingen
  2  where evaluatie not in (1,2,3,NULL);
no rows selected
SQL>
```

de tweewaardige logica (zoals  $\text{COMM} = \text{COMM}$ ) in SQL niet meer opgaan!

Null-waarden zullen ons nog vaak last bezorgen. We dienen

voortdurend rekening te houden met hun eventuele bestaan, en moeten ons steeds afvragen op welke manier we ze in de bepaling van het resultaat behandeld willen hebben. Anders zal de correctheid van onze queries op zijn minst twijfelachtig zijn.

We hebben al gezien dat null-waarden in een expressie tot gevolg hebben dat de hele expressie  $\text{NULL}$  oplevert; we zullen nog zien wat functies met null-waarden doen.

Ook ontstaat regelmatig, meestal achteraf, discussie over de formulering van de oorspronkelijke vraag in de natuurlijke taal. Probeer daarom een vraag altijd eerst zo veel mogelijk aan te scherpen in het Nederlands, alvorens de query in SQL te formuleren.

Het is duidelijk dat er veel haken en ogen zitten aan het omgaan met ontbrekende informatie. Ted Codd, de ‘uitvinder’ van het relationele model, heeft in een van zijn boeken zelfs voorgesteld om *twee* soorten null-waarden te introduceren: ‘onbekend’ en ‘niet van toepassing’

(*applicable* en *inapplicable*). Dat zou dan leiden tot een vierwaardige logica (zie Codd, 1990).

Als u nog verder geïnteresseerd bent in de problematiek van null-waarden – of andere theoretische achtergronden van relationele databases – dan zijn de boeken van Chris Date beslist aan te raden (vooral zijn ‘Selected Writings’). Hij weet als geen ander op een begrijpelijke, boeiende wijze over dit soort onderwerpen te schrijven.

Een doordenkertje tot slot van deze paragraaf: waarom levert de query

[van figuur 4.41 geen resultaten](#) op? Dit is een van de opgaven aan het eind van dit hoofdstuk.

## Figuur 4.41

125

4.9

Subqueries

In een van de vorige paragrafen hebben we voor het eerst

kennisgemaakt met de IN-operator; daar gaan we nu mee verder. Stel dat we in verband met een gerichte mailing geïnteresseerd zijn in alle personen die ooit een bouwcursus hebben gevolgd. Dan zou deze query precies het gewenste resultaat opleveren:

```
select cursist  
from inschrijvingen  
where cursus in ('JAV', 'PLS', 'XML')
```

Toch is er een tweetal bezwaren tegen deze oplossing in te brengen. Ten eerste hebben we zelf in de tabel met cursusgegevens opgezocht om welke specifieke cursussen het gaat; in de vraagstelling wordt immers alleen over bouwcursussen gesproken. Dat lukt in onze kleine database nog wél, maar dat is in de praktijk meestal onmogelijk. Ten tweede is deze oplossing nogal star; als we de mailing volgend jaar wegens doorslaand succes willen herhalen, zullen we de query eventueel moeten aanpassen aan het dan actuele aanbod van bouwcursussen.

Een betere oplossing voor dit probleem is het toepassen van een *subquery*. We laten het raadplegen van de cursustabel aan Oracle over, door op de plaats van de verzameling cursuscodes (JAV, PLS en XML) een query te formuleren die de gewenste cursuscodes voor ons ophaalt; zie [figuur 4.42](#).

```
SQL> select cursist
  2  from inschrijvingen
  3  where cursus in (select code
  4                      from cursussen
  5                      where type = 'BLD');

CURSIST
-----
7499
7499
7499
7566
7566
7698
7782
7788
7788
7839
7876
7876
7900

SQL>
```

## Figuur 4.42

Nu zijn beide bezwaren van de eerste oplossing verdwenen. Het resultaat van de subquery – een aantal cursuscodes – wordt gesubstitueerd tussen de haakjes, en daarna wordt de hoofdquery uitgevoerd. In totaal komen 13 oudcursisten uit de bus. Merk op dat diverse cursisten meerdere malen in het resultaat voorkomen. Willen we cursisten hoogstens eenmaal in het resultaat zien, dan zouden we SELECT

DISTINCT ... kunnen gebruiken.

Het is onze verantwoordelijkheid om subqueries zodanig te formuleren dat er geen appels met peren worden vergeleken. De volgende variant op het voorgaande commando geeft bijvoorbeeld geen foutmelding, maar wel een vreemd resultaat ([zie figuur 4.43](#)). **Merk op dat** we hier evaluatiecijfers aan het vergelijken zijn met cursuslengtes; probeer deze query maar eens in het Nederlands onder woorden te brengen...



```

SQL> select cursist
  2  from inschrijvingen
  3 where EVALUATIE in (select LENGTE
  4                      from cursussen
  5                     where type = 'BLD');

CURSIST
-----
7900
7788
7839
7900
7521
7902
7698
7499
7499
7876

SQL>

```

```

SQL> select cursist
  2  from inschrijvingen
  3 where cursus in
  4       (select cursus, begindatum
  5        from uitvoeringen
  6        where locatie = 'UTRECHT');
      (select cursus, begindatum
       *
ERROR at line 4:
ORA-00913: too many values

SQL>

```

### Figuur 4.43

Oracle maakt (gelukkig?) geen onderscheid tussen zinnige en onzinnige vragen. Syntactisch gezien zijn er maar twee beperkingen: ten eerste moeten de datatypes overeenstemmen, en ten tweede mag de subquery niet te veel kolommen selecteren. Daarom gaat [de query van figuur 4.44](#)

fout.

### Figuur 4.44

Het is overigens wel degelijk mogelijk om met subqueries combinaties van waarden te vergelijken. [De query van figuur 4.44 was](#) namelijk een poging om alle cursisten te vinden die ooit in Utrecht een cursus hebben gevolgd.

128

```
SQL> select cursist
  2  from inschrijvingen
  3 where (cursus, begindatum) in
  4       (select cursus, begindatum
  5        from uitvoeringen
  6        where locatie = 'UTRECHT');

CURSIST
-----
7521
7900
7902

SQL>
```

Een cursus-uitvoering is in ons datamodel uniek door de combinatie van cursuscode en begindatum, dus we corrigeren de query als volgt (zie [figuur 4.45, derde regel](#)).

### Figuur 4.45

Ten slotte moet nog worden opgemerkt dat subqueries op hun beurt weer subqueries mogen bevatten. Dit principe (*nesting*) mag nagenoeg onbeperkt worden toegepast; er is wat Oracle betreft geen absolute limiet aan verbonden. Op een gegeven moment raakt wat onszelf betreft het overzicht natuurlijk verloren.

We hebben de subquery geïntroduceerd aan de hand van de IN-operator.

In principe kunnen we het verband tussen hoofd- en subquery ook leggen met behulp van een vergelijkingsoperator (=, <, >, <=, >=, <>).

Er is dan wel één belangrijk verschil: de subquery moet in dat geval precies

één rij produceren. Deze eis ligt voor de hand als we ons realiseren hoe deze operatoren functioneren: ze kunnen maar één waarde met één andere waarde vergelijken.

129

```
SQL> select naam, voorl, gbdatum  
  2  from medewerkers  
  3  where gbdatum > (select gbdatum  
  4                      from medewerkers  
  5                      where mnr = 7876);
```

NAAM	VOORL	GBDATUM
JANSEN	JM	02-APR-67
DEN DRAAIER	JJ	28-SEP-68
JANSEN	R	03-DEC-69

```
SQL>
```

```
SQL> select naam, voorl, gbdatum  
  2  from medewerkers  
  3  where gbdatum > (select gbdatum  
  4                      from medewerkers  
  5                      where mnr = 99999);
```

```
no rows selected
```

```
SQL>
```

## Figuur 4.46

Dit levert alle medewerkers op die jonger zijn dan medewerker 7876.

Omdat MNR de primaire sleutel van de medewerkerstabel is, zal er nooit meer dan één resultaat uit de subquery komen.

Als er géén medewerker blijkt te bestaan met het gegeven nummer, resulteert dat in de melding ‘no rows selected’. De subquery (die dus eigenlijk helemaal geen rijen retourneert) wordt behandeld alsof er één rij terugkomt, die bestaat uit een null-waarde. Anders geformuleerd: SQL behandelt deze situatie dus

net alsof er wél een medewerker met dat nummer bestaat, maar van wie de geboortedatum onbekend is. Dit klinkt misschien vreemd, maar is volledig conform de ANSI/ISO SQL-standaard. Kijk maar naar het resultaat in [figuur 4.47](#):

### Figuur 4.47

Als er toch meerdere rijen uit de subquery komen, dan reageert Oracle met de foutmelding [van figuur 4.44](#) (merk op dat er twee medewerkers Jansen heten).

130

```
SQL> select naam, voorl, gbdatum
  2  from medewerkers
  3  where gbdatum > (select gbdatum
  4                      from medewerkers
  5                      where naam = 'JANSEN');
where gbdatum > (select gbdatum
*
ERROR at line 3:
ORA-01427: single-row subquery returns more than one row

SQL>
```

### Figuur 4.48

Tot nu toe hebben we uitsluitend subqueries bekijken in de WHERE-component van een (hoofd-)query; Oracle SQL ondersteunt ook subqueries in de FROM-component en de SELECT-component van een

[query. In hoofdstuk 7 komen](#) we nog op subqueries terug.

## 4.10 Opgaven

1

Geef code en omschrijving van alle cursussen die precies vier dagen duren.

2

Geef alle medewerkers, alfabetisch gesorteerd op functie, en per functie op leeftijd (van jong naar oud).

3

Welke cursussen zijn in Utrecht en/of in Maastricht uitgevoerd?

4

Welke medewerkers hebben zowel de Java als de XML cursus gevolgd? Geef hun nummers.

5

Geef de naam en voorletters van alle medewerkers, behalve van R. Jansen.

6

Geef nummer, functie en geboortedatum van alle medewerkers die vóór 1960 geboren zijn, en trainer of verkoper zijn.

7

Geef de nummers van alle medewerkers die niet aan de afdeling opleidingen zijn verbonden.

8

Geef de nummers van alle medewerkers die de Java-cursus niet hebben gevolgd.

9

Welke medewerkers hebben voorvoegsels in hun naam?

10 Welke medewerkers hebben ondergeschikten? En welke niet?

11 Geef een overzicht van alle uitvoeringen van algemene cursussen (type ALG) in 1999.

12 Geef naam en voorletters van iedereen die ooit bij N. Smit een 131 cursus heeft gevolgd. Aanwijzing: gebruik subqueries, en werk vervolgens van binnen naar buiten. Dus: bepaal het nummer van N.

Smit, zoek dan naar de cursussen die hij heeft gegeven, etc.

13 Wat is de verklaring van het resultaat ‘no rows selected’ in figuur

4.41?

132

## **Hoofdstuk 5**

### **Raadpleging – functies**

Dit hoofdstuk is een logisch vervolg op het vorige hoofdstuk, en gaat nog steeds over raadpleging. In dit hoofdstuk komen functies en reguliere expressies aan de orde, waardoor het schrijven van queries gemakkelijker wordt.

Dit hoofdstuk behandelt overigens niet de groepsfuncties, waarmee gegevens kunnen worden geaggregeerd; dat gebeurt in hoofdstuk 8.

Oracle ondersteunt zeer veel functies; niet alleen de functies die onderdeel zijn van de ANSI/ISO-standaard, maar ook vele Oracle-specifieke toevoegingen.

In de eerste paragrafen komen respectievelijk de reken- en tekstfuncties aan bod.

Een aantal tekstfuncties ondersteunen het zoeken van patronen met behulp

van reguliere expressies; de vierde paragraaf gaat daarop in.

Na de reguliere expressies volgt de behandeling van

datumfuncties en conversiefuncties. De laatste paragraaf van dit hoofdstuk gaat over het zelf definiëren van functies met behulp van PL/SQL. Dit wordt vrij summier behandeld, omdat het

eigenlijk buiten het bestek van dit boek valt.

Het hoofdstuk wordt afgesloten met enkele opgaven.

## 5.1

### Inleiding

We hebben gezien dat SQL de volgende standaard-operatoren kent:

#### **Rekenkundig:**

+ – \* /

#### **Alfanumeriek:**

|| (concatenatie)

We kunnen daarnaast allerlei andere bewerkingen uitvoeren met behulp van functies. Functies zijn te herkennen aan het feit dat ze een naam hebben, gevolgd door één of meer argumenten (tussen haakjes). Over 133

---

het algemeen mogen deze argumenten op hun beurt weer expressies zijn, waarin zelfs opnieuw functies mogen voorkomen. In dat geval spreekt men van geneste functies. Soms zijn argumenten optioneel; met andere woorden, ze mogen worden weggelaten. In zulke gevallen hanteert Oracle een standaard (of default) waarde.

Functies mogen vrijwel overal in queries worden toegepast: in de SELECT-, de WHERE-, de GROUP BY-, de HAVING- en in de ORDER BY-component.

Er worden enkele eisen gesteld aan het datatype van argumenten, die meestal vanzelfsprekend zijn. Bovendien zal Oracle waar mogelijk impliciete datatype-conversie plegen. Dat betekent dat als een getal wordt aangetroffen op een plaats waar tekst wordt

verwacht, het getal alfanumeriek wordt geïnterpreteerd. Zodoende ontstaan alleen foutsituaties in gevallen waarin een resultaat ongedefinieerd is, zoals de wortel uit de naam van een medewerker.

**Tip:** Het is *niet* verstandig om het op deze impliciete datatype-conversie te laten aankomen. Gebruik zo nodig conversiefuncties; het maakt SQL beter leesbaar en robuuster.

Mede omdat hun aantal nogal groot is, zijn de functies verdeeld in een aantal groepen. Het criterium daarbij is het soort argumenten waarop je de functies kunt toepassen:

### **Rekenfuncties:**

Toepasbaar op numerieke gegevens

### **Tekstfuncties :**

Toepasbaar op alfanumerieke gegevens

### **Datumfuncties:**

Toepasbaar op datums

### **Algemene functies:**

Toepasbaar op ieder datatype

### **Conversiefuncties:**

Conversie naar een ander datatype

### **Groepsfuncties:**

Toepasbaar op verzamelingen waarden

De laatstgenoemde categorie wordt behandeld [in hoofdstuk 8, als](#) we de GROUP BY- en HAVING-component van het SELECT-commando

behandelen. Ze zijn daar meer op hun plaats. De overige functies worden hierna besproken.

134

5.2

Rekenfuncties

De rekenfuncties van Oracle zijn:

**ROUND(  $n[ ,m]$  )**

Rond  $n$  af op  $m$  decimale posities

**CEIL(  $n$  )**

Rond  $n$  naar boven af op geheel getal

**FLOOR(  $n$  )**

Rond  $n$  naar beneden af op geheel getal

**TRUNC(  $n[ ,m]$  )**

Kap  $n$  af op  $m$  decimale positie

**ABS(  $n$  )**

De absolute waarde van  $n$

**SIGN(  $n$  )**

–1, 0, 1 naar gelang  $n$  negatief, nul of positief

is

**SQRT(  $n$ )**

Vierkantswortel uit  $n$

**EXP(  $n$ )**

e ( $= 2,7182813\dots$ ) tot de  $n$ -de macht

**LN(  $n$ ), LOG(  $m,n$ )**

Natuurlijke logaritme en logaritme basis  $m$

**POWER(  $n,m$ )**

$n$  tot de  $m$ -de macht

**MOD(  $n,m$ )**

Rest na deling van  $n$  door  $m$

**SIN(  $n$ ), COS(  $n$ ),**

sinus, cosinus, en tangens van  $n$  ( $n$  in

**TAN(  $n$ )**

radialen)

**ASIN(  $n$ ), ACOS(  $n$ ),**

arcsinus, -cosinus en -tangens van  $n$

**ATAN(  $n$ )**

**SINH(  $n$ ), COSH(  $n$ ),**

sinus-, cosinus- en tangens hyperbolicus van

## TANH( *n* )

*n*

In de twee gevallen waarin *m* optioneel is geldt 0 (nul) als default-waarde, en zijn negatieve waarden voor *m* ook toegestaan, zoals blijkt uit de volgende voorbeelden (zie de figuren 5.1 tot en met 5.4).

135

```
SQL> select round(345.678), ceil(345.678), floor(345.678)
  2  from  dual;

ROUND(345.678) CEIL(345.678) FLOOR(345.678)
-----
          346           346           345

SQL> select round(345.678, 2)
  2 ,      round(345.678,-1)
  3 ,      round(345.678,-2)
  4 from  dual;

ROUND(345.678,2) ROUND(345.678,-1) ROUND(345.678,-2)
-----
          345.68           350           300

SQL>
```

```
SQL> select abs(-123), abs(0), abs(456)
  2 ,      sign(-123), sign(0), sign(456)
  3 from  dual;

ABS(-123) ABS(0) ABS(456) SIGN(-123) SIGN(0) SIGN(456)
-----
          123           0           456           -1           0           1

SQL>
```

```
SQL> select power(2,3), power(-2,3)
  2 ,      mod(8,3),   mod(13,0)
  3 from dual;

POWER(2,3) POWER(-2,3) MOD(8,3) MOD(13,0)
----- ----- -----
          8          -8          2         13

SQL>
```

Figuur 5.1

Figuur 5.2

Figuur 5.3

136

```
SQL> select mnr as oneven_mnr
  2 ,      naam
  3 from medewerkers
  4 where mod(mnr,2) = 1;

ONEVEN_MNR NAAM
-----
7369 SMIT
7499 ALDERS
7521 DE WAARD
7839 DE KONING

SQL>
```

```

SQL> select naam
  2 ,      floor((sysdate-gbdatum)/7)    as weken
  3 ,      floor(mod(sysdate-gbdatum,7)) as dagen
  4 from medewerkers
  5 where afd = 10;

```

NAAM	WEKEN	DAGEN
CLERCKX	2014	4
DE KONING	2669	6
MOLENAAR	2190	5

SQL>

```

SQL> select sin(30*3.14159265/180), tanh(0.5)
  2 ,      exp(4), log(2,32), ln(32)
  3 from dual;

SIN(30*3.14159265/180) TANH(0.5)    EXP(4)  LOG(2,32)    LN(32)
----- ----- ----- -----
          .5     .4621172   54.59815      5 3.465736

```

SQL>

## Figuur 5.4

In het volgende voorbeeld rekenen we in weken en (additionele) dagen uit hoe oud de medewerkers van afdeling 10 zijn. We gebruiken hierbij de pseudo-kolom SYSDATE, dus uw resultaten zullen ongetwijfeld anders zijn dan in [figuur 5.5](#).

## Figuur 5.5

## Figuur 5.6

137

5.3

Tekstfuncties

De voornaamste tekstfuncties van Oracle zijn:

**LENGTH( *t* )**

Aantal karakters (lengte) van *t*

**ASCII( *t* )**

Ascii-waarde eerste karakter van *t*

**CHR( *n* )**

Karakter met ascii-waarde *n*

**UPPER( *t* ), LOWER( *t* )**

*t* in hoofdletters/kleine letters

**INITCAP( *t* )**

Ieder woord in *t* beginnend met een

hoofdletter;

de rest van ieder woord in kleine letters

**LTRIM( *t*, *k* )**

Haal links een stuk van *t* af,

tot en met het eerste karakter niet in *k*

**RTRIM( *t*, *k* )**

Haal rechts een stuk van *t* af,

na het laatste karakter niet in *k*

**LPAD( *t*, *n*[, *k*] )**

Vul  $t$  links uit met karakter  $k$  tot lengte  $n$  **RPAD(  $t, n[, k]$ )**

Idem, maar dan rechts (default  $k$ : spatie)

**SUBSTR(  $t, n[, m]$ )**

Gedeelte (substring) van  $t$  vanaf positie  $n$ ,  $m$  karakters lang (default: tot einde)

**INSTR(  $t, k$ )**

Positie van het eerste voorkomen van  $k$  in  $t$

**INSTR(  $t, k, n$ )**

Idem, op of na de  $n$ -de positie in  $t$

**INSTR(  $t, k, n, m$ )**

Idem, maar nu het  $m$ -de voorkomen van  $k$

**TRANSLATE(  $t, v, w$ )**

Vervang alle karakters uit  $v$  die in  $t$

voorkomen

door het corresponderende karakter uit  $w$

**REPLACE(  $t, v$ )**

Verwijder uit  $t$  elk voorkomen van  $v$

**REPLACE(  $t, v, w$ )**

Vervang elk voorkomen van string  $v$  in  $w$

**CONCAT(  $t1, t2$ )**

Concateneer  $t1$  en  $t2$  (equivalent met  $\parallel$ )

Posities in strings worden altijd geteld vanaf één (1).

[Figuur 5.7](#) illustreert de functies LOWER, UPPER, INITCAP en LENGTH;

[figuur 5.8 de functies](#) ASCII en CHR. Merk op dat de ASCII-functie uitsluitend kijkt naar het eerste karakter, ook al wordt een string meegegeven.

[Figuur 5.9 geeft een voorbeeld van de functies INSTR en SUBSTR; figuur](#)

[5.10 van LTRIM en RTRIM, en figuur 5.11 van LPAD en RPAD.](#)

138

```
SQL> select lower(functie), initcap(naam)
  2  from medewerkers
  3  where upper(functie) = 'VERKOPER'
  4  order by length(naam);
```

LOWER(FUNC INITCAP(NAAM

```
-----  
verkoper Alders  
verkoper Martens  
verkoper De Wارد  
verkoper Den Draaier
```

SQL>

```
SQL> select ascii('a'), ascii('z')
  2 ,      ascii('A'), ascii('Z')
  3 ,      ascii('ABC'), chr(77)
  4 from dual;
```

ASCII('A') ASCII('Z') ASCII('A') ASCII('Z') ASCII('ABC') CHR(77)

```
-----  
97       122      65      90      65 M
```

SQL>

```

SQL> select naam, substr(naam,4), substr(naam,4,3)
  2 ,      instr(naam,'O'), instr(naam,'O',3), instr(naam,'O',3,2)
  3 from afdelingen;

          SUBSTR           INSTR           INSTR           INSTR
NAAM      (NAAM,4)      SUB (NAAM, 'O') (NAAM, 'O', 3) (NAAM, 'O', 3,2)
-----
HOOFDKANTOOR    FDKANTOOR    FDK        2            3            10
OPLEIDINGEN     EIDINGEN     EID        1            0            0
PERSONEELSZAKEN SONEELSZAKEN SON        5            5            0
VERKOOP          KOOP          KOO        5            5            6

SQL>

```

**Figuur 5.7**

**Figuur 5.8**

**Figuur 5.9**

De uitvoer in [figuur 5.9](#) is enigszins aangepast, om de leesbaarheid te verhogen.

139

```

SQL> select distinct naam
  2 ,      ltrim(naam,'S'), rtrim(naam,'S')
  3 from medewerkers
  4 where afd = 20;

NAAM      LTRIM(NAAM, 'S')  RTRIM(NAAM, 'S')
-----
ADAMS     ADAMS          ADAM
JANSEN    JANSEN         JANSEN
SCHOTTEN CHOTTEN        SCHOTTEN
SMIT      MIT            SMIT
SPIJKER   PIJKER         SPIJKER

SQL>

```

```
SQL> select lpad(naam,15,'@')
  2 ,      rpad(naam,10,'=')
  3 from medewerkers
  4 where afd = 30;

LPAD(NAAM,15,'@') RPAD(NAAM,10,'=')
-----
@@@ALDERS      ALDERS=====
@@@DE WAARD    DE WAARD==
@@@MARTENS     MARTENS===
@@@BLAAK       BLAAK=====
@@@DEN DRAAIER DEN DRAAIE
@@@JANSEN      JANSEN=====

SQL>
```

## Figuur 5.10

Merk op dat de functies LPAD en RPAD niet alleen strings langer kunnen maken, zoals hun naam ook suggereert (to pad = aanvullen), maar ook korter. Kijk maar wat er gebeurt met ‘DEN DRAAIER’ in [figuur 5.11](#).

## Figuur 5.11

Als we de functies LPAD en RPAD in plaats van een constante een expressie met variabelen als tweede argument meegeven, kunnen we er bijvoorbeeld een salaris-histogram mee produceren ([zie figuur 5.12](#)).

```
SQL> select lpad(maandsal,4)||' '||
  2      rpad('o',maandsal/100,'o') as histogram
  3  from medewerkers
  4 where afd = 30;
```

HISTOGRAM

```
-----  
1600 oooooooooooooooo  
1250 oooooooooooooo  
1250 oooooooooooooo  
2850 ooooooooooooooooooooooo  
1500 oooooooooooooooo  
800 oooooooo
```

```
SQL>
```

```
SQL> select translate('melkemmer','melk','bier') as translate
  2 ,      replace ('melkemmer','melk','bier') as replace_1
  3 ,      replace ('melkemmer','melk')          as replace_2
  4 from dual;
```

TRANSLATE REPLACE\_1 REPLACE\_2

```
-----  
bieribbir bieremmer emmer
```

```
SQL>
```

## Figuur 5.12

[Figuur 5.13](#) laat het verschil zien tussen de functies REPLACE en TRANSLATE: TRANSLATE vervangt individuele karakters. REPLACE biedt echter de mogelijkheid om woorden te vervangen door andere woorden.

Merk op wat er gebeurt als de functie REPLACE slechts twee argumenten meekrijgt in plaats van drie; dan worden er woorden verwijderd.

## Figuur 5.13

5.4

Reguliere expressies

In het vorige hoofdstuk hebben we kennismegemaakt met de LIKE operator, en in een eerdere paragraaf van dit hoofdstuk met de INSTR en SUBSTR functies. Alle drie zoeken ze naar tekst, waarbij de LIKE-operator beschikt over twee wildcards (%) en (\_). Voor geavanceerde zoekoperaties is deze functionaliteit soms onvoldoende. Oracle 141

- 
- 
- 
- 
-

<b>Operator</b>	<b>Omschrijving</b>
*	Nul of meer occurrences
+	Een of meer occurrences
?	Nul of een occurrence
	Operator om alternatieven van elkaar te scheiden
^	Het begin van de regel
\$	Het einde van de regel
.	Een willekeurig karakter
[ ]	Een lijst van mogelijkheden; een circumflex (^) aan het begin van de lijst werkt als ontkenning
( )	Groepeert een expressie, om naar te kunnen refereren
{m}	Precies m keer
{m, }	Minstens m keer
{m, n}	Minstens m maar niet meer dan n keer
\n	(n is een cijfer tussen 1 en 9) refereert terug naar de n-de subexpressie-tussen haakjes, voorafgaande aan de \n

- 
- 
- 
- 
- 

ondersteunt daarom de volgende functies die met *reguliere expressies* werken:

REGEXP\_LIKE;

REGEXP\_INSTR;

REGEXP\_COUNT;

REGEXP\_SUBSTR;

REGEXP\_REPLACE.

Reguliere expressies zijn onder andere bekend in de diverse smaken van het Unix-besturingssysteem, en zijn onderdeel van de POSIX-standaard.

Reguliere expressies zijn uitgebreid gedocumenteerd in de SQL Reference; we volstaan hier met de belangrijkste karakters met hun bijzondere betekenis [in figuur 5.14](#).

### Figuur 5.14

De syntax van deze functies die met reguliere expressies overweg kunnen, is als volgt:

REGEXP\_LIKE(tekst, patroon, opties);

REGEXP\_INSTR(tekst, patroon, positie, optreden, return-keuze, opties);

REGEXP\_COUNT(tekst, patroon, positie, opties);

REGEXP\_SUBSTR(tekst, patroon, positie, optreden, opties);

REGEXP\_REPLACE(tekst, patroon, vervanger, positie, optreden, 142

opties).

Hierbij zijn in alle gevallen de eerste twee argumenten (tekst en zoekpatroon) verplicht en zijn de overige argumenten optioneel, met dien verstande dat ze alleen maar van rechts naar links mogen worden weggelaten. Als we bijvoorbeeld het opties-argument van de REGEXP\_INSTR-functie willen gebruiken, dan zijn alle andere argumenten verplicht.

De tekst en het patroon (met daarin de reguliere expressies) spreken voor zich; met het opties-argument kunnen we het gedrag van de functies als volgt beïnvloeden:

i

Case-insensitive zoeken (ongevoelig voor hoofd- en kleine letters)

c

Case-sensitive zoeken (gevoelig voor hoofd- en kleine letters) n

Een newline valt ook onder de punt (.) als willekeurig karakter m

Behandel tekst als meerdere regels; ^ en \$ werken ook als

begin of eind van een van die regels, in plaats van het begin of eind van de hele tekst

Het standaard gedrag wat betreft de gevoeligheid voor hoofd- en kleine letters hangt af van de NLS\_SORT parameter.

In REGEXP\_INSTR en REGEXP\_SUBSTR kunnen we met positie aangeven vanaf welke positie we in de tekst willen beginnen te zoeken (standaard 1) en met optreden kunnen we aangeven hoe vaak we het patroon willen zoeken (standaard 1).

Met de return-keuze van REGEXP\_INSTR kunnen we als volgt invloed uitoefenen op het resultaat van deze functie:

0

De positie van het eerste karakter van de vondst (dit is de standaard)

1

De positie van het eerste karakter dat volgt na de vondst

Laten we eerst de REGEXP\_LIKE functie eens onderzoeken. Dat gaat het eenvoudigst met een SQL\*Plus truc die in een later hoofdstuk wordt uitgelegd. De ampersand (&) in [figuur 5.15 zorgt](#) er voor dat SQL\*Plus vraagt om een waarde voor tekst; we kunnen hetzelfde SQL-143

```

SQL> select 'gevonden!' as resultaat from dual
2 where regexp_like('&tekst', '^.{1,2}.+$', 'i');

Enter value for tekst: bar

RESULTAAT
-----
gevonden!

SQL> /
Enter value for tekst: PAARD

RESULTAAT
-----
gevonden!

SQL> /
Enter value for tekst: ja

no rows selected

```

commando dus steeds herhalen met een andere waarde voor tekst en zodoende de werking van het patroon onderzoeken.

### Figuur 5.15

Uit de resultaten blijkt dat dit patroon het volgende betekent: het eerste karakter is willekeurig, gevolgd door minstens een en hoogstens twee a's, gevolgd door een of meer willekeurige karakters, en dit alles zonder te letten op het verschil tussen hoofden kleine letters. Uit de query in

[figuur 5.15](#) blijkt overigens dat de REGEXP\_LIKE functie een logische (Boolean) functie is; het resultaat is WAAR of ONWAAR.

In [figuur 5.16 zoeken we met](#) de REGEXP\_INSTR functie naar (minstens) acht woorden, of beter: naar minstens acht niet-lege (+) substrings die geen spatie ([^ ]) bevatten. We gebruiken tegelijk REGEXP\_COUNT om het daadwerkelijk aantal woorden te tonen.

```

SQL> select opmerkingen
  2      ,regexp_count(opmerkingen, '[^ ]+', 1) as aantal
  3  from historie
  4 where regexp_instr(opmerkingen, '[^ ]+', 1, 8) > 0;

OPMERKINGEN                                AANTAL
-----
Niet zo geschikt als docent; dan maar naar verkoop!      9
Senior verkoper; zou wel eens een aanwinst kunnen zijn?      9
Project (van een halve maand) voor het hoofdkantoor      8

SQL>

```

```

SQL> select opmerkingen
  2      ,      regexp_substr(opmerkingen, '\([^\)]+\)') as substring
  2  from historie
  3 where opmerkingen like '%(%';

OPMERKINGEN
-----
SUBSTRING
-----
Project (van een halve maand) voor het hoofdkantoor
(van een halve maand)

SQL>

```

## Figuur 5.16

In figuur 5.17 zoeken we met de REGEXP\_SUBSTR functie naar opmerkingen tussen haakjes. Het patroon zoekt naar een haakje openen, gevolgd door minstens één karakter ongelijk aan een haakje sluiten, gevolgd door een haakje sluiten. Merk op dat we de backslash (\) nodig hebben om de haakjes van hun speciale betekenis te ontdoen.

## Figuur 5.17

In figuur 5.18 gebruiken we REGEXP\_REPLACE om tussen alle karakters in een cursusomschrijving een extra spatie tussen te voegen.

```
SQL> select regexp_replace(omschrijving, '(.)', '\1 ') as breder
  2  from cursussen;

BREDER
-----
Introductie cursus SQL
Oracle voor applicatiegebruikers
Java voor Oracle ontwikkelaars
Introductie PL/SQL
XML voor Oracle ontwikkelaars
Datamodellering met ERM
Procesmodelleringstechnieken
Relationeel systeemontwerp
Prototyping
Systeemgeneratie

10 rows selected.

SQL>
```

- 
- 
- 
- 
- 
- 
- 
- 
- 
- 

## Figuur 5.18

5.5

### Datumfuncties

Alvorens de diverse datumfuncties te bespreken bekijken we eerst de manier waarop we in SQL tijdgerelateerde constanten kunnen weergeven:

DATE ‘yyyy-mm-dd’;

TIMESTAMP ‘yyyy-mm-dd hh24:mi:ss.ff [AT TIME ZONE ...];

INTERVAL ‘expr’ <qualifier>.

Voorbeelden:

DATE ‘2004-09-25’;

TIMESTAMP ‘2004-09-25 23:59:59.99999’ AT TIME ZONE ‘CET’;

INTERVAL ‘1’ YEAR;

INTERVAL ‘1 2:3’ DAY TO MINUTE.

De laatste expressie hierboven lichten we kort toe: het betreft een interval dat 1 dag, 2 uur en 3 minuten lang is.

We kunnen hier uitgebreid mee experimenteren met behulp van de volgende query, gebruikmakend van de ampersand (&)

substitutiemethode die we in de vorige paragraaf ook al hebben gezien: SQL> **select <constante expressie> from dual;** 146

Als we alleen maar een alfanumerieke string ingeven, dan moeten we vertrouwen op een impliciete conversie door Oracle. Of deze impliciete conversie slaagt of faalt is afhankelijk van de NLS\_DATE\_FORMAT- en NLS\_TIMESTAMP\_FORMAT-parameterinstellingen. Als we een overzicht willen van alle NLS-parameterwaarden die voor onze sessie van kracht zijn, kunnen we die opvragen met behulp van de volgende query: SQL> **select \* from nls\_session\_parameters;**

We gaan hier verder niet op de resultaten van deze query in.

De belangrijkste datumfuncties van Oracle zijn:

**ADD\_MONTHS( *d, n*)**

Datum *d* plus *n* maanden

**MONTHS\_BETWEEN( *d,***

Maanden verschil tussen datum  $d$  en datum  $e$

$e)$

**LAST\_DAY(  $d$  )**

Laatste dag van de maand waarin datum  $d$

valt

**NEXT\_DAY(  $d$ ,**

De eerste *weekdag* (ma, di, ...) na datum  $d$

*weekdag*)

**NEW\_TIME(  $d$ , z1, z2)**

Converteer datum/tijd van tijdzone z1 naar z2

**ROUND(  $d$ [,  $fmt$ ])**

Datum  $d$  afgerond op  $fmt$  (default

middernacht)

**TRUNC(  $d$ [,  $fmt$ ])**

Datum  $d$  afgekapt op  $fmt$  (default

middernacht)

**EXTRACT(  $c$  FROM  $d$ )**

Extraheer component  $c$  uit datum d

Om met de laatste functie te beginnen: met de EXTRACT-functie kunnen we allerlei componenten extraheren uit datums of andere

tijdgerelateerde expressies. Afhankelijk van het type van de waarde  $d$  (datum,

tijdstip, of interval) zijn de volgende waarden van *c* mogelijk: YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, TIMEZONE\_ABBR, ...

[Figuur 5.19 geeft](#) een toepassingsvoorbeeld van de EXTRACT-functie.

147

```
SQL> select extract(year  from gbdatum) as jaar
  2 ,      extract(month from gbdatum) as maand
  3 ,      extract(day   from gbdatum) as dag
  4 from   medewerkers
  5 where  naam = 'DE KONING';
```

JAAR	MAAND	DAG
1952	11	17

```
SQL>
```

CC, SCC	Eeuw, met of zonder minteken (BC)
[S]YYYY, [S]YEAR, YYY, YY, Y	Jaar (in allerlei gedaantes)
IYYY, IYY, IY, I	ISO jaar
Q	Kwartaal
MONTH, MON, MM, RM	Maand (voluit, afgekort, getal, Romeins getal)
IW, WW	(ISO) weeknummer
W	Dag van de week
DDD, DD, J	Dag (jaar/maand/Juliaans)
DAY, DY, D	Dichtstbijzijnde zondag
HH, HH12, HH24	Uur
MI	Minuut

```
SQL> select naam, months_between(sysdate, gbdatum)
  2  from medewerkers
  3 where afd = 10;
```

NAAM	MONTHS_BETWEEN(SYSDATE, GBDATUM)
CLERCKX	463.3166
DE KONING	614.0586
MOLENAAR	503.865

SQL>

## Figuur 5.19

De volgende tabel geeft de datumformaten (FMT) die door de datumfuncties ROUND en TRUNC worden ondersteund. Het default-formaat is ‘D D’, hetgeen afronden of afkappen op middernacht tot gevolg heeft.

## Figuur 5.20

```

SQL> select add_months('29-JAN-1996', 1) add_months_1
  2 ,      add_months('29-JAN-1997', 1) add_months_2
  3 ,      add_months('11-AUG-1997',-3) add_months_3
  4 from   dual;

ADD_MONTHS_1 ADD_MONTHS_2 ADD_MONTHS_3
-----
29-FEB-1996 28-FEB-1997 11-MAY-1997

SQL>

```

```

SQL> select next_day(sysdate,'SAT') as next_day
  2 ,      last_day(sysdate)          as last_day
  3 ,      round(sysdate,'YY')       as round
  4 ,      trunc(sysdate,'CC')       as trunc
  5 from   dual;

NEXT_DAY      LAST_DAY      ROUND      TRUNC
-----
24-JAN-2004 31-JAN-2004 01-JAN-2004 01-JAN-2001

SQL>

```

## Figuur 5.21

Let op wat er in [figuur 5.21](#) met een niet-schrikkeljaar gebeurt. En er is nog iets aan de hand met de query in [figuur 5.21](#); we zouden namelijk best wel eens een foutmelding kunnen krijgen. Dit komt doordat we vertrouwen op een impliciete conversie van de drie strings naar een datum. Of dit lukt hangt van de instelling van NLS\_DATE\_FORMAT af, zoals we eerder in deze paragraaf al hebben gezien. Eigenlijk zou het beter zijn om de datums in deze query aan te geven met behulp van het sleutelwoord DATE (zie het begin van deze paragraaf) of met de TO\_DATE-conversiefunctie; zie een van de volgende paragrafen voor details.

## Figuur 5.22

5.6

Algemene functies

De belangrijkste algemene (datatype-onafhankelijke) functies zijn:  
**GREATEST( a, b, ... )**

Grootste waarde uit de argumenten

149

```
SQL> select greatest(12*6,148/2,73)
  2 ,      least    (12*6,148/2,73)
  3 from   dual;

GREATEST(12*6,148/2,73) LEAST(12*6,148/2,73)
-----
          74                  72

SQL>
```

**LEAST( a, b, ... )**

Kleinste waarde uit de argumenten

**NULLIF(a, b)**

Geef NULL als a=b, anders a

**COALESCE(a, b, ... )**

Retourneer het eerste argument dat niet NULL

is (en NULL als ze allemaal NULL zijn)

**NVL( x, y )**

Geef y als x NULL is, anders x zelf **NVL2( x, y, z )**

Geef y als x niet NULL is, anders z

**DECODE( x, a1, b1,**

Geef:  $b_1$  als  $x = a_1$ ,

$a_2, b_2,$

$b_2$  als  $x = a_2, \dots$

$\dots, a_n, b_n$

$b_n$  als  $x = a_n,$

$[, y])$

en anders  $y$  (of default: NULL)

Al deze functies zijn ook uit te drukken als een CASE-expressie, omdat ze allemaal procedureel van aard zijn. We hebben ze dus eigenlijk niet nodig. Soms zijn ze toch prettig in het gebruik, bijvoorbeeld omdat ze compactere code opleveren. We moeten ons ook realiseren dat (naast CASE-expressies) alleen de NULLIF- en COALESCE-functies onderdeel zijn van de ANSI/ISO-standaard; de andere vijf functies (GREATEST, LEAST, NVL, NVL2 en DECODE) zijn specifiek voor Oracle SQL. Willen we overdraagbare SQL-code schrijven, dan moeten we daar dus rekening mee houden.

Het herschrijven van diverse commandovoорbeelden in deze paragraaf naar het gebruik van CASE, NULLIF en COALESCE wordt aan de lezers overgelaten in de opgaven aan het einde van dit hoofdstuk.

### **Figuur 5.23**

```
SQL> select naam, maandsal, comm
  2 ,      12*maandsal+nvl(comm,0) as jaarsal
  3 from medewerkers
  4 where naam like '%T%';
```

NAAM	MAANDSAL	COMM	JAARSAL
SMIT	800		9600
MARTENS	1250	1400	16400
SCHOTTEN	3000		36000

```
SQL>
```

```

SQL> select functie, naam
  2 ,      decode(greatest(maandsal,2500)
  3 ,2500,'goedkoop','duur') as klasse
  4 from medewerkers
  5 where gbdatum < to_date('01/01/1964','dd/mm/yyyy')
  6 order by decode(functie,'DIRECTEUR',1,'MANAGER',2,3);

```

FUNCTIE	NAAM	KLASSE
DIRECTEUR	DE KONING	duur
MANAGER	BLAAK	duur
VERKOPER	ALDERS	goedkoop
VERKOPER	DE WAARD	goedkoop
BOEKHOUDER	MOLENAAR	goedkoop
TRAINER	SPIJKER	duur
TRAINER	SCHOTTEN	duur
VERKOPER	MARTENS	goedkoop

SQL>

## **Figuur 5.24**

De NVL-functie wordt toegepast op componenten van een expressie om te voorkomen dat de expressie als geheel NULL oplevert.

De functie DECODE is een typisch overblijfsel uit de tijd dat Oracle nog geen CASE-expressies ondersteunde. De leesbaarheid van de DECODE-functie is nogal gering, en CASE-expressies zijn ook nog krachtiger, dus we kunnen het gebruik van de DECODE-functie beter vermijden.

## **Figuur 5.25**

5.7

Conversiefuncties

151

```

SQL> select 123, to_char(123)
  2 ,      to_char(123,'$09999.99')
  3 ,      to_number('123')
  4 from   dual;

    123 TO_ TO_CHAR(12 TO_NUMBER('123 ')
-----
123 123 $00123.00          123

SQL>

```

De belangrijkste conversiefuncties van Oracle zijn:

### **TO\_CHAR( *n*[, *fmt*])**

Zet getal *n* om naar een string

### **TO\_CHAR( *d*[, *fmt*])**

Zet datum *d* om naar een string

### **TO\_NUMBER( *t*)**

Zet string *t* om naar een getal

### **TO\_DATE ( *t*[, *fmt*])**

Zet string *t* om naar een datum

Het probleem met het datatype DATE is dat het op papier of op het scherm uitsluitend kan worden weergegeven als een string. Dat doet Oracle dan ook, volgens een default-datumformaat. Dit default-formaat is instelbaar met behulp van de parameter NLS\_DATE\_FORMAT, zoals in

[hoofdstuk 2 al](#) is behandeld.

Andersom kunnen we eigenlijk ook geen datums via een toetsenbord invoeren. We kunnen een string invoeren, samen met een afspraak (formaat) over hoe deze string als datum moet worden opgevat. Als we het default-

formaat hanteren, kan het explicet aangeven van dit formaat achterwege blijven.

Hoe we het moeten aanpakken als we van de standaard willen afwijken, blijkt uit de voorbeelden van de figuren 5.26 en 5.27.

## Figuur 5.26

152

```
SQL> select sysdate          as vandaag
  2 ,      to_char(sysdate,'hh24:mi:ss') as tijd
  3 ,      to_char(to_date('01/01/2005','dd/mm/yyyy')
  4 ,           '"valt op "Day') as nieuwjaar_2005
  5 from dual;

VANDAAG      TIJD      NIEUWJAAR_2005
-----
18-JAN-2004 20:26:16 valt op Zaterdag

SQL>
```

## Figuur 5.27

Merk overigens op dat we in het Nederlands worden aangesproken ('Zaterdag'). Dit komt doordat de parameter NLS\_LANGUAGE de waarde 'Dutch' heeft.

Met betrekking tot conversiefuncties (TO\_CHAR en TO\_DATE) en datums zijn onder andere de volgende formaten mogelijk:

[S]CC

Eeuw; S voor het minteken (BC)

[S]YYYY

Jaartal, met of zonder minteken

**YYY, YY, Y**

Laatste 3, 2 of 1 cijfer(s) van het jaartal

**[S]YEAR**

Jaartal uitgespeld, met minteken (S)

**BC, AD**

BC/AD-indicator

**Q**

Kwartaal (1,2,3,4)

**MM**

Maand (01 - 12)

**MONTH**

Maandnaam, met spaties uitgevuld tot lengte 9

**MON**

Drieletterige afkorting maand

**WW, IW**

(ISO) weeknummer (01-52)

**W**

Weeknummer van de maand (1-5)

**DDD**

Dagnummer van het jaar (1-366)

**DD**

Dagnummer van de maand (1-31)

**D**

Dagnummer van de week (1-7)

**DAY**

Dagnaam, met spaties uitgevuld tot lengte 9

**DY**

Drieletterige afkorting van de dag

**J**

Juliaanse datum; dagnummer sinds 01/01/4712 BC

**AM, PM**

AM/PM-indicator

**HH[12]**

Uur van de dag (01-12)

153

```
SQL> select to_char(sysdate,'DAY dy Dy') as dag
  2 ,      to_char(sysdate,'MONTH mon') as maand
  3 from dual;

DAG          MAAND
-----
ZONDAG      zo  Zo JANUARI    jan

SQL>
```

**HH24**

Uur van de dag (00-23)

**MI**

Minuten (00-59)

**SS**

Seconden (00-59)

**SSSS**

Seconden na middernacht (0-86399)

/.,

Deze leestekens worden letterlijk in de datum

weergegeven

“ ... ”

String tussen aanhalingstekens wordt in de datum

weergegeven

Verder zijn er nog enkele toevoegingen mogelijk:

**FM**

Fill mode

**TH**

Ordinaalgetal

(4th)

**SP**

Uitgespeld getal

(four)

**THSP, SPTH**

Uitgespeld ordinaalgetal

(fourth)

In *fill-mode* wordt aanvulling met spaties onderdrukt, en worden geen voorloopnullen aan getallen toegevoegd. We kunnen dit fill-mechanisme binnen dezelfde string aanen uitzetten door FM te herhalen (het is een *toggle*). Ordinaalgetallen zijn aanduidingen van rangorde.

Denk eraan dat de formaten upper/lowercase-gevoelig zijn; zie figuur

5.28.

**Figuur 5.28**

Wat datatype-conversie betreft kan veel aan Oracle worden overgelaten.

Uit het oogpunt van duidelijkheid is het echter beter om de conversies met de juiste functies explicet aan te geven. Let bijvoorbeeld op de

query van figuur 5.29:

154

```
SQL> select naam, substr(gbdatum,8)+16
  2  from medewerkers
  3  where afd = 10;
```

NAAM	SUBSTR (GBDATUM, 8) +16
CLERCKX	1981
DE KONING	1968
MOLENAAR	1978

```
SQL>
```

## Figuur 5.29

Deze query zal door Oracle intern als volgt worden uitgevoerd: SQL> select naam, **to\_number**(substr(to\_char(gbdatum),8))+16

2

from medewerkers

3

where afd = 10;

Dus hadden we het commando ook beter zo kunnen formuleren.

5.8

## Opgeslagen functies

Alsof Oracle nog niet genoeg functies ondersteunt, is het mogelijk om zelf functies aan SQL toe te voegen met behulp van PL/SQL, de standaardprogrammeertaal in de Oracle-omgeving. PL/SQL is een superset van SQL, waarbij een aantal procedurele mogelijkheden zijn toegevoegd. Deze taal wordt weliswaar in het geheel niet in dit boek behandeld, maar het is toch aardig om hiervan minstens één toepassing te bekijken. We kunnen bijvoorbeeld een functie definiëren die het

[aantal medewerkers van een gegeven afdeling berekent \(zie figuur\)](#)

## 5.30).

155

```
SQL> create or replace function m_aantal(afd_nr in number)
  2  return number is
  3      m_count number(2) := 0;
  4  begin
  5      select count(*) into m_count
  6      from medewerkers m
  7      where m.afd = afd_nr;
  8      return (m_count);
  9  end;
10 /
```

Functie is aangemaakt.

```
SQL>
```

```
SQL> select anr, naam, locatie
  2 ,      m_aantal(anr)
  3 from afdelingen;
```

ANR	NAAM	LOCATIE	M_AANTAL (ANR)
10	HOOFDKANTOOR	LEIDEN	3
20	OPLEIDINGEN	DE MEERN	5
30	VERKOOP	UTRECHT	6
40	PERSONEELSZAKEN	GRONINGEN	0

```
SQL>
```

## Figuur 5.30

We kunnen nu vrij simpel een overzicht krijgen van alle afdelingen, met het aantal medewerkers. Dit zou een lastige opgave zijn zonder deze functie; met name afdeling 40 (zonder medewerkers) komt niet vanzelfsprekend tevoorschijn. Daarvoor is een zogenaamde OUTER JOIN

nodig ([zie hoofdstuk 8](#), [of het](#) gebruik van een subquery in de SELECT-component ([zie hoofdstuk 9](#)).

## **Figuur 5.31**

Let op wat het SQL\*Plus-commando DESCRIBE met dit soort functies doet:

156

```
SQL> describe m_aantal
FUNCTION m_aantal RETURNS NUMBER
Argumentnaam          Type           In/Out Standaard?
-----  -----  -----
AFD_NR                NUMBER         IN
SQL>
```

## **Figuur 5.32**

5.9

Opgaven

1

Geef van alle medewerkers eerst de achternaam, dan een komma, gevolgd door de voorletter(s) en voorvoegsels.

2

Geef van alle medewerkers hun naam en de geboortedatum, in het formaat zoals bijvoorbeeld ‘11 april 1997’.

3

Op welke dag ben (of was!) je precies 10.000 dagen oud? Op welke dag van de week valt/viel dat?

4

Herschrijf het voorbeeld [in figuur 5.24](#) met gebruikmaking van de NVL2-

functie.

5

Herschrijf het voorbeeld [in figuur 5.25](#) met gebruikmaking van CASE-expressies, zowel in de SELECT-component als in de ORDER BY-component.

6

Herschrijf het voorbeeld [in figuur 5.21](#) met gebruikmaking van DATE- en INTERVAL-constantes, zodat ze onafhankelijk worden van de NLS\_DATE\_FORMAT-instelling.

7

Onderzoek het verschil tussen de datumformaten WW en IW (weeknummer en ISO-weeknummer) aan de hand van een willekeurige datum, en verklaar het eventuele verschil.

157

- 
- 

## **Hoofdstuk 6**

### **Datamanipulatie**

In dit hoofdstuk gaan we de inhoud van de database veranderen.

De commando's waarmee dat in SQL kan worden gedaan, vormen tezamen het DML-gedeelte (Data Manipulation Language) van SQL.

In de eerste drie paragrafen komen de belangrijkste commando's van deze

categorie aan de orde: INSERT, UPDATE, DELETE.

In een productieomgeving wordt datamanipulatie vaak gepleegd via applicaties, vooral als het om grote hoeveelheden gegevens gaat. Dit soort applicaties wordt over het algemeen gebouwd met behulp van applicatie-frameworks, zoals .Net en diverse Java-frameworks of tools van Oracle zelf (Application Express, ADF, Forms).

Toch is het soms wel eens handig om datamanipulatie in

SQL\*Plus uit te voeren, bijvoorbeeld als het gaat om globale updates (een bepaalde kolom wijzigen voor een groot aantal rijen tegelijk), het leegmaken van tabellen of om een commando even snel uit te proberen.

In paragraaf 6.4 komen het transactiemechanisme en de bijbehorende SQL-commando's aan bod: COMMIT, SAVEPOINT,

ROLLBACK.

Dit hoofdstuk is ten slotte de meest voor de hand liggende plaats om aandacht te besteden aan de termen 'read consistency' en

'locking'. Dit zal geen technische uiteenzetting worden, maar vooral een verhaal om deze begrippen aannemelijk te maken.

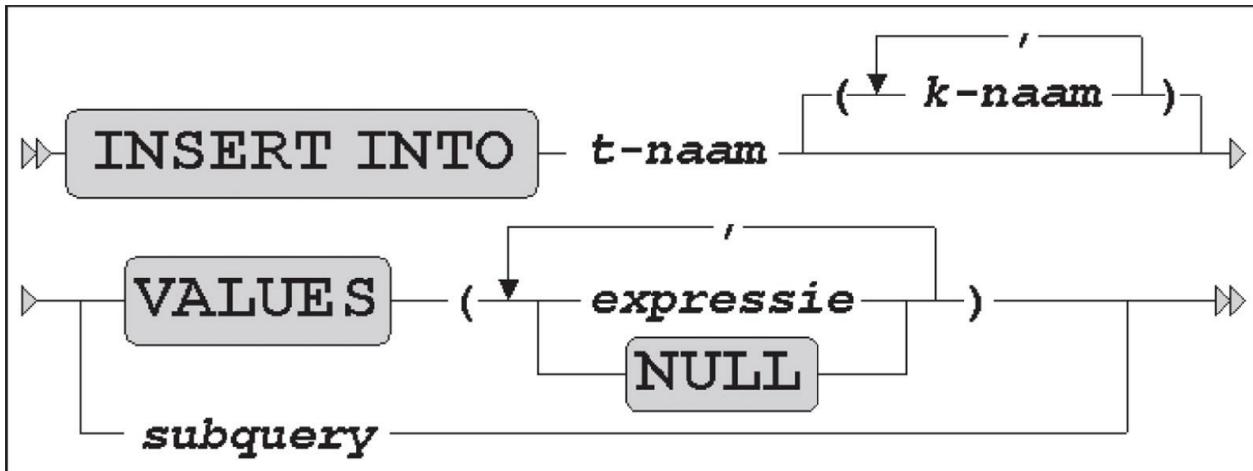
## 6.1

### Het INSERT-commando

Met behulp van het INSERT-commando kunnen rijen aan een tabel worden toegevoegd. Dat kan in principe op twee manieren:

Via de values-component, door een verzameling kolomwaarden op te geven.

Met behulp van een subquery, gebruikmakend van bestaande



gegevens.

Met de eerste methode kan per uitvoering van het INSERT-commando slechts één rij aan een tabel worden toegevoegd. Met de tweede methode kunnen zoveel rijen als door de subquery worden

gereturneerd, in één uitvoering aan de tabel worden toegevoegd. Beide mogelijkheden blijken uit het syntax-diagram [van figuur 6.1](#).

### Figuur 6.1

Als men de fysieke kolomvolgorde van een tabel kent (zoals het DESCRIBE-commando ze weergeeft) hoeven de kolomnamen niet te worden vermeld. Het is dan noodzakelijk dat er precies genoeg waarden

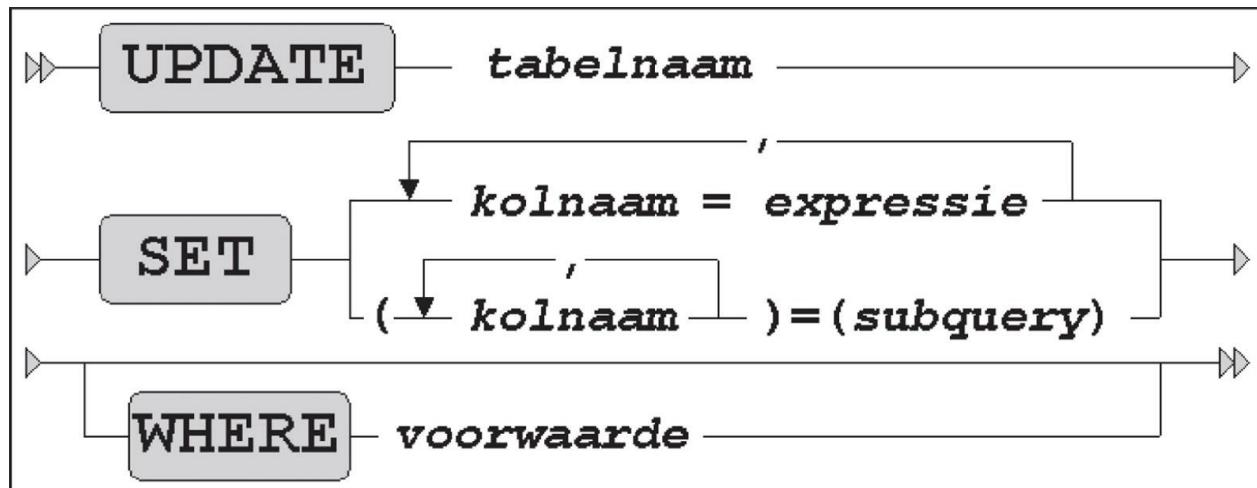
– in de juiste volgorde – worden verstrekt. Vaak wordt in de VALUES-component een rijtje constanten gebruikt, maar in principe zijn expressies toegestaan. Met behulp van het gereserveerde woord NULL

kan een kolom van een null-waarde worden voorzien.

De tweede methode vult een tabel met behulp van een subquery. Aan deze query worden geen eisen gesteld, anders dan dat de kolommen die de query selecteert moeten overeenkomen met de kolommen van de tabel waarin geinsert wordt. Het is zelfs mogelijk dat het een query is op de tabel die wordt gevuld, hoewel dat weinig zal gebeuren. Het is wel de snelste manier om een tabel vol te laten lopen, mits er geen sprake is van unieke sleutelconstraints.

Voorbeelden van INSERT-commando's volgen na de behandeling van de andere twee datamanipulatie-commando's van SQL: UPDATE en DELETE.

159



6.2

## Het UPDATE-commando

Met het UPDATE-commando kunnen kolomwaarden van bestaande rijen in een tabel worden gewijzigd. Het commando heeft drie componenten, met de volgende functie:

**UPDATE** ... Welke tabel wil ik wijzigen.

**SET** ... Om welke wijziging gaat het.

**WHERE** ... Op welke rijen wil ik die toepassen.

[Deze drie componenten zien we terug in het syntaxdiagram van figuur 6.2.](#)

## Figuur 6.2

Als de WHERE-component wordt weggelaten zal de wijziging op alle rijen van de tabel worden uitgevoerd. Dit illustreert het feit dat het UPDATE-

commando in principe op tabelniveau werkt en niet op rijniveau; de WHERE-component werkt ook hier als de relationele restrictie-operator.

De SET-component biedt de mogelijkheid om meerdere wijzigingen tegelijkertijd uit te voeren, waarbij de wijzigingen onderling door komma's worden gescheiden.

Een wijziging kan één kolom betreffen, of een combinatie van kolommen. Deze mogelijkheid is vooral aantrekkelijk in combinatie met een subquery.

160



6.3

### Het DELETE-commando

Het eenvoudigste datamanipulatie-commando is DELETE. Ook dit commando werkt op tabelniveau; met de WHERE-component wordt de restrictie tot bepaalde rijen van de tabel gerealiseerd. Als men de WHERE-component achterwege laat, zal het resultaat van het DELETE-commando een lege tabel zijn.

[Het syntax-diagram van het commando DELETE ziet eruit zoals in figuur 6.3.](#)

### Figuur 6.3

Let op het verschil tussen de volgende twee commando's:

SQL> **drop table afdelingen;**

SQL> **delete from afdelingen;**

Het commando DROP TABLE verwijdert niet alleen de inhoud van de tabel maar óók de tabel zelf, en daarmee alle samenhangende structuren zoals indexen en privileges.

Het is een datadefinitie- (DDL) -commando. Het commando DELETE verandert niets aan de structuur van de database, maar alleen aan de inhoud; het is een datamanipulatie-(DML) -commando.

Bovendien kan het commando DROP TABLE niet met een ROLLBACK worden teruggedraaid, terwijl dat met een DELETE-commando wel mogelijk is. Het ROLLBACK-commando komt in de volgende paragraaf aan de orde.

Verwijderen van rijen lijkt eenvoudig, maar toch kunnen er complicaties optreden als gevolg van constraints. Voor UPDATE- en INSERT-commando's geldt overigens hetzelfde. In het volgende hoofdstuk over 161

```
SQL> insert into afdelingen
  2  values (90,'BEDRIJFSRESTAURANT','NAARDEN', NULL);

1 row created.

SQL> insert into afdelingen(naam,locatie,anr)
  2  values('CONSULTANCY','TIEL', 10);

insert into afdelingen(naam,locatie,anr)
*
ERROR at line 1:
ORA-00001: unique constraint (BOEK.A_PK) violated

SQL>
```

datadefinitie komen we hierop terug.

We gaan nu enige INSERT-, UPDATE- en DELETE-commando's op onze database uitproberen. Door zo nu en dan het commando ROLLBACK uit te voeren herstellen we steeds de beginsituatie.

[In figuur 6.4 zien](#) we eerst hoe we zonder specificatie van kolomnamen een rij aan de AFDELINGEN-tabel kunnen toevoegen, en zien we ook een toepassing van het gereserveerde woord NULL; daarna zien we dat de primaire sleutel-constraint wordt geschonden (we hebben namelijk al een afdeling 10).

## Figuur 6.4

[In figuur 6.5 zien](#) we hoe een salarisschaal wordt verwijderd, en hoe we met behulp van een subquery vier nieuwe schalen kunnen toevoegen.

Vervolgens draaien we deze wijzigingen terug met behulp van het commando ROLLBACK.

162

```
SQL> delete from schalen where snr=5;
1 row deleted.

SQL> insert into schalen
  2 select snr+4,ondergrens+2300,least(bovengrens+2300,9999), 500
  3 from   schalen;

4 rows created.

SQL> rollback;

Rollback complete.

SQL>
```

```
SQL> update medewerkers
  2  set      functie  ='VERKOPER'
  3  '      maandsal = maandsal - 500
  4  '      comm     = 0
  5  '      afd      = 30
  6 where mnr      = 7876;

1 row updated.

SQL> rollback;

Rollback complete.

SQL>
```

## Figuur 6.5

In figuur 6.6 wijzigen we met één UPDATE-commando tegelijkertijd een aantal attributen van medewerker 7876. Hierbij wordt het maandsalaris met 500 verminderd.

In figuur 6.7 zien we ten slotte hoe we in de WHERE-component van het commando UPDATE gebruik kunnen maken van een subquery die werkt op basis van een combinatie van twee kolomwaarden (cursus, begindatum).

## Figuur 6.6

```
SQL> update inschrijvingen
  2  set evaluatie = 1
  3  where (cursus,begindatum)
  4    in (select cursus,begindatum
  5         from uitvoeringen
  6         where locatie = 'UTRECHT');

3 rows updated.

SQL> rollback;

Rollback complete.

SQL>
```

- 
- 
- 

## Figuur 6.7

Nu we de drie standaard DML-commando's van SQL hebben besproken (INSERT, UPDATE en DELETE) mag een aantal gerelateerde commando's niet onvermeld blijven:

Het MERGE-commando (informeel ook wel bekend als 'UPSERT') is het commando dat inserts en/of updates kan uitvoeren, al naar gelang bepaalde rijen al bestaan. Dit commando is met name bedoeld voor laadprocessen van data warehouse-omgevingen: van nieuwe rijen die geladen moeten worden kan eerst gechecked worden of ze al aanwezig zijn in het data warehouse, zo ja dan zal een update uitgevoerd worden, zo nee dan zal een insert uitgevoerd worden. We behandelen het MERGE-commando verder niet in dit boek.

Het INSERT-ALL commando (ook wel bekend als 'Multi Table

Insert') is een variant op het INSERT commando. INSERT ALL is in staat om in meer dan één tabel tegelijk te inserten. Dit commando is ook met name bedoeld voor (laadprocessen van) data warehouse-omgevingen en wordt verder niet behandeld in dit boek.

Het TRUNCATE-commando stelt ons in staat om alle rijen van een tabel tegelijk te verwijderen, op een efficiëntere manier dan met het DELETE-commando. Het TRUNCATE-commando behoort echter tot de DDL-commando's; vandaar dat TRUNCATE niet hier, maar in het volgende hoofdstuk wordt behandeld.

## 6.4

### Transactieverwerking

Wijzigingen in de inhoud van de database krijgen in eerste instantie een voorlopig karakter. Dat houdt onder meer in dat we in onze sessie met Oracle de resultaten van DML-commando's die wij uitgevoerd hebben, zien. Maar dat andere databasegebruikers (in hun sessie met Oracle) bij raadpleging deze resultaten nog niet zien, en de oude situatie voorgesloten krijgen.

Bovendien zullen ze rijen die wij gedeleteert of geupdateert hebben, niet mogen wijzigen zolang wij onze wijzigingen nog niet definitief hebben gemaakt (met COMMIT, zie hierna), dan wel hebben geannuleerd met ROLLBACK, zoals in de voorgaande voorbeelden.

Het commando om wijzigingen in de database definitief te maken is COMMIT. Zodoende kan men steeds een aantal mutaties verrichten, ze bevestigen met een COMMIT, weer enkele mutaties verrichten, enzovoort.

Wat COMMIT en ROLLBACK feitelijk doen, is het afsluiten van een lopende *transactie*. De start van een transactie vindt impliciet plaats door uitvoering van het eerste DML-commando.

Met de term transactie bedoelen we een aantal logisch bij elkaar horende mutaties. We willen dat een transactie als geheel slaagt of faalt, in geen geval gedeeltelijk.

In het betalingsverkeer bestaat een transactie bijvoorbeeld uit twee mutaties: een afschrijving van rekening A en een bijschrijving op rekening B. Dan ligt het voor de hand om per af- en bijschrijving een commit te geven. Stel namelijk dat er halverwege iets misgaat – terwijl de ene mutatie al is doorgevoerd en de andere nog niet – dan zou de administratie bepaald niet

meer in orde zijn. Bovendien, een raadpleging van een andere databasegebruiker zou er precies tussendoor kunnen lopen, waardoor een financiële rapportage er vreemd uit zou kunnen zien.

Als we te lang nalaten onze mutaties te ‘committen’, lopen we het risico dat bij een systeemstoring al ons werk verloren gaat. Om de consistentie van de database te waarborgen zal Oracle namelijk, na herstel van de storing, een ROLLBACK uitvoeren over alle transacties die nog niet waren gecommit. Vervelend, maar noodzakelijk. Dit illustreert het feit dat niet alleen wij als gebruikers COMMIT en ROLLBACK *explicit* kunnen hanteren, maar dat Oracle dat zelf ook *implicit* kan doen.

Een tweede gevolg van het niet committen is dat we andere gebruikers die dezelfde gegevens ook willen muteren, blokkeren; hoe dat komt wordt in de volgende paragraaf uitgelegd.

165

Alle DDL-commando’s (zoals CREATE, ALTER, DROP, GRANT, REVOKE) hebben een *impliciete* COMMIT tot gevolg. Anders gezegd: elk DDL-commando vormt een transactie op zichzelf, bestaande uit één commando, dat onmiddellijk wordt gecommit.

Diverse tools, waaronder SQL\*Plus, kennen een AUTOCOMMIT-optie, waarmee het mogelijk is om ook DML-commando’s onmiddellijk te laten committen. Alleen onder bijzondere omstandigheden is het verstandig om van deze mogelijkheid gebruik te maken; het gevolg is namelijk dat er geen ROLLBACK meer mogelijk is na vergissingen, en dat het transactieprincipe (sommige mutaties horen logischerwijs bij elkaar) verloren gaat.

In SQL\*Plus kan de AUTOCOMMIT-optie met het SQL\*Plus-commando SET worden geregeld. Het is ook mogelijk om na elke ‘x’ (zelf te bepalen aantal) DML-commando’s door SQL\*Plus een impliciete commit te laten geven:

**SQL> set autocommit on**

**SQL> show autocommit**

autocommit IMMEDIATE

SQL> **set autocommit 42**

SQL> **show autocommit**

AUTOCOMMIT ON for every 42 DML statements

SQL> **set autocommit off**

SQL>

In de loop van een transactie kunnen we *savepoints* definiëren. Dit zijn punten in een transactie waarop we kunnen terugvallen zónder de transactie zelf af te sluiten. Stel dat we bijvoorbeeld vier mutaties hebben verricht, waarna we de laatste twee mutaties ongedaan willen maken. Zie [figuur 6.8](#).

```

SQL> delete from historie      where mnr=7654;
2 rows deleted.

SQL> delete from medewerkers  where mnr=7654;
1 row deleted.

SQL> savepoint EEN;
Savepoint created.

SQL> delete from uitvoeringen where cursus='ERM';
1 row deleted.

SQL> delete from cursussen    where code='ERM';
1 row deleted.

SQL> rollback to savepoint EEN;
Rollback complete.

SQL> select omschrijving from cursussen where code='ERM';

OMSCHRIJVING
-----
Datamodellering met ERM

SQL> rollback;
Rollback complete.

SQL>

```

## Figuur 6.8

6.5

### Read consistency en locking

Het komt vaak voor dat databasegegevens door vele gebruikers tegelijkertijd worden benaderd. Dit verschijnsel wordt ook wel *concurrency* genoemd. Het DBMS moet ervoor zorgen dat die benadering correct verloopt. De meest rigoureuze methode die een DBMS daarbij kan toepassen is: alle transacties van gebruikers één voor één afhandelen, en de bijbehorende gegevens exclusief blokkeren tot de transacties afgesloten zijn. Deze sequentiële handelwijze leidt echter vrijwel zeker tot nodeloos lange wachttijden.

Gegevens blokkeren gebeurt over het algemeen met behulp van *locking*.

Locking is bijvoorbeeld noodzakelijk om te voorkomen dat iemand gegevens wijzigt waarop nog mutaties van andere gebruikers uitstaan.

Deze paragraaf vertelt iets over de manier waarop Oracle concurrency afhandelt. Om te beginnen maken we onderscheid tussen twee soorten databasegebruikers:

### **Lezers**

(select)

### **Schrijvers**

(insert, update, delete)

In principe wordt door Oracle *géén* locking toegepast bij raadpleging.

Dat betekent dat *lezers* nooit last van elkaar hebben. Dat betekent tevens dat *schrijvers* nooit hoeven te wachten op *lezers*, en andersom.

Het enige geval waarin wachtsituaties kunnen optreden, is als meerdere gebruikers tegelijkertijd proberen dezelfde rijen te muteren. Elke mutatie leidt namelijk automatisch tot een *lock op rijniveau*, dat pas wordt vrijgegeven bij het afsluiten van de transactie. Op deze manier worden onderlinge wachttijden tot een minimum gereduceerd.

In een databaseomgeving is *read consistency* een belangrijk begrip. Het is de basis van een garantie op correcte resultaten bij de benadering van gegevens. Oracle moet ervoor zorgen dat elk SQL-commando, ongeacht hoe lang het loopt en wat er intussen allemaal in de database gebeurt, een *snapshot* van de benodigde gegevens tot zijn beschikking houdt.

Een snapshot is te beschouwen als een momentopname van een voortdurend veranderende database-inhoud.

Het kan voorkomen dat gegevens (door een andere gebruiker) zijn gewijzigd en met een commit zijn bekraftigd, terwijl een langlopende query nog steeds bezig is. Oracle moet dan de situatie reconstrueren, zoals die bestond toen de

query begon. De manier waarop Oracle dat realiseert wordt in dit boek niet besproken.

Zelfs als er géén sprake is van andere databasegebruikers is read consistency toch nog belangrijk. Stel dat we bijvoorbeeld alle medewerkers een salarisverdubbeling willen geven als hun huidige salaris onder het gemiddelde salaris ligt. Dit kunnen we doen met het volgende update-commando.

168

```
SQL> update medewerkers
  2  set maandsal = 2 * maandsal
  3  where maandsal < (select avg(maandsal)
  4                      from medewerkers);

8 rows updated.

SQL>
```

### Figuur 6.9

Dan zou je als medewerker haast hopen dat je zo laat mogelijk aan de beurt komt, zodat de reeds toegekende salarisverhogingen het gemiddelde inmiddels zódanig hebben beïnvloed dat je ook voor een verhoging is aanmerking komt.

In een Oracle-database is dat ijdele hoop, omdat het read consistency-principe ervoor zal zorgen dat de subquery in het update-statement om het gemiddelde salaris te bepalen steeds dezelfde waarde oplevert, hoe vaak de subquery ook wordt uitgevoerd.

Belangrijk is wel dat we ons realiseren dat read consistency op commandoniveau is geïmplementeerd. Willen we read consistency op transactieniveau – bijvoorbeeld ten behoeve van een rapportage die bestaat uit meerdere queries – dan moeten we maatregelen nemen om te kunnen garanderen dat de rapportage consistent is.

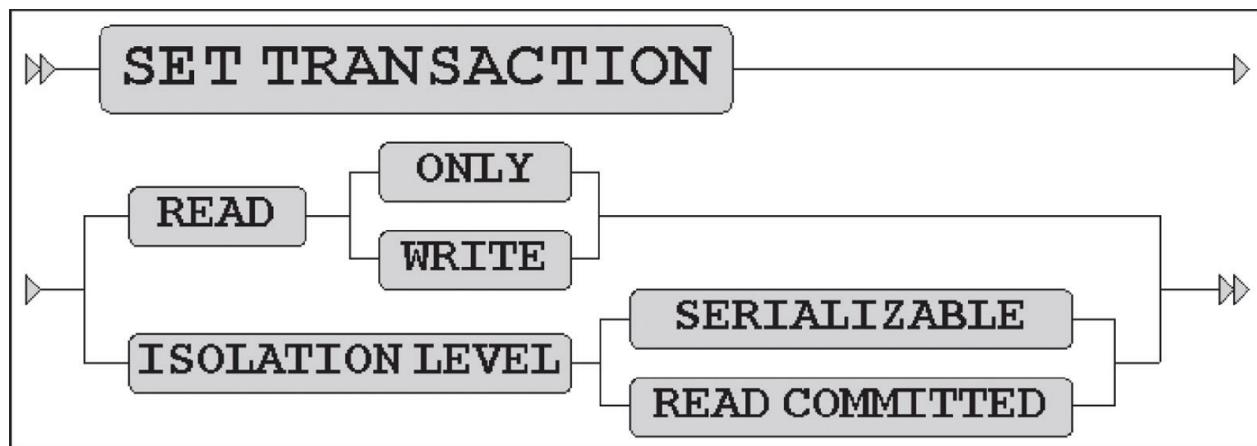
We kunnen in zo'n situatie besluiten om handmatig locking toe te passen, met de daartoe beschikbare SQL-commando's. Dan sluiten we gelijktijdige

transacties feitelijk uit, zodat eventuele andere transacties zullen moeten wachten.

Er is ook een andere oplossing, die dat nadeel niet heeft omdat er geen locking aan te pas komt: het SQL-commando SET TRANSACTION.

Hiermee kunnen we zelf het ‘isolation level’ van onze transactie aangeven (volgens de ANSI/ISO SQL-standaard). We volstaan hier met de (gedeeltelijke) syntax van dit commando (zie [figuur 6.10](#)).

169



**Figuur 6.10**

Omdat Oracle hierbij geen locking of andere blokkeringstechniek toepast, is er wel een risico dat op een gegeven moment de oude situatie niet meer te reconstrueren is, zeker als de read-only-transactie erg lang duurt terwijl er intussen allerlei mutaties worden uitgevoerd. Hier gaan we niet verder op in.

170

## **Hoofdstuk 7**

### **Datadefinitie – deel II**

In [hoofdstuk 3](#) hebben we net genoeg van het onderwerp datadefinitie (DDL) behandeld om de casustabellen te kunnen creëren. In dit hoofdstuk gaan we dieper op enkele aspecten van datadefinitie in.

Ook in dit hoofdstuk zullen we verre van compleet zijn; om een voorbeeld te geven, het create table-commando alleen al beslaat meer dan 100 (honderd) pagina's in de Oracle-documentatie...

We zullen allereerst het create table-commando en de Oracle-datatypes uitvoeriger bespreken. Vervolgens introduceren we het alter table-commando, waarmee de structuur van een bestaande tabel kan worden gewijzigd; het declareren van en omgaan met constraints komen daarna aan de orde.

In [paragraaf 7.5](#) behandelen we indexen. Indexen dienen in principe ter verbetering van query-responslijden. Op deze

paragraaf volgt dan ook een paragraaf over performance.

Uitgebreide behandeling past niet in de context van dit boek, maar het is goed om ook hier iets van te weten.

De meest efficiënte manier om in Oracle volgnummers te

[genereren is het gebruik van sequences; ze worden in paragraaf](#)

[7.7 geïntroduceerd. Dan komen synoniemen aan de orde, in](#)

[paragraaf 7.8. Door](#) synoniemen te creëren kunnen we bijvoorbeeld met afkortingen van tabellenamen werken, of de naam van de eigenaar van een tabel verbergen.

In [paragraaf 7.9](#) komt het commando DROP TABLE aan de orde.

Hier zullen we ook het concept van een RECYCLE BIN behandelen.

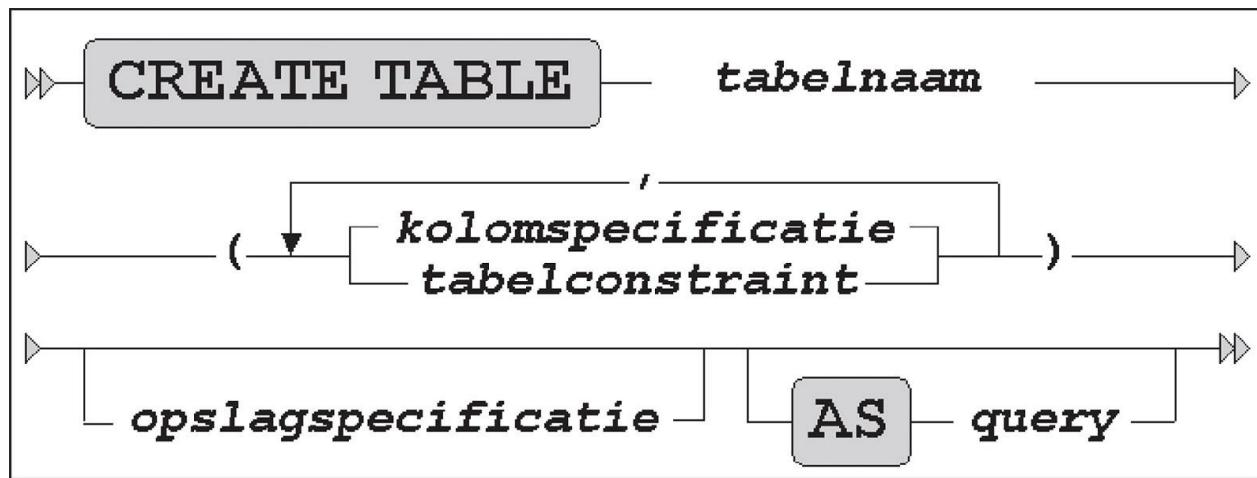
In de laatste paragraaf behandelen we ten slotte nog enkele met tabeldefinitie samenhangende commando's.

Het hoofdstuk sluiten we af met opgaven.

## 7.1

### CREATE TABLE

In een eerder hoofdstuk hebben we het CREATE TABLE-commando al besproken. In deze paragraaf zullen we er uitvoeriger op ingaan. Een 171



vollediger syntaxdiagram ziet eruit als [in figuur 7.1](#).

### Figuur 7.1

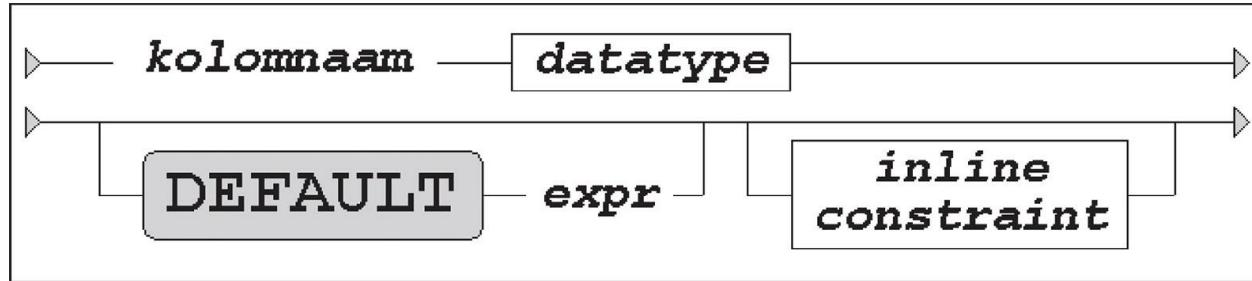
Uit deze figuur blijkt dat een CREATE TABLE-commando twee soorten componenten kent: kolomspecificaties en tabelconstraint-definities. Op constraints komen we nog terug in een van de volgende paragrafen.

Verder kunnen we in een STORAGE-component (*opslagspecificatie* in [figuur 7.1](#)) allerlei aanwijzingen opnemen omtrent de gewenste fysieke opslag van de gegevens. Dit is een belangrijk middel om de fysieke opslag van de gegevens op schijf te optimaliseren; bespreking van deze mogelijkheden valt echter buiten het bestek van dit boek.

Tabellen kunnen ook worden gecreëerd met behulp van een subquery in de AS-component. Deze constructie is te vergelijken met een van de mogelijkheden van het INSERT-commando, dat we in het vorige hoofdstuk hebben behandeld. Daarbij werden rijen aan een bestaande tabel toegevoegd via een query. Het verschil is dat de tabel nu ook nog wordt gecreëerd. In dit geval kunnen de kolomspecificaties achterwege gelaten worden (deze worden dan afgeleid van het resultaat van de subquery) en er mogen in ieder geval geen datatypes worden gespecificeerd. Ook de datatypes worden in dat geval namelijk bepaald door het resultaat van de subquery.

Een kolomspecificatie ziet er in het algemeen uit zoals in figuur 7.2.

172



## Figuur 7.2

Hieruit blijkt dat we constraints niet alleen als zelfstandige *tabelconstraint* (ook wel *out of line constraint* genoemd) in een CREATE

TABLE-commando kunnen opnemen ([zie figuur 7.1](#)) maar ook als *inline constraint* in een kolomspecificatie.

Met de DEFAULT-optie kan een waarde worden gespecificeerd die wordt gebruikt als een INSERT-commando geen expliciete waarde voor de betreffende kolom bevat.

7.2

# Datatypes

We hebben het al eerder over datatypes gehad. Een uitgebreider overzicht van de belangrijkste Oracle datatypes ziet er als volgt uit: **CHAR( n )**

## Karakterstring met vaste lengte $n$

**VARCHAR[2](*n*)**

## Variabele string, maximaal $n$ karakters

**DATE**

Datum (4712 BC t/m 9999 AD)

## **TIMESTAMP**

Tijdstip, met of zonder tijdzone-informatie

## **INTERVAL**

Tijdsinterval

## **BLOB**

Ongestructureerde binaire gegevens

## **CLOB**

Grote stukken tekst

## **RAW( *n* )**

Binaire gegevens; maximaal *n* bytes

## **NUMBER**

Drijvende-kommagetal met een maximale  
precisie van 38 cijfers

## **NUMBER( *n* )**

Geheel getal van maximaal *n* cijfers

## **NUMBER( *n, m* )**

Precisie van *n* cijfers, waarvan *m* achter de decimale punt

## **BINARY\_FLOAT**

32-bits drijvende komma getal

## **BINARY\_DOUBLE**

64-bits drijvende-komma-getal

173

■

■

Ten behoeve van de uitwisselbaarheid met andere DBMS-sen

ondersteunt Oracle diverse datatype-synoniemen. Zo is CHARACTER

hetzelfde als CHAR, DECIMAL(n,m) hetzelfde als NUMBER(n,m) en  
NUMBER

heeft zelfs diverse synoniemen, zoals INTEGER, REAL en SMALLINT.

Ieder datatype heeft zijn eigen maximale breedte of precisie: **Datatype:**

**Limiet:**

**NUMBER**

38 cijfers

**CHAR**

2000

**VARCHAR[2]**

4000

**BLOB**

8 TB

**CLOB**

8 TB

De limiet van 8 TB voor de LOB-datatypes is niet helemaal correct; het feitelijke maximum kan nog veel hoger zijn, en hangt af van een aantal Oracle-configuratie-parameters.

Sinds Oracle7 betekenen VARCHAR en VARCHAR2 hetzelfde. Oracle adviseert echter om zoveel mogelijk gebruik te maken van VARCHAR2, omdat in de toekomst een verschil in behandeling zou kunnen ontstaan tussen de datatypes VARCHAR en VARCHAR2, en wel in de manier waarop strings van ongelijke lengte met elkaar worden vergeleken. Daar zijn namelijk twee manieren voor:

**Padded** (aangevuld).

**Non-padded** (niet-aangevuld).

In het eerste geval wordt een eventueel kortere string eerst aangevuld met spaties tot de lengte van de langste string, waarna vergelijking plaatsvindt. Dat betekent dat spaties aan het einde van een string geen invloed hebben op het resultaat van een vergelijking. Bijvoorbeeld: **Padded vergelijking**:

**Non-padded vergelijking:**

‘aap’ = ‘aap ’

‘aap’ < ‘aap ’

Als we nu het datatype VARCHAR2 in plaats van VARCHAR gebruiken, zijn we verzekerd van *non-padded* vergelijking, ongeacht de toekomstige ontwikkeling van het VARCHAR-datatype volgens de ISO/ANSI-standaard.

```
<SQL> select 5.1d, 42f from dual;  
  
      5.1D      42F  
----- -----  
 5.1E+000  4.2E+001  
  
SQL>
```

Het verschil tussen RAW en VARCHAR2 is dat RAW-gegevens nooit zullen worden geïnterpreteerd, terwijl VARCHAR2-gegevens bijvoorbeeld bij transport van een *ascii*- naar een *utf*-omgeving automatisch zullen worden geconverteerd.

Aan het eind van deze paragraaf komen we nog even terug op getallen.

Oracle heeft in het verleden getallen altijd opgeslagen in een eigen intern formaat, om daarmee maximale *portabiliteit* (overdraagbaarheid) naar allerlei verschillende platforms (besturingssystemen) te bereiken.

Oracle doet dat nog steeds, als we gebruikmaken van het NUMBER-datatype, of van een van de eerder genoemde synoniemen voor dat datatype.

Sinds Oracle 10g worden ook drijvendekomma-(floating point)getallen ondersteund, waarbij niet meer gebruikgemaakt wordt van de interne opslag van het NUMBER-datatype. Drijvendekomma getallen zijn niet zo precies als NUMBER-getallen, maar ze leveren veel betere responstijden op bij (intensieve) berekeningen. Er zijn twee nieuwe datatypes in deze categorie:

BINARY\_FLOAT: 32-bit, enkele precisie.

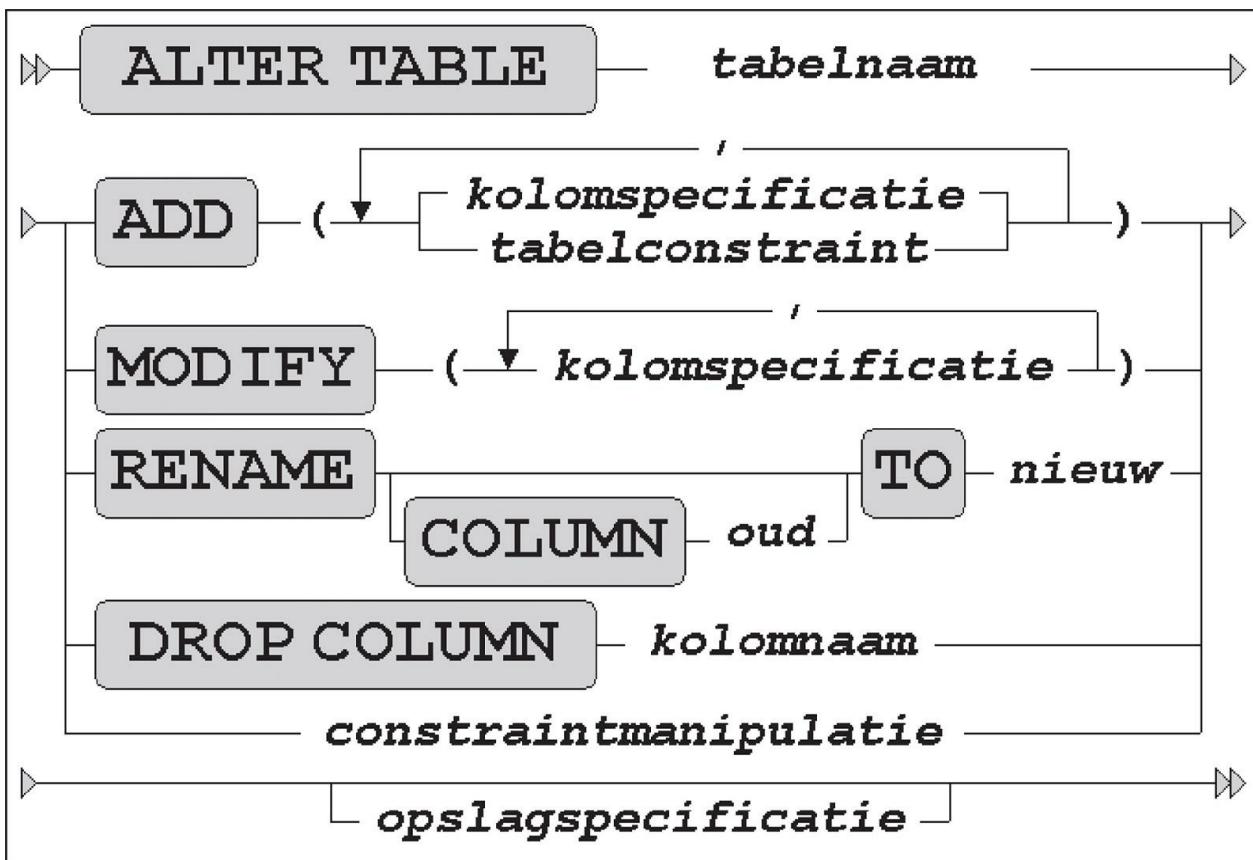
BINARY\_DOUBLE: 64-bit, dubbele precisie.

Constanten van deze datatypes kunnen worden aangegeven in een string met een F of een D aan het eind:

We zullen deze twee drijvendekomma datatypes in dit boek verder niet gebruiken; zie de Oracle-documentatie voor meer details.

## ALTER TABLE

Soms blijkt het noodzakelijk om de structuur van een bestaande tabel aan te passen. Een kolom is bijvoorbeeld bij nader inzien iets te smal gedefinieerd, we willen aan de tabel een extra kolom toevoegen, of we 175



willen een wijziging aanbrengen in de constraints. Daartoe bestaat het ALTER TABLE-commando, waarvan het syntaxdiagram eruitziet als in

[figuur 7.3.](#)

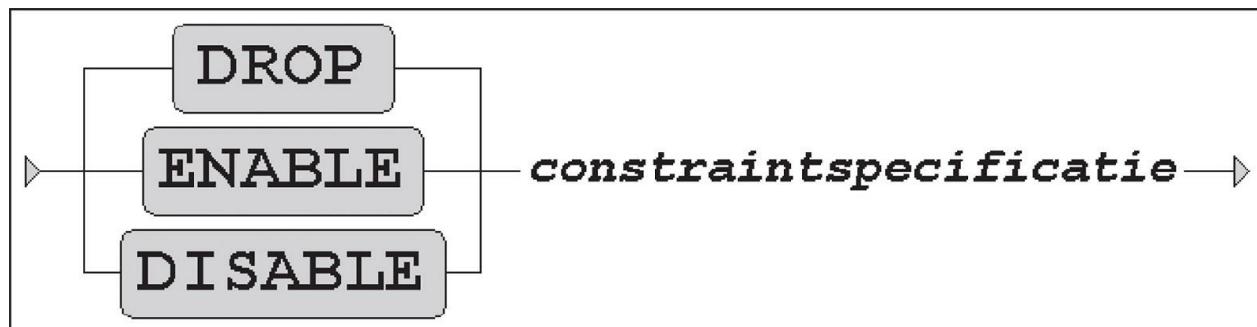
### Figuur 7.3

**Opmerking:** Het ALTER TABLE-commando is veel complexer en uitgebreider dan [in figuur 7.3](#) is weergegeven; zie de Oracle-documentatie voor meer details.

Met de ADD-optie kunnen we kolommen aan de tabel toevoegen, of nieuwe tabel-constraints definiëren. Met de MODIFY-optie kan de definitie van bestaande kolommen worden gewijzigd. Een kolom kan bijvoorbeeld breder worden gemaakt, of hij kan NULL of NOT NULL

worden gemaakt.

We kunnen met behulp van het DROP COLUMN-optie kolommen van tabellen verwijderen. Dit moet met de nodige voorzichtigheid gebeuren; misschien zijn bepaalde toepassingsprogramma's van een kolom afhankelijk, die door het verwijderen van de kolom niet meer functioneren. We kunnen de kolommen ook onbruikbaar maken met 176



ALTER TABLE ... SET UNUSED, en op een later tijdstip met ALTER TABLE

... DROP UNUSED COLUMNS fysiek uit de database verwijderen. We gaan hier verder niet op in.

De RENAME COLUMN-optie stelt ons in staat om de naam van een kolom veranderen.

Met de optie voor *constraint-manipulatie* kunnen we constraints verwijderen, aan- of uitzetten. Figuur 7. 4 geeft een detallering van deze optie.

In de volgende paragraaf behandelen we constraints verder.

#### Figuur 7.4

Net als bij CREATE TABLE kan ook in het ALTER TABLE-commando invloed worden uitgeoefend op de fysieke opslag van de tabel; bespreking valt zoals al eerder gezegd buiten het bestek van dit boek.

## 7.4

### Constraints

In de commando's CREATE TABLE en ALTER TABLE kunnen constraint-definities worden opgenomen. We onderscheiden de volgende soorten constraints:

#### *Not-null-constraints*

Deze geven aan dat een kolom voor elke rij in de tabel een waarde dient te bevatten:

NULL's worden niet toegestaan.

#### *Check-constraints*

177

Met deze constraints kunnen we aangeven dat een eenvoudige voorwaarde voor elke rij in de tabel dient te gelden. Met 'eenvoudig'

wordt bedoelt dat de CHECK-constraint een (boolean) expressie dient te bevatten waarin alleen referenties naar in de tabel beschikbare kolommen voorkomen, én dat deze expressie geen subqueries mag bevatten.

#### *Sleutel-constraints*

Met deze constraints kunnen we aangeven dat een waarde in een kolom (of een combinatie van waarden in een combinatie van kolommen) niet in meer dan één rij van de tabel mag voorkomen. Elke rij dient dus (een) unieke waarde(n) te hebben op deze kolom(men). De ANSI/ISO-standaard onderscheidt twee soorten sleutel-constraints: PRIMARY KEY en UNIQUE. Het belangrijke verschil tussen deze twee soorten 'keys' is dat bij UNIQUE, NULL's toegestaan worden.

#### *Vreemde-sleutel-constraints*

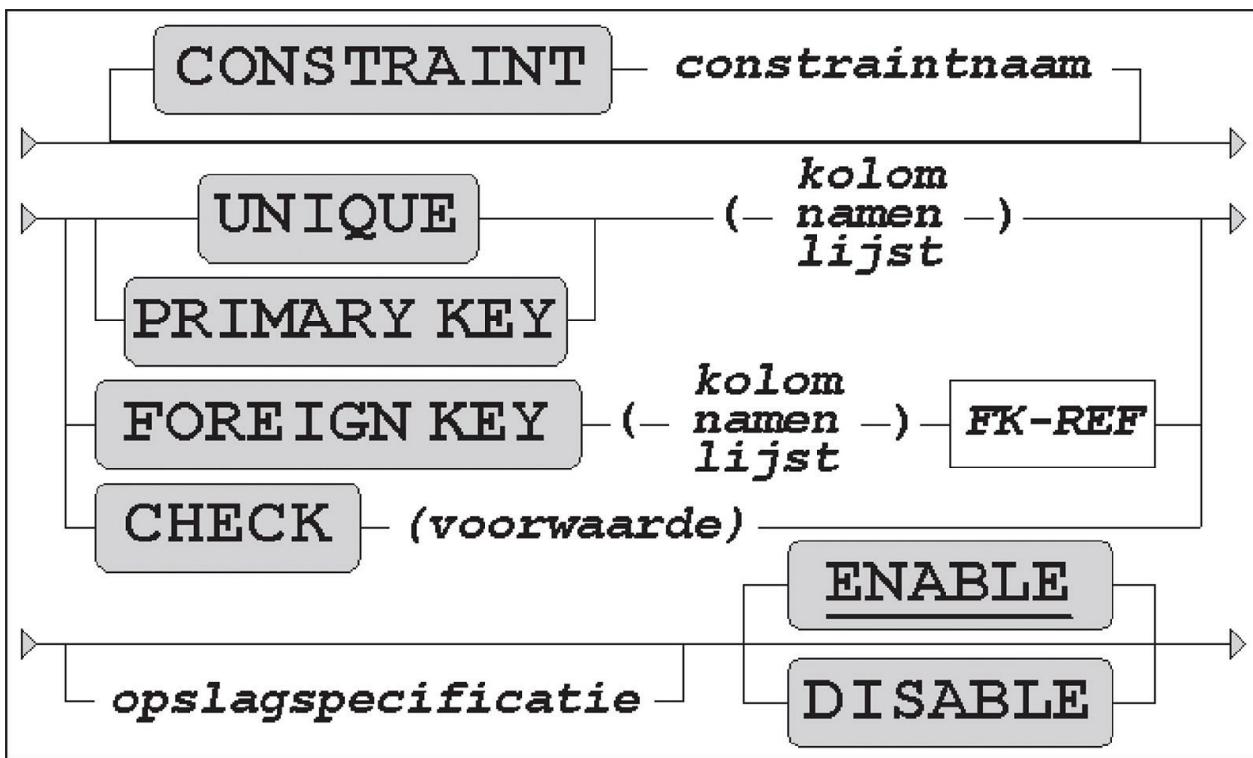
Met deze constraints kunnen we aangeven dat de waarde in een kolom ook voor dient te komen als waarde in een andere kolom (vaak van een andere tabel) waarop ook een sleutel-constraint is gespecificeerd. Denk bijvoorbeeld aan de AFD-kolom van de MEDEWERKERS-tabel waarin we een afdelingsnummer moeten vastleggen dat ook als afdelingsnummer voorkomt

in de AFDELINGEN-tabel. We noemen deze constraints ook wel *refererende* sleutels. In SQL specificeren we deze constraints middels de FOREIGN KEY-constructie. Vaak gebruiken we de term *referentiële integriteit* van een database als we het over de vreemde sleutels van een database hebben.

Constraints kunnen in de CREATE TABLE- en ALTER TABLE-commando's als separaat onderdeel worden gedefinieerd (bijvoorbeeld aan het eind, out of line), of als onderdeel van een kolomdefinitie zelf (inline).

Een optioneel onderdeel van iedere constraint-definitie is de specificatie van een naam (zie bijvoorbeeld [figuur 7.5](#)). Het is beslist aan te raden van deze mogelijkheid gebruik te maken; als we een constraint namelijk zelf geen naam geven, genereert Oracle een (weinig informatieve) naam voor ons: SYS\_C\_nnnnn, waarin nnnnn een willekeurig volgnummer is.

Constraint-namen zijn nodig om constraints later te kunnen manipuleren (aanzetten, uitzetten, verwijderen), en ze komen ook in foutmeldingen 178



voor als ze worden geschonden. Een goedkozen constraint-naam maakt foutmeldingen dan ook informatiever; zie bijvoorbeeld [figuur 7.7](#).

De syntax voor out of line constraints is weergegeven in [figuur 7.5](#), en

de syntax voor inline constraints kunnen we vinden in [figuur 7.8](#). Let met name op het verschil tussen PRIMARY KEY-, UNIQUE- en FOREIGN

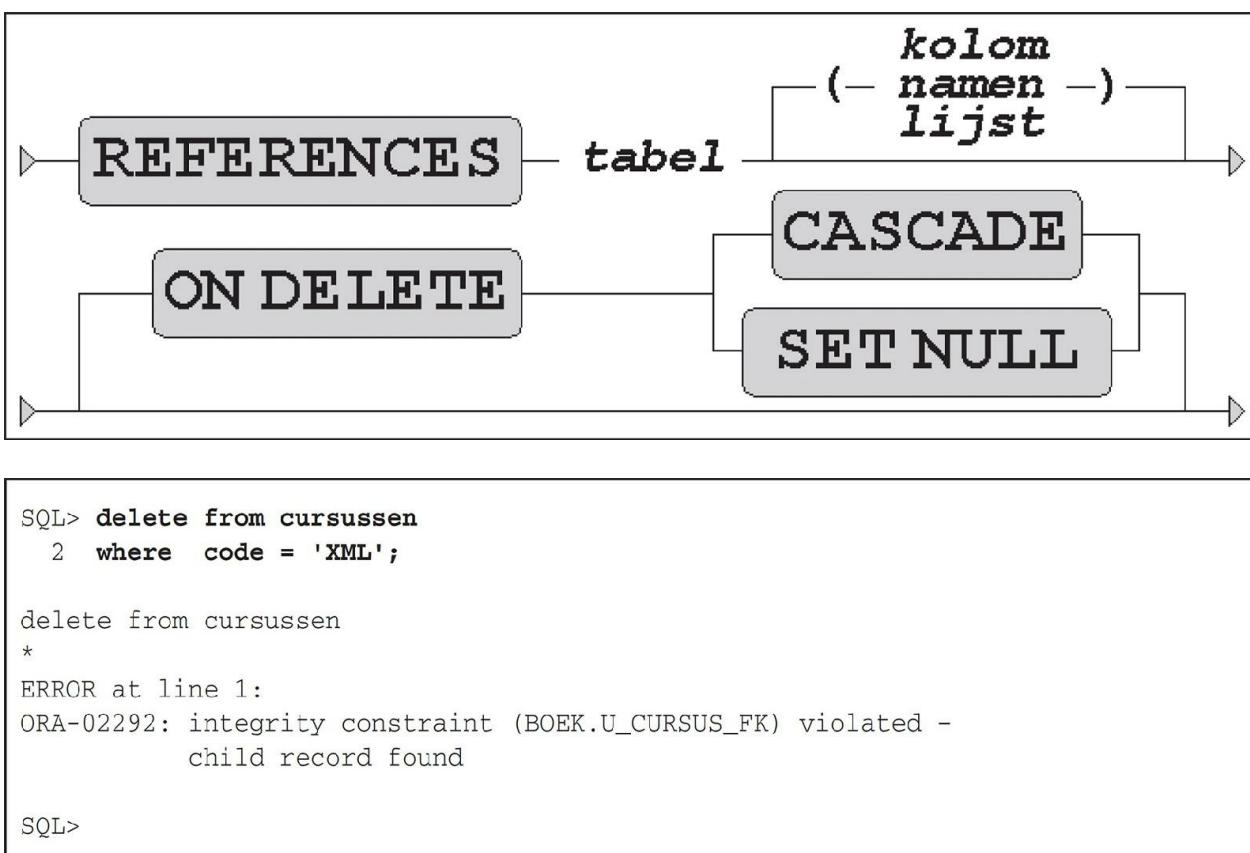
KEY-constraints: als we deze inline specificeren, dan betreffen ze altijd één kolom (namelijk die, waar we ze specificeren). Dit houdt in dat, zodra deze constraints meer dan één kolom betreffen, we verplicht zijn om ze out of line te specificeren omdat we alleen daar expliciet een *kolumnamenlijst* van meer dan één kolom kunnen opgeven. Een soortgelijke restrictie geldt trouwens ook voor een CHECK-constraint: inline kan de voorwaarde slechts één kolom betreffen, out of line kunnen we meerdere kolommen betrekken in de voorwaarde.

## Figuur 7.5

[In figuur 7.5 \(en figuur 7.6\) staat \*kolumnamenlijst\*](#) voor een opsomming van

één of meer kolomnamen, onderling gescheiden door komma's. In principe worden constraints onmiddellijk aangezet, tenzij de DISABLE-optie wordt gebruikt; de default-optie is dus ENABLE. De referentie van een FOREIGN KEY constraint (*FK-REF*) ziet eruit als [in figuur 7.6](#).

De *kolomnamenlijst* kan achterwege gelaten worden, waarbij de referentie automatisch wordt gemaakt naar de primaire sleutel van de tabel.



genoemde *tabel*.

### Figuur 7.6

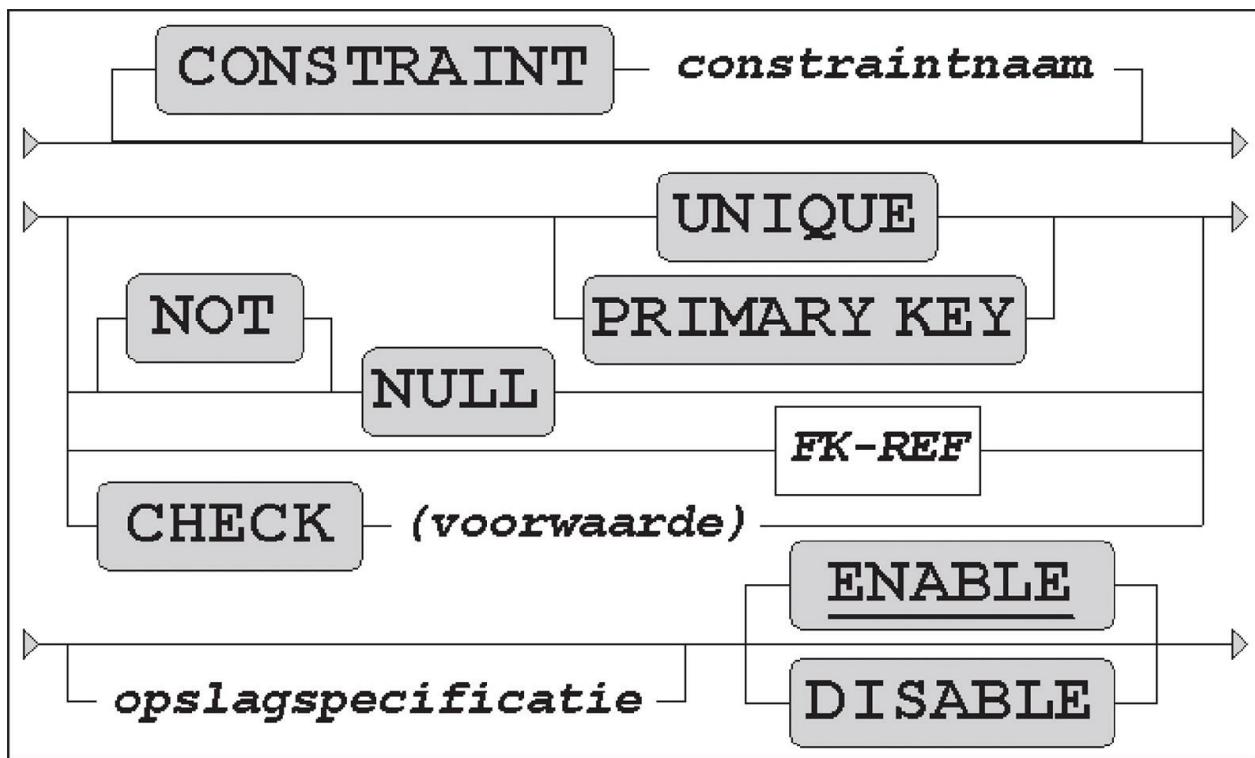
Normaal gesproken is het onmogelijk een ‘parent’-rij te verwijderen als er nog ‘child’-rijen aanwezig zijn; zie [figuur 7.7](#), waarin we proberen een cursus te verwijderen waarvan nog cursusuitvoeringen aanwezig zijn.

### Figuur 7.7

De definitie van de constraint U\_CURSUS\_FK, die hier geschonden wordt, is overigens te zien [in figuur 7.15. De optie](#) ON DELETE CASCADE zorgt in dit soort situaties voor een ander effect: de problemen worden dan opgelost door behalve de parent-rij impliciet alle child-rijen te verwijderen. De optie ON DELETE SET NULL verwijdert de child-rijen niet, maar verandert de refererende kolom(men) in null-waarden; dit is uiteraard alleen mogelijk als die kolommen ook null-waarden mogen bevatten.

Zoals eerder al is aangehaald, is de syntax van *inline constraints* iets afwijkend van de syntax van *out of line constraints*: 180

- 
- 
-



Kolomnamen zijn niet nodig ([zie figuur 7.8](#)) want inline constraints behoren impliciet tot de kolomdefinitie.

De foreign key-referentie ( *FK-REF* ) is hetzelfde als in tabelconstraints ([zie figuur 7.6](#)) maar de woorden FOREIGN KEY hoeven niet te worden vermeld; REFERENCES is genoeg.

In de context van een inline constraint is een NOT NULL-constraint toegestaan; in een tabel-constraint is dat alleen via een omweg mogelijk door gebruik te maken van de CHECK (*voorwaarde*)-constructie, bijvoorbeeld: CHECK(naam is NOT NULL).

Constraint-definities worden in de datadictionary opgeslagen. De twee belangrijkste tabellen zijn USER\_CONSTRAINTS en USER\_CONS\_COLUMNS.

[In figuur 7.9 zien](#) we hoe we een overzicht kunnen krijgen van de referentiële integriteit.

**Figuur 7.8**



```

SQL> select table_name
  2 ,      constraint_name
  3 ,      status
  4 ,      r_constraint_name as references
  5 from user_constraints
  6 where constraint_type = 'R';

TABLE_NAME          CONSTRAINT_NAME      STATUS REFERENCES
-----              -----          -----
AFDELINGEN          A_HOOFD_FK        ENABLED M_PK
HISTORIE            H_MNR_FK         ENABLED M_PK
HISTORIE            H_AFD_FK         ENABLED A_PK
INSCHRIJVINGEN     I_CURSIST_FK      ENABLED M_PK
INSCHRIJVINGEN     I_UITV_FK         ENABLED U_PK
MEDEWERKERS          M_CHEF_FK        ENABLED M_PK
MEDEWERKERS          M_AFD_FK         ENABLED A_PK
UITVOERINGEN        U_CURSUS_FK       ENABLED C_PK
UITVOERINGEN        U_DOCENT_FK       ENABLED M_PK

SQL>

```

```

create table medewerkers
( mnr          NUMBER(4)    constraint M_PK      primary key
, naam        VARCHAR2(12)   constraint M_MNR_CHK  check (mnr > 7000)
, voorl       VARCHAR2(5)    constraint M_VOORL_NN not null
, functie     VARCHAR2(10)
, chef         NUMBER(4)    constraint M_CHEF_FK  references medewerkers
, gbdatum     DATE         constraint M_GEBDAT_NN not null
, maandsal    NUMBER(6,2)   constraint M_MNDSAL_NN not null
, comm         NUMBER(6,2)
, afd          NUMBER(2)    default 10
, constraint  M_VERK_CHK  check (decode(functie,'VERKOPER',0,1) +
                                nvl2(comm,           1,0) = 1)
);

```

## Figuur 7.9

Als illustratie volgen in de figuren 7.10 tot en met 7.17 de commando's waarmee de casustabellen kunnen worden gecreëerd, met daarin specificatie van constraints.

## Figuur 7.10

```

create table afdelingen
( anr      NUMBER(2)    constraint A_PK      primary key
  constraint A_ANR_CHK check ( mod(anr,10) = 0 )
, naam     VARCHAR2(20) constraint A_NAAM_NN not null
  constraint A_NAAM_UN unique
  constraint A_NAAM_CHK check (naam = upper(naam)    )
, locatie  VARCHAR2(20) constraint A_LOC_NN not null
  constraint A_LOC_CHK check (locatie=upper(locatie))
, hoofd    NUMBER(4)     constraint A_HOOFD_FK references medewerkers
) ;

```

```

alter table medewerkers add
(constraint M_AFD_FK foreign key (afd) references afdelingen);

```

```

create table schalen
( snr      NUMBER(2)    constraint S_PK      primary key
, ondergrens NUMBER(6,2) constraint S_ONDER_NN not null
  constraint S_ONDER_CHK check (ondergrens >= 0)
, bovengrens NUMBER(6,2) constraint S_BOVEN_NN not null
, toelage    NUMBER(6,2) constraint S_TOELG_NN not null
, constraint S_OND_BOV   check ( ondergrens <= bovengrens )
) ;

```

```

create table cursussen
( code      VARCHAR2(4)  constraint C_PK      primary key
, omschrijving VARCHAR2(50) constraint C_OMSCHR_NN not null
, type      CHAR(3)      constraint C_TYPE_NN  not null
, lengte    NUMBER(2)     constraint C LENGTE_NN not null
, constraint C_CODE_CHK  check (code = upper(code)    )
, constraint C_TYPE_CHK  check (type in ('ALG','BLD','DSG'))
) ;

```

**Figuur 7.11**

**Figuur 7.12**

**Figuur 7.13**

**Figuur 7.14**

```

create table uitvoeringen
( cursus      VARCHAR2(4)  constraint U_CURSUS_NN not null
, begindatum   DATE        constraint U_BEGIN_NN  not null
, docent       NUMBER(4)   constraint U_DOCENT_FK references medewerkers
, locatie      VARCHAR2(20)
, constraint   U_PK        primary key (cursus,begindatum)
) ;

```

```

create table inschrijvingen
( cursist     NUMBER(4)   constraint I_CURSIST_NN not null
, cursus       VARCHAR2(4)  constraint I_CURSIST_FK references medewerkers
, begindatum   DATE        constraint I_BEGIN_NN  not null
, evaluatie    NUMBER(1)   constraint I_EVAL_CHK
                           check      (evaluatie in (1,2,3,4,5) )
, constraint   I_PK        primary key (cursist,cursus,begindatum)
, constraint   I_UITV_FK   foreign key (cursus,begindatum)
                           references uitvoeringen
) ;

```

```

create table historie
( mnr        NUMBER(4)   constraint H_MNR_NN  not null
,           NUMBER(4)   constraint H_MNR_FK  references medewerkers
                           on delete cascade
, beginjaar   NUMBER(4)   constraint H_BJAAR_NN not null
, begindatum   DATE        constraint H_BEGIN_NN not null
, einddatum    DATE
, afd         NUMBER(2)   constraint H_AFD_NN  not null
,           NUMBER(2)   constraint H_AFD_FK  references afdelingen
, maandsal    NUMBER(6,2)  constraint H_SAL_NN  not null
, opmerkingen VARCHAR2(60)
, constraint   H_PK        primary key (mnr,begindatum)
, constraint   H_BEG_EIND  check      (begindatum < einddatum)
) ;

```

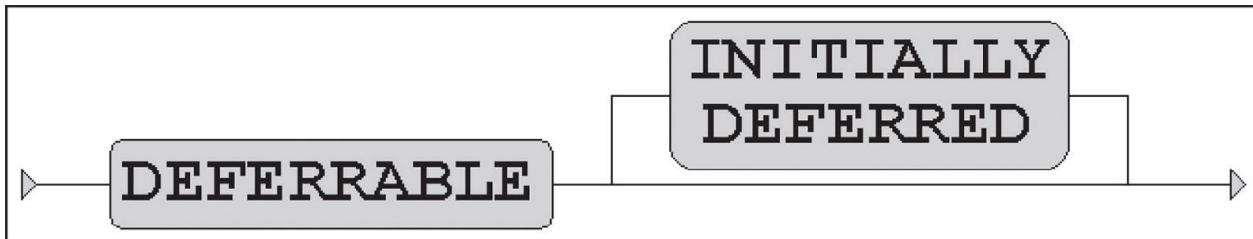
**Figuur 7.15****Figuur 7.16**

## Figuur 7.17

Overigens kent Oracle het SQL-commando CREATE SCHEMA, waarmee we onder andere in staat zijn om in één DDL-transactie een aantal tabellen tegelijk te creëren. Het voordeel daarvan is dat de actie als geheel slaagt of faalt; bovendien zijn we dan af van het probleem dat hierboven met een ALTER TABLE-commando moet worden opgelost (een tabel waarnaar een constraint verwijst moet al bestaan). We kunnen 184

```
SQL> create schema authorization BOEK
  2      create table medewerkers (... )
  3      create table afdelingen (... )
  4      create table cursussen (... )
  5      create table uitvoeringen (... )
  6      create table inschrijvingen (... )
  7      create table historie (... );
```

- 
- 
- 



de casustabellen dus ook als volgt creëren ([zie figuur 7.18](#)).

## Figuur 7.18

In dit commando kunnen eventueel ook in één moeite door views worden gecreëerd en privileges worden verleend. De volgorde waarin de componenten binnen het CREATE SCHEMA-commando worden geplaatst, is volledig vrij.

Oracle ondersteunt ook *deferrable constraints*, hetgeen wil zeggen dat kan worden aangegeven wanneer een constraint moet worden

gecontroleerd. Dit zijn de twee mogelijkheden:

Onmiddellijk (immediate): op statement niveau.

Uitgesteld (deferred): aan het einde van de transactie.

Met een deferred foreign key-constraint zijn we dus in staat om wel een ‘parent’-rij te verwijderen als er nog ‘child’-rijen aanwezig zijn.

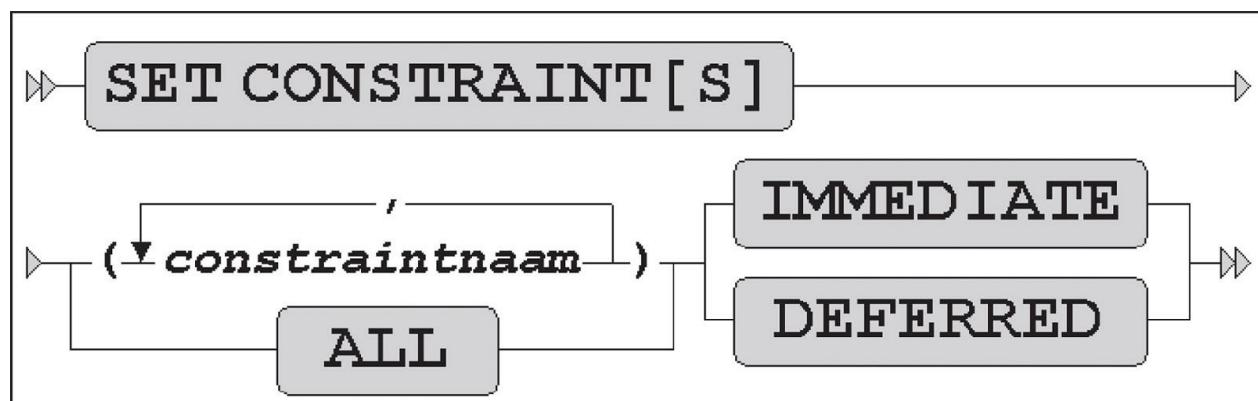
En als we in dezelfde transactie alsnog ook de bijbehorende ‘child’-rijen verwijderen, zal de controle van deze foreign key-constraint (die nu bij COMMIT plaatsvindt) slagen.

Bij het definiëren van de constraint dient expliciet te worden vastgelegd dat uitstel van controle toegestaan is; de default is NOT DEFERRABLE. In

de constraint-definitie wordt dan vóór de opslagspecificatie (zie figuur 7.5 en 7.8) het volgende toegevoegd:

### Figuur 7.19

Deferrable constraints staan default op INITIALLY IMMEDIATE: dat wil  
185



zeggen ze kunnen op DEFERRED gezet worden, maar worden default op statement niveau gevalideerd. In de praktijk zie dat als er gebruik gemaakt wordt van deferrable constraint, het default gedrag ook expliciet op

INITIALLY DEFERRED gezet wordt. Dit gedrag kan in een transactie aangepast worden met het commando SET CONSTRAINTS (zie [figuur 7.20](#)).

## Figuur 7.20

7.5

### Indexen

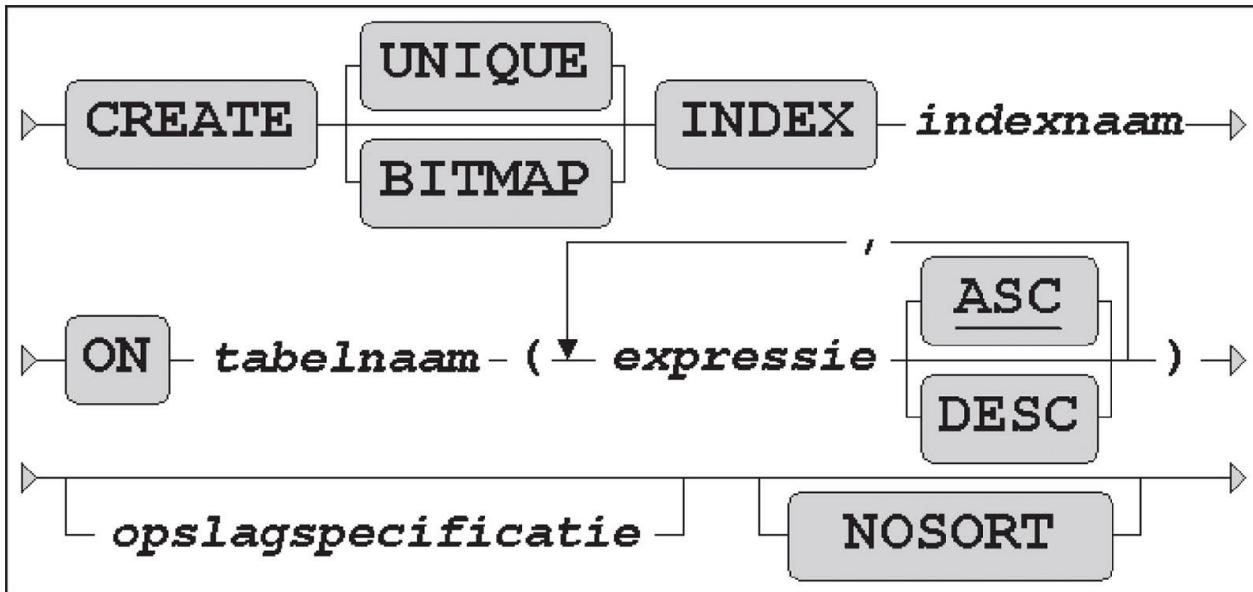
De rijen binnen een tabel hebben over het algemeen een willekeurige volgorde. We mogen er in ieder geval tijdens raadpleging niet van uitgaan dat de rijen op een bepaalde manier geordend zijn opgeslagen.

Stel nu dat de medewerkerstabel 50.000 rijen bevat, en dat we een overzicht willen van alle medewerkers waarvan de naam met een B

begint. Normaal gesproken is er dan maar één methode denkbaar: alle rijen van de tabel zullen moeten worden doorlopen,

en van iedere rij moet de naam worden onderzocht. Het antwoord op deze vraag zal enige tijd op zich laten wachten, omdat er vrij veel leesopdrachten en vergelijkingen moeten worden uitgevoerd.

Een index zou hier uitkomst kunnen bieden. Dat wil zeggen dat het systeem een gesorteerde lijst bijhoudt van alle voorkomende namen, met daarbij een verwijzing naar de corresponderende rij(en) van de tabel. Nu kan namelijk deze index worden doorlopen tot de eerste naam die met een B begint; vanaf dat punt wordt doorgezocht tot de eerste naam die met een C begint. Met behulp van de verwijzingen worden de gewenste rijen uit de tabel gehaald.



Het aantal benodigde leesopdrachten én vergelijkingen wordt op deze wijze aanzienlijk verminderd, waardoor de responstijd van de query waarschijnlijk zal verbeteren.

Voor een andere query zou misschien een index op afdelingsnummer of geboortedatum handig zijn. Zo kunnen op iedere tabel meerdere indexen worden gedefinieerd.

Het is duidelijk dat queries voordeel kunnen hebben van indexen. Dit geldt zeker als de tabellen groot zijn en de query sterk selectief is. Met andere woorden: als er relatief weinig rijen van de geraadpleegde tabel(len) in het resultaat zullen verschijnen.

[De syntax van het commando CREATE INDEX is als volgt \(zie figuur](#)

[7.21\).](#)

### Figuur 7.21

Unieke indexen hebben niet alleen als functie de tabel sneller toegankelijk te maken, maar ook duplicitaardelen te voorkomen.

Uniciteit kan overigens het beste met de constraint PRIMARY KEY of UNIQUE worden bewaakt.

Indexen zijn voornamelijk efficiënt als er zo veel mogelijk verschillende waarden in de geïndexeerde kolom(men) voorkomen. Oracle

ondersteunt naast ‘gewone’ indexen ook *bitmap-indexen*; dit zijn indexen die juist geschikt zijn voor kolommen waarin niet al te veel verschillende waarden voorkomen.

187

- 
- 
- 

[In figuur 7.21 kunnen we als expressie](#) niet alleen kolomnamen gebruiken, maar ook expressies. Deze indexen worden ook wel *functiegebaseerde indexen* genoemd. Zo is het bijvoorbeeld mogelijk om een index te maken op de volgende expressie (functie):

$(12 * \text{MAANDSAL} + \text{COALESCE}(\text{COMM}, 0))$

Met de *opslagspecificatie* kunnen we invloed uitoefenen op de plaats waar en de manier waarop de index ruimte zal alloceren in de database.

We gaan daar niet verder op in. Als de rijen toevallig al op volgorde zijn ingevoerd, kunnen we de NOSORT-optie gebruiken. Oracle zal dan de sorteerslag die normaal gesproken bij het maken van een index nodig is, achterwege laten. Dat zal enige tijdwinst opleveren tijdens het aanmaken van de index.

Een keerzijde van de (index)medaille is dat datamanipulatie trager zal verlopen; iedere wijziging in de tabelinhoud wordt immers door Oracle automatisch ook onmiddellijk in alle indexen doorgevoerd. Bovendien kosten indexen extra opslagruimte. Vandaar dat zorgvuldig moet worden afgewogen op welke kolom(men) indexen zullen worden gecreëerd. Richtlijnen zouden kunnen zijn:

Indexen op alle refererende sleutels.

Indexen op kolommen die vaak in where-condities van queries voorkomen.

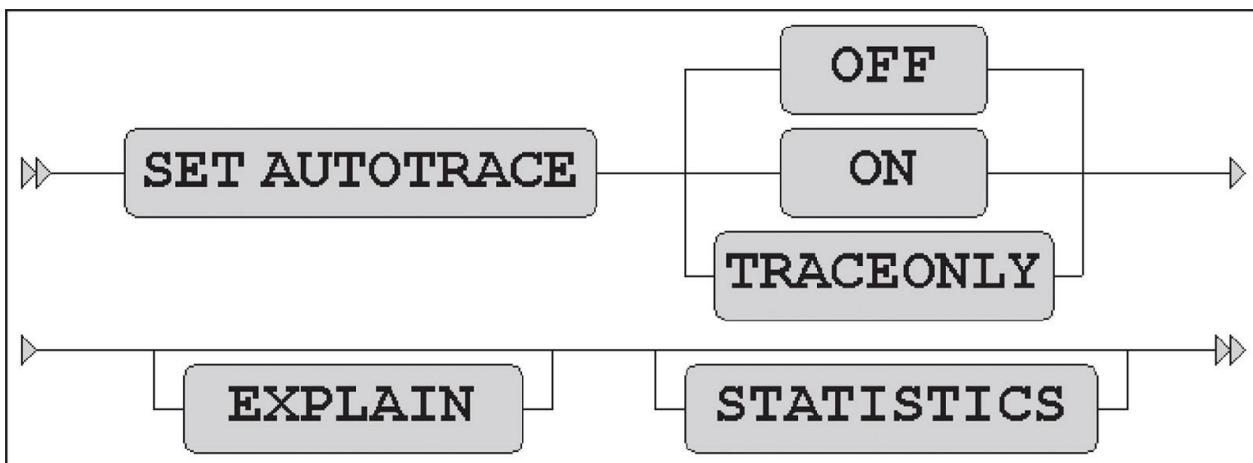
Indexen op kolommen die vaak in order-by-clauses van queries voorkomen.

We kunnen indexen verwijderen met behulp van het commando DROP

INDEX:

SQL> **drop index <indexnaam>** ; In een periode waarin relatief veel datamanipulatie wordt gepleegd en weinig queries worden uitgevoerd, kan men overwegen indexen tijdelijk te verwijderen. Een typisch praktijk voorbeeld hiervan is het uitvoeren van een grote datamigratie en/of -conversie. Door gericht de indexen die tijdens deze periode niet nodig zijn even te verwijderen, zal de datamanipulatie sneller verlopen. Na afloop van zo'n periode kunnen deze indexen dan weer opnieuw worden gecreëerd.

Wij kunnen als gebruikers met de commando's CREATE INDEX en DROP



INDEX beslissen over het al dan niet bestaan van indexen; of ze daadwerkelijk tijdens raadpleging worden gebruikt, is altijd een beslissing van het RDBMS. Om precies te zijn: deze beslissing wordt genomen door een onderdeel van het RDBMS dat de *optimizer* wordt genoemd. De optimizer bepaalt voor ieder SQL-commando de strategie waarmee het zal worden uitgevoerd, en of daarbij indexen worden gebruikt of niet.

## 7.6

### Performance

Het is mogelijk de strategie van de optimizer met betrekking tot een SQL-statement op te vragen en deze vervolgens te analyseren, met behulp van de zogeheten *diagnostic tools* van Oracle (TRACE, TKPROF en EXPLAIN). Bespreking van deze zeer nuttige hulpmiddelen valt buiten het bestek van dit boek; zie de Oracle-documentatie voor details (Performance Tuning Guide).

SQL\*Plus biedt voor dit soort analyses een zeer gebruiksvriendelijk alternatief: de setting AUTOTRACE.

### Figuur 7.22

Hierbij wordt een tabel met de naam PLAN\_TABLE aanwezig

verondersteld. Het script om deze tabel te creëren heet utlxplan.sql, en wordt door Oracle standaard meegeleverd (in de subdirectory `rdbms\admin`). Verder dient de gebruiker bepaalde privileges te hebben, die in

de database-rol PLUSTRACE zijn ondergebracht. Deze rol kan worden gecreëerd met het script plustrce.sql, door Oracle meegeleverd in de subdirectory sqlplus.

189

```
SQL> set autotrace on explain
SQL> select naam from medewerkers where mnr < 7500;

NAAM
-----
SMIT
ALDERS

Execution Plan
-----
0      SELECT STATEMENT Optimizer=ALL_ROWS
          (Cost=2 Card=3 Bytes=36)
1      0      TABLE ACCESS (BY INDEX ROWID) OF 'MEDEWERKERS' (TABLE)
          (Cost=2 Card=3 Bytes=36)
2      1      INDEX (RANGE SCAN) OF 'M_PK' (INDEX (UNIQUE))
          (Cost=1 Card=3)

SQL>
```

Laten we eens het voorbeeld [figuur 7.23 bekijken](#).

### Figuur 7.23

Hieruit kunnen we opmaken dat de optimizer gekozen heeft voor een [benadering via de unieke index M\\_PK op de primaire sleutel](#). In figuur [7.24 zien we hoe met de AUTOTRACE optie TRACEONLY STATISTICS het](#)

resultaat van het commando (de uitvoer) wordt onderdrukt, en hoe allerlei interessante statistieken kunnen worden opgevraagd, waarvan de besprekking hier te ver zou voeren.

```

SQL> set autotrace traceonly statistics
SQL> select * from medewerkers;

Statistics
-----
0 recursive calls
0 db block gets
4 consistent gets
0 physical reads
0 redo size
1485 bytes sent via SQL*Net to client
499 bytes received via SQL*Net from client
2 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
14 rows processed

SQL> set autotrace off
SQL>

```

**Figuur 7.24**

## 7.7

### Sequences

Nu we het toch over performance hebben: in informatiesystemen komt het vaak voor dat unieke *volgnummers* moeten worden gegenereerd, bijvoorbeeld voor orders of facturen. Dat kan worden geïmplementeerd met een hulptabel waarin we het eerstvolgende vrije volgnummer bijhouden, maar dat kan met name in een multi-useromgeving een ernstig performance-probleem opleveren. In dergelijke situaties zijn *sequences* een uitstekend alternatief.

Sequences kunnen respectievelijk worden gecreëerd, gewijzigd of verwijderd met het volgende drietal SQL-commando's:

SQL> **create sequence** <*sequencenaam*> ...

SQL> **alter sequence** <*sequencenaam*> ...

SQL> **drop sequence** <*sequencenaam*>; Een sequence-definitie kan onder

meer de volgende componenten bevatten: startwaarde, stapgrootte, minimum-of maximumwaarde.

Verder kunnen we nog aangeven of de volgnummers cyclisch moeten zijn of niet.

191

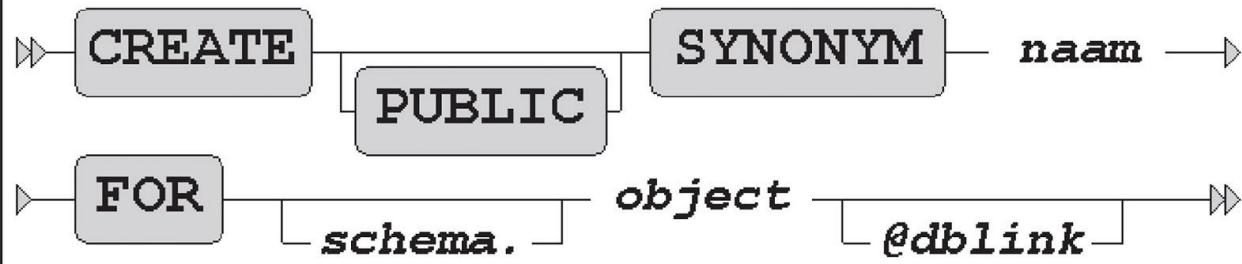
```
SQL> create sequence afd_nr
  2  start with 50 increment by 10;

Sequence created.

SQL> select afd_nr.nextval, afd_nr.currvall
  2  from dual;

NEXTVAL    CURRVAL
-----  -----
      50        50

SQL>
```



Een sequence heeft twee pseudo-kolommen: NEXTVAL en CURRVAL, waarvan de betekenis voor zich spreekt. [In figuur 7.25 creëren we](#)

bijvoorbeeld een sequence AFD\_NR, waarmee we afdelingsnummers zouden kunnen genereren. We laten hier het gebruik alleen zien op de DUAL-tabel.

**Figuur 7.25**

7.8

Synoniemen

Met behulp van het commando CREATE SYNONYM kunnen synoniemen worden gedefinieerd voor tabel- of view-namen. Deze synoniemen kunnen in alle SQL-commando's worden toegepast als alternatief voor de feitelijke namen die erachter schuilgaan.

### Figuur 7.26

Synoniemen kunnen bijvoorbeeld worden toegepast als tabelnamen erg lang zijn. Vooral als men regelmatig objecten van andere gebruikers raadpleegt zijn synoniemen plezierig; anders moet steeds de naam van de eigenaar (schema) voor de tabelnaam worden gespecificeerd. De tabellen van de datadictionary zijn hiervan een prima voorbeeld; we 192

```
SQL> show user
User is "BOEK"
SQL> alter session set current_schema=scott;
Session altered.

SQL> select * from dept;

DEPTNO DNAME          LOC
----- -----
 10 ACCOUNTING      NEW YORK
 20 RESEARCH        DALLAS
 30 SALES           CHICAGO
 40 OPERATIONS      BOSTON

SQL> alter session set current_schema=boek;
Session altered.

SQL>
```

kunnen ze gewoon raadplegen, hoewel we natuurlijk zelf geen eigenaar zijn van deze centrale tabellen.

Overigens is er een alternatief om zonder synoniemen toch andermans tabellen te raadplegen zonder de naam van de eigenaar steeds te specificeren: met behulp van het ALTER SESSION-commando. Stel dat het demo-schema SCOTT met de tabellen EMP en DEPT ook in onze database aanwezig is; dan kunnen we het ALTER SESSION-commando als volgt gebruiken (zie [figuur 7.27](#)):

## **Figuur 7.27**

Dit werkt zo ongeveer hetzelfde als het veranderen van directory (bijvoorbeeld met het CD-commando) in een besturingssysteem; we kunnen nu alle objecten direct aanspreken. Dit verandert overigens niets aan onze privileges; de query hierboven lukt omdat gebruiker BOEK

blijkbaar SELECT privileges heeft op de tabel DEPT van gebruiker SCOTT.

We maken onderscheid tussen privé-synoniemen en publieke

synoniemen. Publieke synoniemen kunnen worden gebruikt door alle databasegebruikers, maar mogen alleen door een databasebeheerder worden gecreëerd. De synoniemen op de datadictionary zijn

bijvoorbeeld van dit laatste type. Privé-synoniemen kunnen ook door gewone databasegebruikers worden gemaakt en kunnen dan alleen door die gebruiker worden toegepast.

```
SQL> create synonym m for medewerkers;
Synonym created.
```

```
SQL> describe m
Name           Null?    Type
-----
MNR            NOT NULL NUMBER(4)
NAAM           NOT NULL VARCHAR2(12)
VOORL          NOT NULL VARCHAR2(5)
FUNCTIE        VARCHAR2(10)
CHEF           NUMBER(4)
GBDATUM       NOT NULL DATE
MAANDSAL      NOT NULL NUMBER(6,2)
COMM           NUMBER(6,2)
AFD            NUMBER(2)
```

```
SQL> select * from cat;
```

TABLE_NAME	TABLE_TYPE
AFDELINGEN	TABLE
AFD_NR	SEQUENCE
CURSUSSEN	TABLE
HISTORIE	TABLE
INSCHRIJVINGEN	TABLE
M	SYNONYM
MEDEWERKERS	TABLE
SCHALEN	TABLE
UITVOERINGEN	TABLE

```
SQL> select synonym_name, table_owner, table_name
  2  from user_synonyms;
```

SYNONYM_NAME	TABLE_OWNER	TABLE_NAME
M	BOEK	MEDEWERKERS

```
SQL> drop synonym m;
Synonum dropped.
```

```
SQL>
```

## **Figuur 7.28**

Ten slotte worden synoniemen met name in gedistribueerde databases 194



gebruikt om lokatie onafhankelijkheid te implementeren: de gebruiker of applicatie hoeft niet te weten waar (in welke database) tabellen of views zich in het netwerk bevinden. Dat gebeurt met de optie @ via zogeheten *database-links*.

7.9

DROP TABLE

Met het commando

SQL> **drop table <tabelnaam>** ; wordt een tabel uit de database verwijderd. Het effect van dit commando is onherroepelijk; er is geen ROLLBACK mogelijk. Deze eigenschap geldt overigens voor alle DDL-commando's van SQL, zoals we in het vorige hoofdstuk hebben gezien.

Omdat het verwijderen van tabellen toch wel eens per ongeluk kan gebeuren, is sinds Oracle 10 g het concept van een prullenmand (RECYCLE BIN) aanwezig. Alle tabellen die worden verwijderd komen in eerste instantie in de prullenmand terecht. We kunnen de RECYCLE BIN

bekijken met behulp van USER\_RECYCLEBIN view:

SQL> **select \* from user\_recyclebin;**

We kunnen tabellen uit de RECYCLE BIN terughalen met behulp van het commando

FLASHBACK TABLE:

SQL> **flashback table <tabelnaam> to before drop 2 [rename to <nieuwe naam>];**

**Let op:** We hebben niet de garantie dat dit commando altijd slaagt.

Het kan namelijk zijn dat de RECYCLE BIN inmiddels expliciet (door een databasebeheerder) of impliciet (door Oracle) is geleegd.

Willen we een tabel verwijderen, zonder dat hij in de RECYCLE BIN terecht komt, dan kan dat door aan het drop table commando de optie PURGE toe te voegen:

195

SQL> **drop table < tabelnaam> purge;** Met het droppen van een tabel worden ook allerlei daarmee

samenhangende zaken uit de database verwijderd, zoals privileges op de tabel die aan bepaalde gebruikers waren verleend, en indexen die op de inhoud waren gebaseerd. Dat betekent dat bij een reorganisatie van de database niet kan worden volstaan met het opnieuw creëren van de tabel, maar dat ook de hele structuur eromheen opnieuw moet worden opgezet.

Het kan voorkomen dat een tabel niet zomaar kan worden verwijderd omdat er vanuit een andere tabel een refererende sleutel op is gedefinieerd. Dit probleem kan worden opgelost met de optie CASCADE

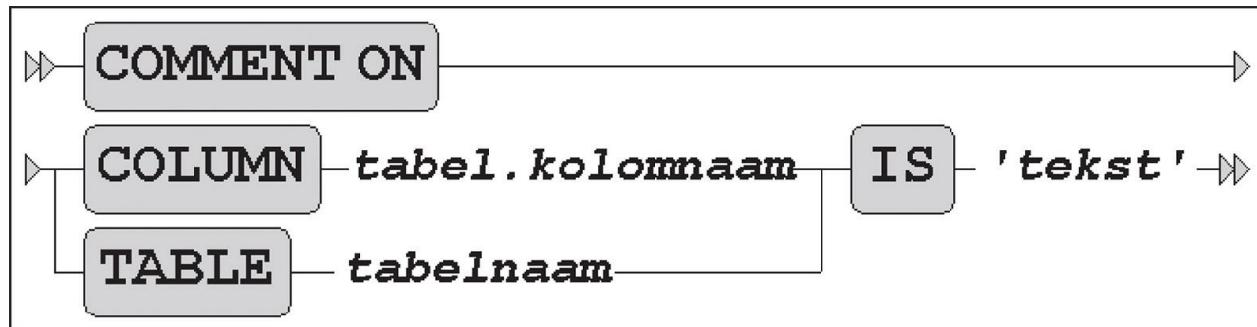
CONSTRAINTS, waardoor impliciet alle constraints worden gedropt die van (sleutel-constraints op) de tabel afhankelijk zijn:

SQL> **drop table < tabelnaam> cascade constraints;** 7.10 Overige commando's

Oracle SQL heeft een speciaal commando om met name grote tabellen op efficiënte wijze leeg te maken zonder ze te verwijderen: SQL> **truncate table < tabelnaam>** ; Dit commando heeft niet het hierboven genoemde nadeel van het commando DROP TABLE (het verloren gaan van privileges en indexen), en is bovendien aanzienlijk sneller dan het commando DELETE. De prijs die hiervoor wordt betaald, is dat een rollback onmogelijk is; de transactie wordt onmiddellijk gecommit.

Met behulp van het volgende commando kan de naam van een tabel (of view: zie hoofdstuk 10) worden veranderd:

SQL> **rename <oude\_naam> to <nieuwe\_naam>** ; Met behulp van het commando COMMENT kan men verklarende tekst bij tabellen en/of kolommen opslaan in de datadictionary, zoals blijkt uit 196



```

SQL> comment on table schalen is 'salarisschalen en netto toelages';
Comment created.

SQL> comment on column medewerkers.comm is 'alleen voor verkopers';
Comment created.

SQL> select comments from user_tab_comments
  2 where table_name = 'SCHALEN';

COMMENTS
-----
salarisschalen en netto toelages

SQL> select comments from user_col_comments
  2 where table_name = 'MEDEWERKERS'
  3 and column_name = 'COMM';

COMMENTS
-----
alleen voor verkopers

SQL>

```

[figuur 7.29.](#)

### Figuur 7.29

In [figuur 7.30 wordt](#) eerst op een tabel en een kolom commentaar gecreëerd, en daarna wordt de datadictionary geraadpleegd.

### Figuur 7.30

#### 7.11 Opgaven

1

In [figuur 7.10 wordt](#) de constraint M\_VERK\_CHK op een nogal cryptische manier gedefinieerd; formuleer dezelfde constraint zonder DECODE en NVL2 te gebruiken.

2

Waarom zou de constraint in [figuur 7.12](#) met een apart ALTER

■

TABLE-commando moeten worden gedefinieerd?

3

Het is weliswaar niet behandeld, maar beredeneer waarom bij het gebruik van sequences de pseudo-kolom CURRVAL in een transactie niet kan worden gebruikt zonder eerst een beroep te doen op NEXTVAL.

4

Hoe komt het dat de kolom evaluatie van de tabel inschrijvingen null-waarden accepteert, ondanks de constraint I\_EVAL\_CHK (zie

[figuur 7.16](#))?

5

Als een PRIMARY KEY of UNIQUE constraint wordt gedefinieerd, creëert Oracle normaal gesproken impliciet een unieke index om de constraint te kunnen bewaken. Onderzoek wat er gebeurt als een dergelijke constraint DEFERRABLE wordt gedefinieerd.

6

Met behulp van functiegebaseerde indexen kunnen we ‘conditionele uniciteit’ implementeren. Maak een unieke index op de

INSCHRIJVINGEN-tabel die ervoor zorgt dat de OAG-cursus maar één keer mag worden gevuld. Test de oplossing met het volgende commando (dat zou moeten falen):

SQL> insert into inschrijvingen values

(7900,’OAG’,sysdate,null);

**Tip:** We kunnen hiervoor een CASE-expressie gebruiken.

## Hoofdstuk 8

### Raadpleging – meerdere tabellen en aggregatie

In dit hoofdstuk gaan we dieper in op de raadpleegmogelijkheden van SQL; het sluit wat dat betreft aan op de hoofdstukken 4 en 5.

Allereerst wordt het begrip rij- of tuple-variabele geïntroduceerd.

Hiervan hebben we tot nu toe geen gebruik gemaakt. Als we in de from-component echter meerdere tabellen gaan specificeren, is het verstandig om deze tuple-variabelen – in Oracle ook wel ‘tabel-aliasen’ genoemd – consequent toe te passen.

[In paragraaf 8.2 worden](#) joins besproken; [paragraaf 8.3 gaat](#) over de ANSI/ISO standaard join syntax en [paragraaf 8.4 gaat](#) over outer joins.

Vaak zijn we geïnteresseerd in geaggregeerde informatie, zoals een overzicht van het aantal deelnemers per cursus, met het gemiddelde evaluatiecijfer. De daarvoor benodigde queries

kunnen we met behulp van group by formuleren. Daarbij spelen groepsfuncties een belangrijke rol. Met having kunnen we

vervolgens op groepen selecteren. Deze onderwerpen komen aan bod in de paragrafen 8.5, 8.6 en 8.7.

In de achtste paragraaf van dit hoofdstuk komen nog enkele geavanceerde mogelijkheden van de group by-component aan de orde, zoals CUBE, ROLLUP, GROUPING SETS, en gepartitioneerde outer joins.

[In paragraaf 8.9 worden](#) de drie verzamelingsoptatoren van SQL besproken:

UNION, MINUS en INTERSECT.

Het hoofdstuk wordt afgesloten met opgaven.

8.1

Tuple-variabelen

Tot nu toe hebben we SQL-commando's op de volgende manier ingegeven:

199

**SQL> select naam, voorl, functie**

**2 from medewerkers**

**3 where afd = 20**

Feitelijk is dit commando nogal onvolledig. In dit hoofdstuk moeten we de puntjes op de i gaan zetten, omdat de SQL-commando's langzaam maar zeker gecompliceerder worden. De volledige formulering van hetzelfde commando is als volgt:

**SQL> select m.naam, m.voorl, m.functie**

**2 from medewerkers m**

**3 where m.afd = 20**

Hierin is *m* een *tuple-variabele*. Tupel is een deftig woord voor rij.

Tuple-variabelen worden ook wel *aliasen* genoemd, en de ANSI/ISO-standaard heeft het over *correlatiennenamen*. Let op de syntax: in de FROM-component 'declareren' we de tuple-variabele, en wel achter de tabelnaam. De tuple-variabele loopt over de tabel, met andere woorden: *m* heeft steeds als waarde een rij uit de medewerkerstabel. Binnen de context van een specifieke rij kunnen we refereren naar een kolomwaarde, zoals in de SELECT- en

WHERE-component van het voorbeeld gebeurt. De tuple-variabele komt voor de kolomnaam, met een punt als scheidingsteken ertussen.

We kunnen de verwerking van het gegeven SQL-commando als volgt voorstellen:

1

De tuple-variabele  $m$  loopt rij voor rij over de tabel MEDEWERKERS (de volgorde is daarbij onbelangrijk).

2

Iedere rij  $m$  die voldoet aan de WHERE-component komt in een tussenresultaat terecht.

3

Voor iedere rij van het tussenresultaat worden de expressies van de SELECT-component ‘berekend’.

In eenvoudige queries hoeven we ons niet druk te maken om tuple-variabelen; Oracle begrijpt toch wel wat we bedoelen. Zodra commando’s echter complexer worden, is het verstandig – of zelfs noodzakelijk – om ze toe te passen. In ieder geval verhogen ze de leesbaarheid en onderhoudbaarheid.

200

```
SQL> select anr, locatie, naam, voorl
  2  from medewerkers, afdelingen;
select anr, locatie, naam, voorl
*
ERROR at line 1:
ORA-00918: column ambiguously defined
SQL>
```

8.2

## Joins

We kunnen in de FROM-component van een query meerdere tabellen aanspreken. Let op wat er bij het volgende commando in [figuur 8.1](#) gebeurt.

### Figuur 8.1

Uit dit commando is niet op te maken welke NAAM-kolom we bedoelen: beide tabellen hebben namelijk een kolom NAAM, vandaar de foutmelding.

We wagen een tweede poging ([zie figuur 8.2](#)). Nu krijgen we wel een resultaat, maar dit is ook weer niet wat we bedoeld hadden. De tuplevariabelen  $m$  en  $a$  lopen vrij over beide tabellen; er is geen WHERE-component die ze daarbij iets in de weg legt. Het resultaat is het *Cartesiaans product* van beide tabellen, resulterend in 56 rijen: 14

medewerkers maal 4 afdelingen geeft 56 mogelijke combinaties.

```

SQL> select a.anr, a.locatie, m.naam, m.voort
  2  from medewerkers m, afdelingen a;

ANR LOCATIE        NAAM      VOORT
----- -----        -----
 10 LEIDEN          SMIT      N
 10 LEIDEN          ALDERS    JAM
 10 LEIDEN          DE WAARD  TF
 10 LEIDEN          JANSEN    JM
 10 LEIDEN          MARTENS  P
 10 LEIDEN          BLAAK     R
...
 40 GRONINGEN       ADAMS    AA
 40 GRONINGEN       JANSEN    R
 40 GRONINGEN       SPIJKER   MG
 40 GRONINGEN       MOLENAAR  TJA

56 rows selected.

SQL>

```

## Figuur 8.2

Daarmee is ook duidelijk wat er nog aan de query ontbreekt: een WHERE-component. We voegen er ook een ORDER BY-component aan toe (zie [figuur 8.3](#)).

```

SQL> select a.anr, a.locatie
  2 ,      m.naam, m.voorl
  3 from medewerkers m, afdelingen a
  4 where m.afd = a.anr
  5 order by a.anr, m.naam;

ANR LOCATIE          NAAM      VOORL
-----  -----
 10 LEIDEN           CLERCKX    AB
 10 LEIDEN           DE KONING  CC
 10 LEIDEN           MOLENAAR   TJA
 20 DE MEERN         ADAMS     AA
 20 DE MEERN         JANSEN    JM
 20 DE MEERN         SCHOTTEN  SCJ
 20 DE MEERN         SMIT      N
 20 DE MEERN         SPIJKER   MG
 30 UTRECHT          ALDERS    JAM
 30 UTRECHT          BLAAK     R
 30 UTRECHT          DE WAARD   TF
 30 UTRECHT          DEN DRAAIER JJ
 30 UTRECHT          JANSEN    R
 30 UTRECHT          MARTENS   P

```

14 rows selected.

SQL>

```

SQL> select a.anr
  2 ,      a.locatie
  3 ,      m.naam
  4 ,      m.voorl
  5 from medewerkers m
  6 ,      afdelingen a
  7 where m.afd = a.anr
  8 order by a.anr
  9 ,      a.locatie

```

### Figuur 8.3

We noemen de WHERE-component op regel 4 hierboven, de *join-conditie*.

Het wordt steeds belangrijker om onze SQL-commando's zo overzichtelijk mogelijk te formuleren, zowel vanwege de leesbaarheid als

vanwege de onderhoudbaarheid. Het laatste commando kunnen we ook als volgt ingeven, over meerdere regels verdeeld:

Deze methode heeft in de praktijk bewezen erg handig te zijn. Let op de plaatsing van de komma's, waardoor het onderhoud (het verwijderen en toevoegen van regels) vrij eenvoudig is.

```

SQL> select m.naam          medewerker
  2 ,      m.maandsal+s.toelage totaalsalaris
  3 from  medewerkers m
  4 ,      schalen     s
  5 where m.maandsal between s.ondergrens
       and s.bovengrens;

```

MEDEWERKER	TOTAALSALARIS
SMIT	800
JANSEN	800
ADAMS	1100
DE WAARD	1300
MARTENS	1300
MOLENAAR	1350
DEN DRAAIER	1600
ALDERS	1700
CLERCKX	2650
BLAAK	3050
JANSEN	3175
SCHOTTEN	3200
SPIJKER	3200
DE KONING	5500

14 rows selected.

```
SQL>
```

Wat we nu hebben heet een *join*, om precies te zijn: een *equi-join*. Dit type join komt vrij veel voor. Als we in plaats van het isgelijkteken een andere vergelijkingsoperator gebruiken, dan heet het een non-equijoin (zie bijvoorbeeld [figuur 8.4](#)).

#### Figuur 8.4

Overigens, wat de naamgeving van tuple-variabelen als *m* en *s* betreft: die is geheel vrij. De enige reden waarom in dit boek zoveel mogelijk gekozen wordt voor de beginletters van de tabelnamen is de leesbaarheid. Ook meerdere karakters (combinaties van letters en cijfers) zijn toegestaan.

We gaan de query uitbreiden: we willen nu ook nog voor iedere medewerker de naam van de afdeling erbij. Dat is informatie die uit de tabel AFDELINGEN moet worden opgehaald. We voegen daartoe een drietal

regels aan het commando toe (zie [figuur 8.5](#)).

Het principe is vrij eenvoudig. We hebben nu drie vrije tuple-variabelen ( $m$ ,  $s$  en  $a$ ) dus moeten we (minstens) twee voorwaarden specificeren 204

```

SQL> select m.naam          medewerker
  2 ,      m.maandsal+s.toelage totaalsalaris
  3 ,      a.naam            afdeling
  4 from  medewerkers m
  5 ,      schalen   s
  6 ,      afdelingen a
  7 where m.maandsal between s.ondergrens
        and s.bovengrens
  8 and   m.afd = a.anr;

MEDEWERKER    TOTAALSALARIS AFDELING
-----
SMIT           800 OPLEIDINGEN
JANSEN         800 VERKOOP
ADAMS          1100 OPLEIDINGEN
DE WAARD       1300 VERKOOP
MARTENS        1300 VERKOOP
MOLENAAR       1350 HOOFDKANTOOR
DEN DRAAIER    1600 VERKOOP
ALDERS          1700 VERKOOP
CLERCKX        2650 HOOFDKANTOOR
BLAAK           3050 VERKOOP
JANSEN          3175 OPLEIDINGEN
SCHOTTEN        3200 OPLEIDINGEN
SPIJKER          3200 OPLEIDINGEN
DE KONING       5500 HOOFDKANTOOR

14 rows selected.

SQL>

```

om de juiste rijcombinaties in het resultaat te krijgen.

### **Figuur 8.5**

Volledigheidshalve vermelden we hier dat in de taal SQL tabellenamen als default-tuple-variabele mogen worden verondersteld, dus zonder ze te declareren. Let op de eerste en de laatste regel:

**SQL> select medewerkers.naam, afdelingen.locatie**

**2 from medewerkers, afdelingen**

**3 where medewerkers.afd = afdelingen.anr**

We zullen van deze mogelijkheid in dit boek geen gebruik maken. Het is verwarrend om op het ene moment te spreken over een tabel, en op een ander moment over een rij uit die tabel zonder daarbij een duidelijk onderscheid in naamgeving te maken. Daarbij zijn de tabelnamen in dit boek zó lang dat het gebruik van éénletterige expliciete tuple-variabelen 205

```

SQL> select m.naam as medewerker
  2 ,      c.naam as chef
  3 from   medewerkers m
  4 ,      medewerkers c
  5 where  m.chef = c.mnr
  6 and    m.gbdatum > date '1965-01-01';

```

MEDEWERKER	CHEF
SMIT	SPIJKER
JANSEN	DE KONING
CLERCKX	DE KONING
DEN DRAAIER	BLAAK
ADAMS	SCHOTTEN
JANSEN	BLAAK

6 rows selected.

SQL>

al snel de moeite loont, door de winst op het aantal te plegen toetsaanslagen.

We kunnen in SQL een tabel ook joinen met zichzelf. Er is een aparte naam voor, hoewel er geen essentieel verschil is met een gewone join.

We noemen zo'n join een *self-join*. Overigens levert dit nog een extra argument op voor het gebruik van expliciete tuple-variabelen: hier zijn ze beslist noodzakelijk.

[Figuur 8.6](#) toont een voorbeeld van een self-join. We krijgen een overzicht van alle medewerkers die zijn geboren na 1 januari 1965, met daarbij de naam van hun chef in een tweede kolom. Dit soort resultaten laat zich eenvoudig controleren met behulp van het plaatje in bijlage A, waarin de hiërarchische structuur van de medewerkerstabel is weergegeven.

## Figuur 8.6

### 8.3

#### De ANSI/ISO standaard join syntax

De tot nu toe in dit hoofdstuk gegeven voorbeelden maken steeds gebruik van het Cartesiaans product (de komma in de FROM-component), waarop we vervolgens met behulp van de WHERE-component de

gewenste resultaatrijen selecteren. Daar is op zichzelf niets mis mee, en 206

```
SQL> select m.naam as medewerker
  2 ,      c.naam as chef
  3 from  medewerkers m
  4 JOIN
  5     medewerkers c
  6     ON m.chef = c.mnr
  7 where m.gbdatum > date '1965-01-01'
  8 order by medewerker;
```

MEDEWERKER	CHEF
ADAMS	SCHOTTEN
CLERCKX	DE KONING
DEN DRAAIER	BLAAK
JANSEN	BLAAK
JANSEN	DE KONING
SMIT	SPIJKER

6 rows selected.

```
SQL>
```

het is ook volledig conform de ANSI/ISO SQL-standaard, maar diezelfde standaard ondersteunt ook nog een andere manier om joins te produceren. Daarover gaat deze paragraaf.

In [figuur 8.6 hebben we bij](#) voorbeeld feitelijk met twee soorten condities te maken; regel 5 is de join-conditie die ervoor zorgt dat we niet de verkeerde verbanden leggen, en regel 6 is een ‘echte’conditie die filtert op de geboortedatum van de medewerkers.

Merk op dat we in deze syntax ([zie figuur 8.7\) geen komma](#) gebruiken in de FROM-component.

## Figuur 8.7

Dit ziet er in die zin beter uit, dat de join nu volledig in de FROM-component wordt gespecificeerd, terwijl we de WHERE-component alleen gebruiken voor de niet-join-condities.

We kunnen ook de natuurlijke join als operator gebruiken. Kijk maar eens

naar het voorbeeld [in figuur 8.8](#); hoe komt het dat we vijftien rijen in het resultaat zien?

207

```
SQL> select naam, beginjaar, maandsal, afd
  2  from medewerkers
  3      natural join
  4      historie;
```

NAAM	BEGINJAAR	MAANDSAL	AFD
SMIT	2000	800	20
ALDERS	1999	1600	30
DE WAARD	1992	1250	30
DE WAARD	2000	1250	30
JANSEN	1999	2975	20
MARTENS	1999	1250	30
BLAAK	1989	2850	30
CLERCKX	1988	2450	10
SCHOTTON	2000	3000	20
DE KONING	2000	5000	10
DEN DRAAIER	2000	1500	30
ADAMS	2000	1100	20
JANSEN	2000	800	30
SPIJKER	2000	3000	20
MOLENAAR	2000	1300	10

15 rows selected.

SQL>

- 
- 
- 

## Figuur 8.8

Blijkbaar komt iedere medewerker eenmaal voor, behalve DE WAARD. De natuurlijke join in [figuur 8.8 doet](#) namelijk het volgende: Bepaal welke kolommen de twee tabellen (MEDEWERKERS en

HISTORIE) gemeenschappelijk hebben. Dat zijn in dit geval MNR, MAANDSAL en AFD.

Join de twee tabellen met een equi-join over die gemeenschappelijke kolommen.

Onderdruk in het resultaat de dupicaatkolommen, die door deze werkwijze

zijn ontstaan (daarom krijgen we geen foutmelding dat MAANDSAL en/of AFD in [figuur 8.8](#) ambigu zijn gedefinieerd).

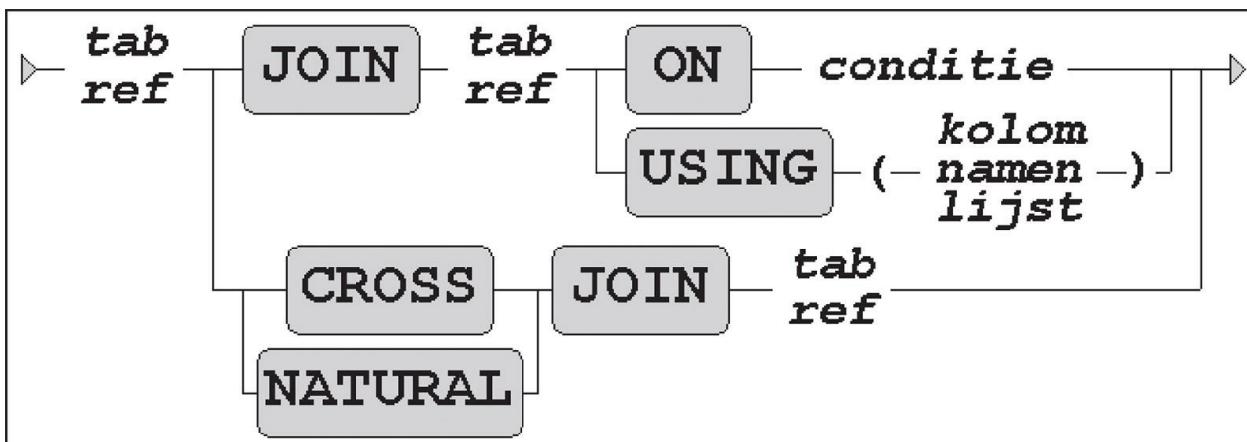
Blijkbaar heeft medewerker De Waard in zijn historie tweemaal voor dezelfde afdeling 30 gewerkt, met hetzelfde salaris (1250)...

Uit dit voorbeeld blijkt dat we erg voorzichtig moeten zijn met het gebruik van de natuurlijke join. Misschien wel het grootste gevaar schuilt in het feit dat een natuurlijke join plotseling ongewenste resultaten kan gaan geven als we aan tabellen kolommen toevoegen, die toevallig dezelfde naam hebben als een al bestaande kolom in een 208

```
SQL> select m.naam, m.gbdatum
  2 ,      h.afd, h.maandsal
  3 from  medewerkers m
  4 join
  5 historie h
  6 using (mnr)
  7 where m.functie = 'BOEKHOUDER';

NAAM      GBDATUM      AFD  MAANDSAL
-----  -----
JANSEN    03-DEC-69     30    800
MOLENAAR  23-JAN-62     10   1275
MOLENAAR  23-JAN-62     10   1280
MOLENAAR  23-JAN-62     10   1290
MOLENAAR  23-JAN-62     10   1300

SQL>
```



andere tabel.

Het is beter om expliciet aan te geven over welke kolommen we willen joinen; dat kan met behulp van de ON-component, maar als de kolomnamen gelijk zijn kunnen we ook de USING-component gebruiken.

Zie [figuur 8.9 voor een voorbeeld](#).

### **Figuur 8.9**

Merk op dat de tuple-variabelen nu weer noodzakelijk zijn, omdat we alleen maar joinen over MNR. De kolommen H.AFD en M.AFD zijn nu dus verschillend.

### **Figuur 8.10**

[Figuur 8.10 geeft](#) een overzicht van de ANSI/ISO join-209

```

SQL> select a.anr, a.locatie
2 ,      m.naam, m.voorl
3 from medewerkers m, afdelingen a
4 where m.afd = a.anr
5 order by a.anr, m.naam;

ANR LOCATIE          NAAM    VOORL
-----  -----
10 LEIDEN           CLERCKX   AB
10 LEIDEN           DE KONING CC
10 LEIDEN           MOLENAAR  TJA
20 DE MEERN         ADAMS     AA
20 DE MEERN         JANSEN    JM
20 DE MEERN         SCHOTTEN SCJ
20 DE MEERN         SMIT      N
20 DE MEERN         SPIJKER   MG
30 UTRECHT          ALDERS    JAM
30 UTRECHT          BLAAK     R
30 UTRECHT          DE WAARD  TF
30 UTRECHT          DEN DRAAIER JJ
30 UTRECHT          JANSEN    R
30 UTRECHT          MARTENS  P

14 rows selected.

SQL>

```

syntaxmogelijkheden weer. Merk op dat we er ook een CROSS JOIN

bestaat, die precies hetzelfde produceert als de komma in de FROM-component: het Cartesiaans product. In de rest van dit boek zullen we zo nu en dan de join-syntax uit [paragraaf 8.2 gebruiken](#), en soms de syntax uit deze paragraaf.

## 8.4

### De outerjoin

We hebben al eerder het volgende commando uitgevoerd (zie [figuur 8.3](#), enige bladzijden terug):

### Figuur 8.11

Wat in het resultaat opvalt is dat afdeling 40 ontbreekt. Dat komt doordat er aan afdeling 40 geen medewerkers verbonden zijn. Met andere woorden: als de tuple-variabele  $a$  afdeling 40 als waarde heeft, dan is er geen enkele rij  $m$  te vinden zodat aan de WHERE-conditie wordt voldaan.

Als we het feit dat die afdeling wel degelijk bestaat toch in het resultaat 210

- 
-

```

SQL> select a.anr, a.locatie
  2 ,      m.naam, m.voort
  3 from medewerkers m, afdelingen a
  4 where m.afd (+) = a.anr
  5 order by a.anr, m.naam;

ANR LOCATIE          NAAM        VOORT
----- -----          -----
 10 LEIDEN           CLERCKX     AB
 10 LEIDEN           DE KONING   CC
 10 LEIDEN           MOLENAAR    TJA
 20 DE MEERN         ADAMS       AA
 20 DE MEERN         JANSEN      JM
 20 DE MEERN         SCHOTTEN   SCJ
 20 DE MEERN         SMIT        N
 20 DE MEERN         SPIJKER    MG
 30 UTRECHT          ALDERS      JAM
 30 UTRECHT          BLAAK       R
 30 UTRECHT          DE WAARD    TF
 30 UTRECHT          DEN DRAAIER JJ
 30 UTRECHT          JANSEN      R
 30 UTRECHT          MARTENS    P
 40 GRONINGEN

15 rows selected.

SQL>

```

willen zien, kunnen we dat bereiken met een zogeheten *outerjoin*. Voor outerjoins hebben we in Oracle twee keuzes voor de syntax: De ‘oude’ syntax, die al lang bestond voordat de ANSI/ISO

standaard een outerjoin-syntax ondersteunde.

De ANSI/ISO standaard outerjoin-syntax.

We zullen van beide varianten een voorbeeld geven, gebaseerd op

### figuur 8.11.

Verander de vierde regel van het commando als volgt:

### **Figuur 8.12**

Nu komt afdeling 40 wél in het resultaat voor. Het effect van de toevoeging (+) in de WHERE-component heeft afdeling 40 gecombineerd met twee nullwaarden voor de medewerkersgegevens. Het bezwaar van deze syntax is dat het erg belangrijk is om de (+) operator op de juiste plaats(en) in het SQL-commando te gebruiken, en de leesbaarheid laat ook te wensen over. De ANSI/ISO outerjoin-syntax is veel leesbaarder; zie [figuur 8.13](#).

```

SQL> select a.anr, a.locatie
  2 ,      m.naam, m.voorl
  3 from medewerkers m
  4      RIGHT OUTER JOIN
  5      afdelingen a
  6      on m.afd = a.anr
  7 order by a.anr, m.naam;

```

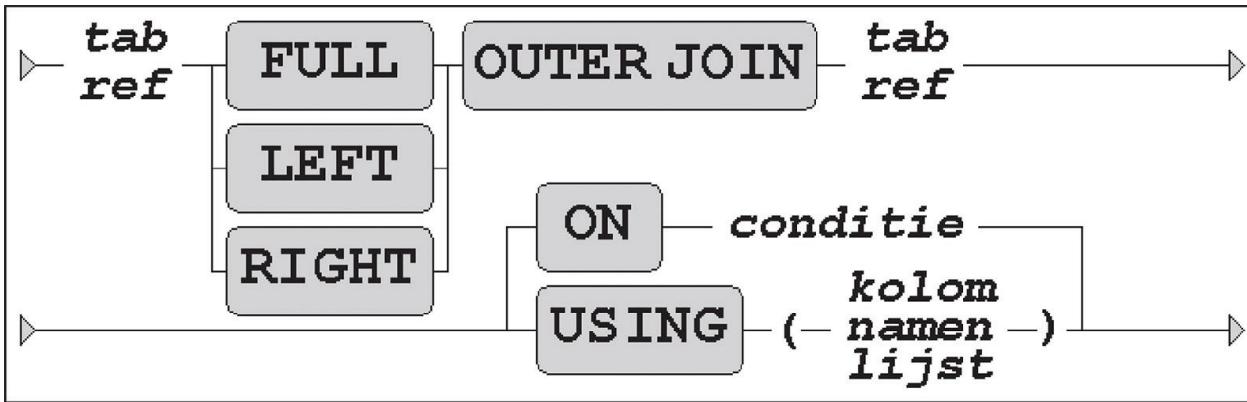
ANR	LOCATIE	NAAM	VOORL
10	LEIDEN	CLERCKX	AB
10	LEIDEN	DE KONING	CC
10	LEIDEN	MOLENAAR	TJA
20	DE MEERN	ADAMS	AA
20	DE MEERN	JANSEN	JM
20	DE MEERN	SCHOTTEN	SCJ
20	DE MEERN	SMIT	N
20	DE MEERN	SPIJKER	MG
30	UTRECHT	ALDERS	JAM
30	UTRECHT	BLAAK	R
30	UTRECHT	DE WAARD	TF
30	UTRECHT	DEN DRAAIER	JJ
30	UTRECHT	JANSEN	R
30	UTRECHT	MARTENS	P
40	GRONINGEN		

15 rows selected.

SQL>

In figuur 8.13 hebben we een RIGHT OUTER JOIN gebruikt, omdat we aan de rechterkant (de tabel AFDELINGEN) de aanwezigheid van rijen vermoeden die geen corresponderende rijen aan de linkerkant (de tabel MEDEWERKERS) hebben. Als de twee tabellen toevallig andersom in de FROM-component hadden gestaan, was de LEFT OUTER JOIN nodig geweest. Oracle ondersteunt ook een FULL OUTER JOIN, waarbij van beide kanten iets wordt gedaan met rijen die geen corresponderende rijen aan de andere kant blijken te hebben. Figuur 8.14 geeft de outerjoin-syntaxmogelijkheden aan.

## Figuur 8.13



```

SQL> select m.afd      as "afdeling"
  2 ,      count(m.mnr) as "aantal medewerkers"
  3 from medewerkers m
  4 group by m.afd;
  
```

afdeling	aantal medewerkers
10	3
20	5
30	6

```
SQL>
```

## Figuur 8.14

De outerjoin is vooral plezierig als we gegevens gaan aggregeren (indikken), bijvoorbeeld wanneer we een overzicht willen hebben van de aantalen medewerkers per afdeling, of het aantal inschrijvingen op geplande cursussen. We willen in een dergelijk overzicht expliciet zien dat een cursus wel gepland is, maar nog geen inschrijvingen heeft. Dit soort vragen gaan we in de volgende paragraaf behandelen.

## 8.5

### De GROUP BY-component

Tot nu toe hebben we uitsluitend queries gezien waarbij informatie werd verwacht met betrekking tot individuele rijen uit onze tabellen. Het komt echter regelmatig voor dat we geïnteresseerd zijn in geaggregeerde informatie. Hiermee wordt bedoeld: informatie die niet meer is gebaseerd op

afzonderlijke rijen, maar op verzamelingen van rijen.

Bijvoorbeeld: we willen een overzicht van het aantal medewerkers per afdeling. Bij dit soort queries hebben we de GROUP BY-component van het SELECT-commando nodig ([zie figuur 8.15](#)).

213

m.mnr	m.functie	m.chef	m.maandsal	m.comm	m.afd
[ 7782 , 7839 , 7934]	[ 'BOEKHOUDER' , 'DIRECTEUR' , 'MANAGER' ]	[ 7782 , 7839 , NULL ]	[ 1300 , 2450 , 5000 ]	[ NULL , NULL , NULL ]	10
[ 7369 , 7566 , 7788 , 7876 , 7902]	[ 'MANAGER' , 'TRAINER' , 'TRAINER' , 'TRAINER' , 'TRAINER' ]	[ 7566 , 7788 , 7839 , 7902 , 7566 ]	[ 800 , 1100 , 2975 , 3000 , 3000 ]	[ NULL , NULL , NULL , NULL , NULL ]	20
[ 7499 , 7521 , 7654 , 7698 , 7844 , 7900]	[ 'BOEKHOUDER' , 'MANAGER' , 'VERKOPER' , 'VERKOPER' , 'VERKOPER' , 'VERKOPER' ]	[ 7698 , 7839 , 7698 , 7698 , 7698 , 7698 ]	[ 950 , 1250 , 1500 , 1600 , 2850 , 1250 ]	[ 0 , 300 , 500 , 1400 , NULL , NULL ]	30

**Figuur 8.15**

Het resultaat van deze query is (uiteraard) een tabel. Er bestaat echter geen rij-naar-rij-koppeling meer van de medewerkerstabel naar de resultaattabel van deze query; de gegevens zijn geaggregeerd.

Er is gebruikgemaakt van de COUNT-functie om het aantal medewerkers per afdeling te tellen. Het is een voorbeeld van een groepsfunctie; in de volgende paragraaf komen we uitgebreid op dit soort functies terug.

Om het principe van groeperen te verduidelijken zullen we hier een stadium van de gegevens introduceren tussen de basistabel en de resultaattabel. Let wel: dit tussenstadium heeft uitsluitend een didactische functie; dergelijke structuren komen in werkelijkheid niet voor. Het effect van GROUP BY is

daarmee als [in figuur 8.16 voor te stellen](#).

Er is een ‘pseudo-tabel’ ontstaan met drie rijen. Omwille van de duidelijkheid zijn enkele kolommen van de tabel weggelaten. In de laatste kolom zien we de drie verschillende afdelingsnummers, en in de andere kolommen treffen we multisets van attribuutwaarden aan. Per pseudo-rij wordt door de functie COUNT(M.MNR) het aantal elementen in de multiset M.MNR geteld.

## Figuur 8.16

214

```
SQL> select i.cursus, i.begindatum
  2 , count(i.cursist)
  3 from inschrijvingen i
  4 group by i.cursus, i.begindatum;
```

CURSUS	BEGINDATUM	COUNT(I.CURSIST)
JAV	13-DEC-1999	5
JAV	01-FEB-2000	3
OAG	10-AUG-1999	3
OAG	27-SEP-2000	1
PLS	11-SEP-2000	3
S02	12-APR-1999	4
S02	04-OCT-1999	3
S02	13-DEC-1999	2
XML	03-FEB-2000	2

```
SQL>
```

Je zou ook kunnen zeggen dat de laatste kolom [in figuur 8.16 \(m.mnr\)](#) nog ‘gewone’ waarden bevat, en dat de andere vijf kolommen

‘verzamelingswaardig’ zijn geworden. In hoofdstuk 12 komen we nog terug op verzamelingswaardige attributen.

Op de mogelijkheden (en onmogelijkheden) van het gebruik van groepsfuncties komen we in de volgende paragraaf terug.

[We kunnen ook groeperen op meerdere kolommen. De query van figuur](#)

[8.17 geeft bijvoorbeeld een overzicht van het aantal inschrijvingen per cursus.](#)

### Figuur 8.17

8.6

#### Groepsfuncties

In de vorige paragraaf hebben we gebruikgemaakt van de COUNT-functie om het aantal medewerkers per afdeling te tellen. Dit is een voorbeeld van een *groepsfunctie*. Deze functies hebben als eigenschap dat ze worden losgelaten op een multiset van waarden, waarna ze één waarde als resultaat retourneren. Vandaar dat groepsfuncties vaak in combinatie met GROUP BY (en met HAVING, zie [paragraaf 8.7](#)) worden toegepast.

De Oracle-groepsfuncties zijn:

215

<b>COUNT( )</b>	Aantal waarden	alle datatypes
<b>SUM( )</b>	Som van de waarden	NUMBER
<b>MIN( )</b>	Minimumwaarde	alle datatypes
<b>MAX( )</b>	Maximumwaarde	alle datatypes
<b>AVG( )</b>	Gemiddelde waarde	NUMBER
<b>STDDEV( )</b>	Standaarddeviatie	NUMBER
<b>VARIANCE( )</b>	Variantie	NUMBER

```

SQL> select m.afd
  2 , count(m.functie)
  3 , sum(m.comm)
  4 , avg(m.maandsal)
  5 from medewerkers m
  6 group by m.afd;

      AFD COUNT(M.FUNCTIE)  SUM(M.COMM)  AVG(M.MAANDSAL)
-----  -----  -----
      10          3           2916.667
      20          5           2175
      30          6           1541.667

SQL>

```

## Figuur 8.18

In de laatste kolom is aangegeven op welk datatype de functies kunnen worden toegepast. Als de functies MIN en MAX op alfanumerieke gegevens worden losgelaten, werken ze net als bij het sorteren.

Bijvoorbeeld: ‘AAP’ < ‘NOOT’.

Conform de SQL-standaard dienen SQL-groepsfuncties null-waarden systematisch te negeren. Bekijk ter illustratie het resultaat van de query

[van figuur 8.19.](#)

## Figuur 8.19

In een multiset kunnen eventuele dupliaatwaarden voorkomen.

Duplicaatwaarden worden blijkbaar afzonderlijk meegerekend; er komen bijvoorbeeld maar twee verschillende functies voor op afdeling 20, terwijl er vijf medewerkers aan verbonden zijn.

Als we dupliaatwaarden willen negeren, moeten we tussen de haakjes vóór de kolomnaam DISTINCT toevoegen. Deze toevoeging heeft natuurlijk weinig zin bij de functies MIN en MAX. Een voorbeeld zien we 216

```

SQL> select count(afd), count(distinct afd)
  2 ,      avg(comm), avg(nvl(comm,0))
  3 from medewerkers;

COUNT(AFD) COUNT(DISTINCT AFD) AVG(COMM) AVG(NVL(COMM,0))
----- ----- -----
        14            3       550      157.1429

SQL>

```

[in figuur 8.20:](#)

## Figuur 8.20

In dit voorbeeld wordt tevens aangetoond dat groepsfuncties kunnen worden geselecteerd in een query zonder GROUP BY-component. Het effect hiervan is dat de hele tabel als één groep wordt beschouwd, en dat het resultaat dus zal bestaan uit één rij.

Bovendien zien we [in figuur 8.20 een toepassing van \*nesting\*](#): een functie toegepast op het resultaat van een andere functie. In dit soort situaties wordt de NVL-functie vaak gebruikt om null-waarden toch op de een of andere manier mee te nemen in het eindresultaat.

De volgende query ([zie figuur 8.21](#)) geeft antwoord op de vraag voor hoeveel verschillende cursussen een docent is ingeroosterd, en hoeveel cursusuitvoeringen hij inmiddels heeft verzorgd.

De query [in figuur 8.22 geeft](#) de gemiddelde evaluatie per docent, over alle cursussen die gegeven zijn. Let overigens op de voorwaarde die aan de begindatums wordt gesteld; die is beslist noodzakelijk om het correcte antwoord te krijgen.

```

SQL> select docent
  2 , count(distinct cursus)
  3 , count(*)
  4 from uitvoeringen
  5 group by docent;

DOCENT COUNT(DISTINCTCURSUS) COUNT(*)
-----
7369          2      3
7566          2      2
7788          2      2
7876          1      1
7902          2      2
                  3      3

```

SQL>

```

SQL> select u.docent, avg(i.evaluatie)
  2 from uitvoeringen u
  3 join
  4 inschrijvingen i
  5 using (cursus,begindatum)
  6 group by u.docent;

DOCENT AVG(I.EVALUATIE)
-----
7369          4
7566        4.25
7788
7876          4
7902          4

```

SQL>

## Figuur 8.21

## Figuur 8.22

Er is een uitzondering op de regel dat groepsfuncties opereren op een multiset van kolomwaarden. De COUNT-functie accepteert naast kolomnamen als argument ook een asterisk (\*), waardoor niet langer een aantal waarden wordt geteld; in plaats daarvan wordt het aantal rijen van de groep als geheel geteld. We zagen hiervan al een voorbeeld in

## figuur 8.21.

In [figuur 8.23](#) is de functie COUNT(\*) toegepast op de tabel MEDEWERKERS.

218

```
SQL> select m.afd, count(*)
  2  from medewerkers m
  3  group by m.afd;
```

AFD	COUNT (*)
10	3
20	5
30	6

```
SQL>
```

```
SQL> select a.anr, count(*)
  2  from medewerkers m
  3      right outer join
  4          afdelingen a
  5      on (m.afd = a.anr)
  6  group by a.anr;
```

ANR	COUNT (*)
10	3
20	5
30	6
40	1

```
SQL>
```

## Figuur 8.23

Als we de outerjoin toepassen om ook afdeling 40 weer te geven, moeten we goed opletten. Wat gaat er mis in [figuur 8.24?](#) Er blijkt plotseling toch iemand voor afdeling 40 te werken.

## Figuur 8.24

En waarom gaat het na de correctie in [figuur 8.25](#) wél goed?

219

```
SQL> select a.anr, count(m.mnr)
  2  from medewerkers m
  3      right outer join
  4          afdelingen a
  5      on (m.afd = a.anr)
  6 group by a.anr;
```

ANR COUNT (M.MNR)

ANR	COUNT (M.MNR)
10	3
20	5
30	6
40	0

SQL>

```
SQL> select afd, count(mnr)
  2  from medewerkers
  3 group by afd
  4 having count(*) >= 4;
```

AFD COUNT (MNR)

AFD	COUNT (MNR)
20	5
30	6

SQL>

## Figuur 8.25

Aan het [eind van hoofdstuk 4 zagen we een voorbeeld](#) van een met PL/SQL gebouwde opgeslagen functie om medewerkers te kunnen tellen per afdeling. Toen werd al opgemerkt dat een en ander in standaard-SQL niet zo eenvoudig was. Welnu, hier zien we dat we inderdaad moeten opletten; we hebben een outerjoin nodig, en we moeten ook oppassen welk argument we meegeven aan de COUNT-functie om het juiste resultaat te produceren.

8.7

## De HAVING-component

Met behulp van de HAVING-component kunnen we restricties op groepsniveau leggen. De HAVING-component volgt in SQL op een GROUP BY-constructie.

**Figuur 8.26**

220

```
SQL> select afd, count(mnr)
  2  from medewerkers
  3  where gbdatum > date '1960-01-01'
  4  group by afd
  5  having count(*) >= 4;

      AFD COUNT(MNR)
----- -----
      30          5

SQL>
```

```
SQL> select mnr
  2  from medewerkers
  3  where maandsal > avg(maandsal);
where maandsal > avg(maandsal)
*
ERROR at line 3:
ORA-00934: group function is not allowed here

SQL>
```

We krijgen nu alleen informatie over afdelingen waaraan minstens vier medewerkers zijn verbonden.

Om het verschil in werking tussen de WHERE-component en de HAVING-component duidelijk te maken voegen we een vijfde regel aan het commando toe ([zie figuur 8.27](#)).

**Figuur 8.27**

De voorwaarde met betrekking tot de geboortedatum is individueel per werknemer te controleren, terwijl de COUNT(\*)-voorwaarde alleen op groepsniveau zinvol is.

Vandaar dat groepsfuncties nooit in een WHERE-component zullen voorkomen, terwijl HAVING-constructies zonder groepsfunctie juist zeer zeldzaam zullen zijn. Let bijvoorbeeld op de foutmelding bij de volgende ‘klassieke’ fout [in figuur 8.28](#).

### Figuur 8.28

Deze query laat zich eigenlijk heel logisch lezen, nietwaar? Wie verdient er meer dan het gemiddelde salaris. Maar als we nu aan tuple-221

```
SQL> select m.mnr
  2  from medewerkers m
  3 where m.maandsal > (select avg(n.maandsal)
  4                           from medewerkers n );
      MNR
-----
 7566
 7698
 7782
 7788
 7839
 7902
SQL>
```

```

SQL> select    m.mnr
  2  from      medewerkers m
  3 ,        medewerkers n
  4 group by m.mnr
  5 ,        m.maandsal
  6 having   m.maandsal > avg(n.maandsal);

      MNR
-----
7566
7698
7782
7788
7839
7902

SQL>

```

variabelen denken, wordt duidelijk waar de schoen wringt: de WHERE-component heeft steeds één rij als context, waardoor de AVG-functie onzinnig is. Dit vraagstuk kan op vele manieren correct in SQL worden opgelost; zie bijvoorbeeld [figuur 8.29 en 8.30 voor een tweetal suggesties](#).

### **Figuur 8.29**

### **Figuur 8.30**

De laatste oplossing verdient misschien niet de schoonheidsprijs, maar is beslist de moeite van het bestuderen waard. Het is een oplossing die is gebaseerd op het Cartesiaans product. Let op het extra groeperen op M.MAANDSAL, wat nodig is om deze kolom in de HAVING-component te

```

SQL> select m.mnr, m.naam, count(*)
  2  from medewerkers m
  3    join
  4      inschrijvingen i
  5        on (m.mnr = i.cursist)
  6 group by m.mnr;
select m.mnr, m.naam, count(*)
*
ERROR at line 1:
ORA-00979: not a GROUP BY expression

SQL>

```

GROUP BY m.MNR			GROUP BY m.MNR,m.NAAM		
m.MNR	m.NAAM	...	m.MNR	m.NAAM	...
7369	['SMIT']		7369	'SMIT'	
7499	['ALDERS']		7499	'ALDERS'	
7521	...		7521	...	
...			...		

mogen aanspreken.

Het (ogenschijnlijk) overbodig groeperen op extra kolommen is wel eens vaker nodig. Stel dat we nummer én naam willen weten van iedere medewerker, met daarnaast het aantal inschrijvingen. Dan levert de

[query van figuur 8.31 een foutmelding op.](#)

### Figuur 8.31

Het pseudo-tussenstadium verklaart wat er fout gaat, en waarom er óók op M.NAAM gegroepeerd moet worden:

### Figuur 8.32

Er is namelijk een belangrijk verschil tussen een multiset die uit één element bestaat – zoals ['SMIT'] – en een constante, zoals 'SMIT'.

[Nog een SQL-commando met een leerzame fout erin zien we in figuur](#)

### 8.33.

223

```
SQL> select afd
  2 ,      sum(maandsal)
  3 from    medewerkers;
select afd
*
ERROR at line 1:
ORA-00937: not a single-group group function

SQL>
```

```
SQL> select    afd
  2 ,      sum(maandsal)
  3 from    medewerkers
  4 group by afd;
```

AFD	SUM(MAANDSAL)
10	8750
20	10875
30	9250

```
SQL>
```

### Figuur 8.33

De SUM-functie zou – bij gebrek aan een GROUP BY-component – één waarde opleveren, terwijl AFD veertien afdelingsnummers zou ophalen.

Twee kolommen van verschillende cardinaliteit laten zich echter niet samen presenteren tot één tabelresultaat. Met de volgende correctie is de query wél toegestaan ([zie figuur 8.34](#)).

### Figuur 8.34

We kunnen het voorgaande als volgt samenvatten:

Als een query een GROUP BY-component bevat, mogen in de SELECT-

component alleen maar groepsexpressies voorkomen. Een groepsexpressie is ofwel een kolomnaam waarop gegroepeerd wordt, ofwel een groepsfunctie, toegepast op een andere kolomexpressie.

## 8.8

### Extra mogelijkheden van de GROUP BY-component

We starten deze paragraaf met het voorbeeld [in figuur 8.35](#).

224

```
SQL> select afd, functie
  2 , count(mnr) headcount
  3 from medewerkers
  4 group by afd, functie;
```

AFD	FUNCTIE	HEADCOUNT
10	MANAGER	1
10	DIRECTEUR	1
10	BOEKHOUDER	1
20	MANAGER	1
20	TRAINER	4
30	MANAGER	1
30	VERKOPER	4
30	BOEKHOUDER	1

8 rows selected.

```
SQL>
```

### **Figuur 8.35**

We krijgen een overzicht per afdeling, en daarbinnen per functie, van het aantal medewerkers. Afdeling 40 (de afdeling zonder medewerkers) laten we nu even buiten beschouwing.

Let op wat er gebeurt als we de GROUP BY-component veranderen als in

[figuur 8.36:](#)



```
SQL> select afd, functie
  2 , count(mnr) headcount
  3 from medewerkers
  4 group by ROLLUP(afd, functie);
```

AFD FUNCTIE	HEADCOUNT
10 MANAGER	1
10 DIRECTEUR	1
10 BOEKHOUDER	1
<b>10</b>	<b>3</b>
20 MANAGER	1
20 TRAINER	4
<b>20</b>	<b>5</b>
30 MANAGER	1
30 VERKOPER	4
30 BOEKHOUDER	1
<b>30</b>	<b>6</b>
	<b>14</b>

```
12 rows selected.
```

```
SQL>
```

## Figuur 8.36

We krijgen nu ook nog een headcount per afdeling over alle functies, en aan het eind ook nog het totale aantal medewerkers. De nieuwe rijen zijn in [figuur 8.36 vet](#) weergegeven.

We kunnen in plaats van ROLLUP ook de CUBE-operator gebruiken (zie [figuur 8.37](#)).

```

SQL> select afd, functie
  2 , count(mnr) headcount
  3 from medewerkers
  4 group by CUBE(afd, functie);

```

AFD FUNCTIE	HEADCOUNT
	14
MANAGER	3
TRAINER	4
VERKOPER	4
DIRECTEUR	1
BOEKHOUDER	2
10	3
10 MANAGER	1
10 DIRECTEUR	1
10 BOEKHOUDER	1
20	5
20 MANAGER	1
20 TRAINER	4
30	6
30 MANAGER	1
30 VERKOPER	4
30 BOEKHOUDER	1

17 rows selected.

SQL>

## Figuur 8.37

We krijgen er nog eens vijf extra rijen bij (vet weergegeven in figuur 8.37) die de aantalen medewerkers per functie aangeven, ongeacht de afdeling waarvoor ze werken.

Door CUBE en ROLLUP worden nogal wat null-waarden gegenereerd in de extra rijen, zoals we in figuur 3.36 en figuur 3.37 kunnen zien. Om deze null-waarden te kunnen identificeren – bijvoorbeeld om ze te vervangen door een verklarende tekst – biedt Oracle de functies GROUPING en GROUPING\_ID. Zie [figuur 8.38 voor een voorbeeld](#) van GROUPING.

```

SQL> select afd
  2 ,      case grouping(functie)
  3 ,          when 0 then functie
  4 ,          when 1 then '**totaal**'
  5 ,      end      functie
  6 ,      count(mnr) headcount
  7 from medewerkers
  8 group by rollup(afd, functie);

```

AFD	FUNCTIE	HEADCOUNT
10	MANAGER	1
10	DIRECTEUR	1
10	BOEKHOUDER	1
10	**totaal**	3
20	MANAGER	1
20	TRAINER	4
20	**totaal**	5
30	MANAGER	1
30	VERKOPER	4
30	BOEKHOUDER	1
30	**totaal**	6
	**totaal**	14

12 rows selected.

SQL>

### Figuur 8.38

Jammer is dat de GROUPING-functie maar twee resultaten kan retourneren: 0 of 1. Daarom kunnen we geen afwijkende tekst aangeven voor de laatste regel. Dat kan wel met de GROUPING\_ID-functie; zie

[figuur 8.39.](#)

```

SQL> select    afd
  2 ,          case grouping_id(afd,functie)
  3             when 0 then functie
  4             when 1 then '** afd  **'
  5             when 3 then '**totaal**'
  6         end      functie
  7 ,          count(mnr)  headcount
  8 from      medewerkers
  9 group by rollup(afd, functie);

```

AFD FUNCTIE HEADCOUNT

AFD FUNCTIE	HEADCOUNT
10 MANAGER	1
10 DIRECTEUR	1
10 BOEKHOUDER	1
10 ** afd **	3
20 MANAGER	1
20 TRAINER	4
20 ** afd **	5
30 MANAGER	1
30 VERKOPER	4
30 BOEKHOUDER	1
30 ** afd **	6
**totaal**	14

12 rows selected.

SQL>

### Figuur 8.39

De waarde 3 waarop we testen in de vijfde regel [in figuur 8.39 komt](#) misschien wat vreemd over. Als we het getal 3 naar een binair getal omzetten wordt het 11. Daarmee hebben we een oplossing voor de situatie dat beide kolommen een null-waarde bevatten. Bij de ROLLUP

komen alleen 1 (binair 01) en 3 (binair 11) voor, maar bij CUBE komt ook 2 (binair 10) voor. Kijk [maar naar figuur 8.40.](#)

```

SQL> select    afd, functie
  2 ,          grouping_id(afd, functie) gid
  3 from      medewerkers
  4 group by CUBE(afd, functie);

```

AFD FUNCTIE	GID
	3
MANAGER	2
TRAINER	2
VERKOPER	2
DIRECTEUR	2
BOEKHOUDER	2
10	1
10 MANAGER	0
10 DIRECTEUR	0
10 BOEKHOUDER	0
20	1
20 MANAGER	0
20 TRAINER	0
30	1
30 MANAGER	0
30 VERKOPER	0
30 BOEKHOUDER	0

17 rows selected.

SQL>

## Figuur 8.40

Overigens zijn GROUP BY CUBE en GROUP BY ROLLUP eigenlijk twee bijzondere gevallen van de GROUP BY GROUPING SETS-syntax, die nog meer flexibiliteit biedt. We kunnen ook de resultaten van verschillende groeperingen in dezelfde GROUP BY-component samenvoegen door ze gescheiden door komma's achter elkaar te zetten. We gaan daar verder niet op in; zie de *SQL Reference* voor meer details.

Wat we tot slot van deze paragraaf nog wel onderzoeken zijn gepartitioneerde outerjoins. Zie [figuur 8.41 voor een voorbeeld](#) van een

'gewone' outerjoin:

```

SQL> break on afdeling skip 1 on functie

SQL> select a.naam as afdeling
  2 ,      m.functie
  3 ,      m.naam as medewerker
  4 from medewerkers m
  5      right outer join
  6      afdelingen a
  7      on a.anr = m.afd
  8 order by a.naam, m.functie;

AFDELING          FUNCTIE    MEDEWERKER
-----
HOOFDKANTOOR      BOEKHOUDER MOLENAAR
                  DIRECTEUR   DE KONING
                  MANAGER     CLERCKX

OPLEIDINGEN       MANAGER    JANSEN
                  TRAINER    SMIT
                  SPIJKER
                  SCHOTTEN
                  ADAMS

PERSONEELSZAKEN
VERKOOP           BOEKHOUDER JANSEN
                  MANAGER    BLAAK
                  VERKOPER   ALDERS
                  MARTENS
                  DEN DRAAIER
                  DE WAARD

15 rows selected.

SQL>

```

## Figuur 8.41

Met behulp van het commando BREAK (zie hoofdstuk 11 voor details) hebben we het resultaat van de query leesbaarder gemaakt. We krijgen vijftien rijen terug; de extra rij is voor personeelszaken, omdat die afdeling geen medewerkers heeft. Nu voegen we een extra clausule toe, vlak voor de operator RIGHT OUTER JOIN, en voeren de query nogmaals uit; zie figuur 8.42.

```

SQL> select a.naam as afdeling
  2 ,      m.functie
  3 ,      m.naam as medewerker
  4 from medewerkers m
  5      PARTITION BY (FUNCTIE)
  6      right outer join
  7      afdelingen a
  8      on a.anr = m.afd
  9 order by a.naam, m.functie;

```

AFDELING	FUNCTIE	MEDEWERKER
HOOFDKANTOOR	BOEKHOUDER	MOLENAAR
	DIRECTEUR	DE KONING
	MANAGER	CLERCKX
	TRAINER	
	VERKOPER	
OPLEIDINGEN	BOEKHOUDER	
	DIRECTEUR	
	MANAGER	JANSEN
	TRAINER	SMIT
		SPIJKER
		ADAMS
		SCHOTTON
	VERKOPER	
PERSONEELSZAKEN	BOEKHOUDER	
	DIRECTEUR	
	MANAGER	
	TRAINER	
	VERKOPER	
VERKOOP	BOEKHOUDER	JANSEN
	DIRECTEUR	
	MANAGER	BLAAK
	TRAINER	
	VERKOPER	ALDERS
		MARTENS
		DEN DRAAIER
		DE WAARD

26 rows selected.

SQL>

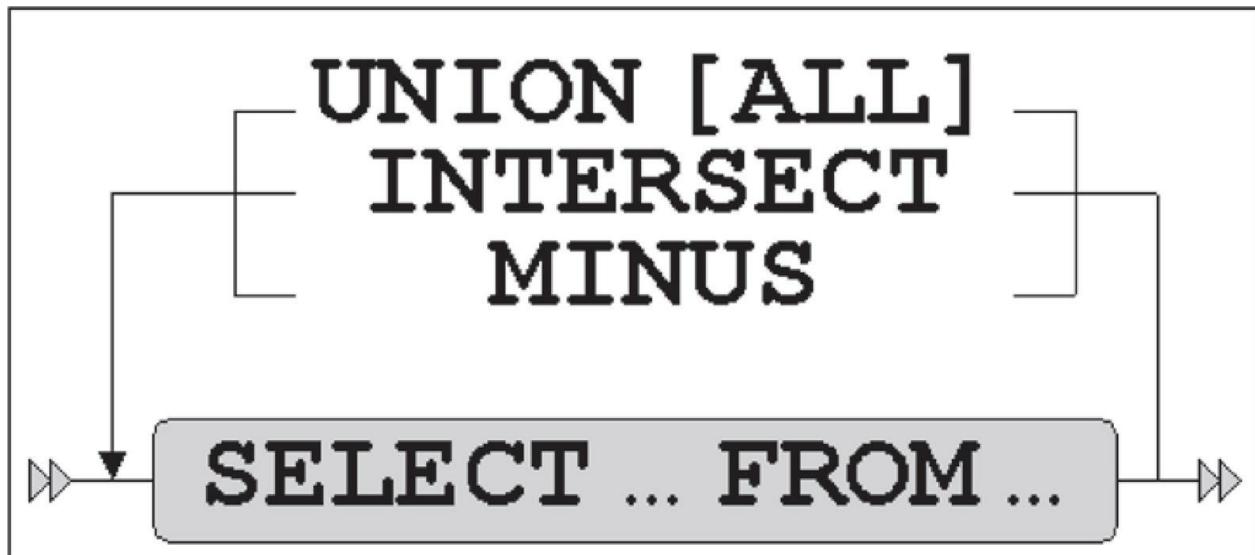
## Figuur 8.42

Nu krijgen we minstens een rij voor iedere functie, per afdeling.

Normaal gesproken wordt bij een ‘gewone’ outerjoin gekeken naar de hele tabel bij het zoeken naar corresponderende rijen aan de andere kant; de gepartitioneerde outerjoin werkt als volgt:

Verdeel een tabel in partities (deeltabellen) op basis van een bepaald criterium (in figuur 8.42 op basis van functie).

- 
- 



criterium ([in figuur 8.42 op basis](#) van functie).

Maak een aparte outerjoin voor elke partitie (deeltabel) met de andere tabel.

Voeg daarna de resultaten samen.

Dit is vooral handig bij aggregatie van gegevens over de tijd. Zie de *SQL Reference* voor meer details en voorbeelden.

8.9

Verzamelingsoperatoren

Met behulp van de verzamelingsoperatoren UNION, MINUS en INTERSECT kunnen we de resultaten van twee queries combineren tot één resultaat.

Ze onderscheiden zich daarmee van de andere SQL-operatoren in die zin dat ze niet op basistabellen of op views ([zie hoofdstuk 10 van de](#) database werken, maar op resultaattabellen die uit queries voortkomen.

In een eerder hoofdstuk zagen we al het volgende plaatje:

#### Figuur 8.43

Er is een duidelijke parallel met de vereniging, de verschilverzameling en de doorsnede uit de wiskunde. De betekenis van deze operatoren in SQL is als volgt:

233

<b>Q1 UNION [ALL] Q2</b>	Alle rijen die in het resultaat van Q1 of van Q2 voorkomen
<b>Q1 MINUS Q2</b>	De rijen van het resultaat van Q1, met uitsluiting van de rijen die óók in het resultaat van Q2 voorkomen
<b>Q1 INTERSECT Q2</b>	De rijen die zowel in het resultaat van Q1 als in het resultaat van Q2 voorkomen

- 
- 
- 
- 
- 

#### Figuur 8.44

Alle drie deze operatoren zullen eventuele dupicaatrijen in hun resultaat achterwege laten – behalve de variant UNION ALL, die dit juist niet doet.

Een voordeel daarvan is dat er geen (dure) sorteerslag hoeft te worden

gemaakt; die wordt normaal gesproken altijd uitgevoerd om dupicaatrijen op te kunnen sporen.

De operatoren UNION, MINUS en INTERSECT stellen enkele eisen aan de queries waarop ze worden toegepast. De twee tussenresultaten van Q1

en Q2 moeten namelijk met elkaar te combineren zijn tot één resultaattabel (ze moeten ‘compatibel’ zijn) hetgeen leidt tot de volgende voor de hand liggende eisen:

Q1 en Q2 moeten evenveel kolommen (of expressies) selecteren.

De datatypes van de geselecteerde expressies moeten overeenstemmen.

Verder zijn er nog enkele ‘kleinigheden’:

De resultaattabel erft de kolomnamen/aliassen van Q1.

Q1 mag geen ORDER BY-component bevatten.

Een eventuele ORDER BY na Q2 geldt voor het eindresultaat van de verzamelingsoperator.

Deze verzamelingsoperatoren worden vaak toegepast om bestaande queries met elkaar te combineren, zodat er geen compleet nieuwe queries hoeven te worden geformuleerd. Dit vereenvoudigt het testen, omdat er meer controle mogelijk is op de correctheid.

In welke plaats vinden wél cursussen plaats, maar zijn géén afdelingen gevestigd?

```
SQL> select u.locatie from uitvoeringen u
  2  MINUS
  3  select a.locatie from afdelingen a;
```

LOCATIE

-----  
MAASTRICHT

```
SQL>
```

```
SQL> select DISTINCT u.locatie
  2  from  uitvoeringen u
  3  where u.locatie not in
  4      (select a.locatie
  5       from   afdelingen a)
```

## Figuur 8.45

Hetzelfde probleem kan ook zonder de MINUS-operator worden opgelost (zie [figuur 8.46](#)). Merk overigens op dat we nu DISTINCT moeten toevoegen; zoals gezegd verwijdert de MINUS-operator automatisch dupliaatrijen.

## Figuur 8.46

Zijn de twee queries [in figuur 8.45 en 8.46 wel](#) equivalent? Een nader onderzoek wordt als opgave (aan het eind van dit hoofdstuk) aan de lezer overgelaten.

Een outerjoin kan ook worden geproduceerd met behulp van de UNION-operator, zoals al in een van de eerdere paragrafen was aangekondigd.

We zullen hier laten zien hoe dat kan.

We beginnen eerst met een gewone join [in figuur 8.47](#); dan voegen we [in figuur 8.48 met](#) behulp van de UNION-constructie aan het resultaat van de join precies de gewenste extra rijen toe, en wel met het juiste aantal werknemers, namelijk 0 (nul).

```
SQL> select    a.anr      as afdeling
  2 ,        count(m.mnr) as bezetting
  3 from      medewerkers m
  4 ,        afdelingen a
  5 where     m.afd = a.anr
  6 group by a.anr;
```

AFDELING BEZETTING

AFDELING	BEZETTING
10	3
20	5
30	6

SQL>

```

SQL> select    a.anr      as afdeling
  2 ,          count(m.mnr) as bezetting
  3 from      medewerkers m
  4 ,          afdelingen a
  5 where     m.afd = a.anr
  6 group by a.anr
  7 UNION
  8 select    x.anr
  9 ,          0
 10 from     afdelingen x
 11 where    x.anr not in (select y.afd
                           from   medewerkers y);
 12

AFDELING BEZETTING
-----
 10      3
 20      5
 30      6
 40      0

SQL>

```

**Figuur 8.47**

**Figuur 8.48**

Merk overigens op dat de join in de eerste [query \(figuur 8.48, regels 1-6\)](#) eigenlijk niet nodig is; afdelingen zonder medewerkers halen we er namelijk toch al bij met behulp van de regels 8-12.

## 8.10 Opgaven

1

Produceer een overzicht van alle cursusuitvoeringen; geef de 236 cursuscode, de begindatum, de cursuslengte en de naam van de docent.

2

Geef in twee kolommen naast elkaar de naam van elke cursist die een S02-cursus heeft gevolgd, met de naam van de docent.

3

Geef van iedere medewerker: naam, voorletters en het jaarsalaris (inclusief toelage en commissie).

4

Geef van alle cursusuitvoeringen: de cursuscode, de begindatum en het aantal inschrijvingen. Sorteer op begindatum.

5

Geef nu code, begindatum en aantal inschrijvingen van alle cursusuitvoeringen in 1999 met minstens drie inschrijvingen.

6

Geef de nummers van alle medewerkers die wél ooit een cursus als docent hebben gegeven, maar nog nooit een cursus hebben gevolgd.

7

Welke medewerkers hebben een bepaalde cursus meer dan één keer gevolgd?

8

Geef van alle docenten: naam en voorletters, het aantal cursussen dat ze hebben gegeven, het totale aantal cursisten dat ze hebben opgeleid, en het gemiddelde evaluatiecijfer. Rond deze berekening af op één decimaal.

9

Geef naam en voorletters van alle trainers die ooit tijdens een algemene cursus (type ALG) hun eigen chef als cursist hebben gehad.

10 Hebben we op een van de cursuslocaties op enig moment twee lokalen tegelijkertijd in gebruik gehad?

11 Geef een matrixoverzicht (voor elke afdeling een kolom, voor elke functie

een rij) met in iedere cel het aantal medewerkers. In een query is het dynamisch bepalen van het aantal kolommen

ondoenlijk; ga daarom uit van de afdelingsnummers 10, 20 en 30.

12 Zijn de twee queries [in figuur 8.45 en 8.46 equivalent](#)? Onderzoek de queries nader, en verklaar het eventuele verschil.

237

## **Hoofdstuk 9**

### **Raadpleging – enkele geavanceerde**

#### **mogelijkheden**

Dit is het vierde hoofdstuk dat is gewijd aan de

raadpleegmogelijkheden van SQL; het sluit in die zin aan op de hoofdstukken 4, 5 en 8.

Allereerst komen we terug op subqueries: [in paragraaf 9.1 worden](#) de drie operatoren ANY, ALL en EXISTS geïntroduceerd. In diezelfde paragraaf komen ook gecorreleerde subqueries aan de orde.

Daarna gaan we subqueries toepassen buiten de WHERE-

component: in de SELECT- en in de FROM-component.

[In paragraaf 9.4 komt](#) de WITH-component aan de orde, waarmee we aan het begin van een SQL-commando een of meer subqueries kunnen definiëren, een naam kunnen geven, en vervolgens aan de subqueries kunnen refereren met hun naam. Dit maakt de SQL-commando's in de eerste plaats leesbaarder en daardoor ook gemakkelijker te onderhouden; het kan in bepaalde gevallen ook nog resulteren in betere responstijden.

Vervolgens komen hiërarchische queries aan bod. Tabellen zijn in principe ‘platte’ structuren, maar ze kunnen hiërarchische structuren representeren door bijvoorbeeld gebruik te maken van refererende sleutels die verwijzen

naar de primaire sleutel van dezelfde tabel. Een klassiek voorbeeld hiervan is de kolom chef van de tabel MEDEWERKERS. Middels *recursive subquery factoring* biedt SQL specifieke syntax om het raadplegen van hiërarchische structuren mogelijk te maken. In [paragraaf 9.6](#) worden vensters geïntroduceerd, en de bijbehorende analytische functies.

Ten slotte zullen we een aardige ‘feature’ van Oracle SQL bekijken waarmee we kunnen reizen in de tijd: dit is mogelijk met de zogeheten ‘flashback’-queries. Het hoofdstuk wordt afgesloten met opgaven.

## 9.1

### Subqueries: vervolg

238

```

SQL> select i.cursist, i.cursus, i.begindatum
  2  from inschrijvingen i
  3 where i.cursus in (select c.code
  4                      from cursussen c
  5                      where c.type='BLD');

```

CURSIST CURS BEGINDATUM

CURSIST	CURS	BEGINDATUM
7499	JAV	13-DEC-1999
7566	JAV	01-FEB-2000
7698	JAV	01-FEB-2000
7788	JAV	13-DEC-1999
7839	JAV	13-DEC-1999
7876	JAV	13-DEC-1999
7788	JAV	01-FEB-2000
7782	JAV	13-DEC-1999
7499	PLS	11-SEP-2000
7876	PLS	11-SEP-2000
7566	PLS	11-SEP-2000
7499	XML	03-FEB-2000
7900	XML	03-FEB-2000

SQL>

```

SQL> select m.mnr, m.naam, m.voortl, m.gbdatum
  2  from medewerkers m
  3 where m.gbdatum > (select n.gbdatum
  4                      from medewerkers n
  5                      where n.mnr = 7566);

```

MNR NAAM VOORTL GBDATUM

7844	DEN DRAAIER	JJ	28-SEP-1968
7900	JANSEN	R	03-DEC-1969

SQL>

We hebben [in hoofdstuk 4 al](#) diverse voorbeelden van subqueries gezien. Ter opfrissing van het geheugen halen we eerst nog een paar voorbeelden terug.

## Figuur 9.1

De subquery-constructie [in figuur 9.1 geeft](#) alle inschrijvingen (13 stuks) voor cursussen van het type ‘BLD’. Medewerkers die jonger zijn dan

collega 7566 kunnen we als volgt ophalen ([zie figuur 9.2](#)).

## Figuur 9.2

239

```
SQL> select m.mnr, m.naam, m.functie, m.maandsal
  2  from medewerkers m
  3 where m.maandsal > ANY (select n.maandsal
  4                           from medewerkers n
  5                           where n.functie='MANAGER');

      MNR    NAAM      FUNCTIE   MAANDSAL
-----  -----  -----
    7839 DE KONING    DIRECTEUR     5000
    7788 SCHOTTEN     TRAINER      3000
    7902 SPIJKER      TRAINER      3000
    7566 JANSEN       MANAGER      2975
    7698 BLAAK        MANAGER      2850

SQL>
```

In dit voorbeeld dient er sprake te zijn van een *single row subquery*, omdat anders de vergelijkingsoperator (>) de mist in gaat. Als in dit soort constructies de subquery toch meer dan één rij retourneert, resulteert dat in een foutmelding, zoals we [in hoofdstuk 4 al](#)

proefondervindelijk hebben vastgesteld.

In deze paragraaf over subqueries worden de mogelijkheden van de operatoren ANY, ALL en EXISTS besproken, en zullen we het begrip *gecorreleerde subquery* introduceren.

Het is in SQL mogelijk om een correct verband te leggen tussen een vergelijkingsoperator (<, >, =, enzovoort) en een subquery die meerdere rijen oplevert. We maken daartoe gebruik van één van de operatoren ANY of ALL. Deze operatoren moeten tussen de

vergelijkingsoperator en de subquery worden geplaatst.

### Figuur 9.3

In figuur 9.3 krijgen we alle medewerkers te zien die per maand meer verdienen dan minstens één van de managers; in figuur 9.4 zijn het de gelukkigen die zelfs meer verdienen dan alle managers.

240

```
SQL> select m.mnr, m.naam, m.functie, m.maandsal
  2  from medewerkers m
  3 where m.maandsal > ALL (select n.maandsal
  4                           from medewerkers n
  5                           where n.functie='MANAGER');

      MNR    NAAM      FUNCTIE   MAANDSAL
-----  -----  -----
    7788 SCHOTTEN    TRAINER     3000
    7839 DE KONING   DIRECTEUR   5000
    7902 SPIJKER     TRAINER     3000

SQL>
```

### Figuur 9.4

Deze operatoren hebben dus de volgende betekenis:

#### ANY

... waarvoor geldt dat ... voor minstens één ...

#### ALL

... waarvoor geldt dat ... voor alle ...

We kunnen dit ook iets formeler onder woorden brengen. Stel dat we met # een willekeurige vergelijkingsoperator ( $<$ ,  $>$ ,  $=$ ,  $\geq$ ,  $\leq$ ,  $\neq$ ) bedoelen. Neem bovendien aan dat we de waarden die door de subquery worden gereturneerd, aangeven met  $w_1$ ,  $w_2$ ,  $w_3$ , enzovoort. Dan kunnen we de werking van ANY en ALL ook als volgt weergeven:  $X \# \text{ANY}(\text{subquery})$

X # ALL(subquery)

(X # w1) OR

(X # w1) AND

(X # w2) OR

(X # w2) AND

(X # w3) OR ...

(X # w3) AND ...

Het is meestal wel mogelijk om een query in SQL zodanig te herformuleren dat ANY en ALL niet nodig zijn. We hadden [in figuur 9.4](#)

bijvoorbeeld ook een groepsfunctie kunnen toepassen om de subquery om te bouwen naar een single row subquery ([zie figuur 9.5](#)).

241

```
SQL> select m.naam, m.functie, m.maandsal
  2  from medewerkers m
  3 where m.maandsal > (select max(n.maandsal)
  4                           from medewerkers n
  5                           where n.functie='MANAGER');


```

NAAM	FUNCTIE	MAANDSAL
SCHOTTEN	TRAINER	3000
DE KONING	DIRECTEUR	5000
SPIJKER	TRAINER	3000

```
SQL>
```

## Figuur 9.5

Merk op dat de volgende constructies equivalent zijn:

X = ANY(subquery) <=> X IN (subquery)

$X <> \text{ALL}(\text{subquery}) \Leftrightarrow X \text{ NOT IN } (\text{subquery})$

Nog een tweetal bijzondere gevallen van deze twee operatoren:  $X = \text{ALL}(\text{subquery})$

$X <> \text{ANY}(\text{subquery})$

Als de subquery twee of meer verschillende waarden oplevert, dan is (ga maar na!) de eerste expressie *altijd onwaar*, en de tweede *altijd waar*.

In SQL zijn ook *gecorreleerde subqueries* mogelijk. Laten we meteen maar naar het voorbeeld [van figuur 9.6 kijken](#), dan zien we vanzelf waarom deze subqueries zo heten.

242

```
SQL> select m.naam, m.voorl, m.maandsal
  2  from medewerkers m
  3 where m.maandsal > (select avg(n.maandsal)
  4                           from medewerkers n
  5                           where n.afd = m.afd);
```

NAAM	VOORL	MAANDSAL
ALDERS	JAM	1600
JANSEN	JM	2975
BLAAK	R	2850
SCHOTTEN	SCJ	3000
DE KONING	CC	5000
SPIJKER	MG	3000

```
SQL>
```

## Figuur 9.6

Deze query geeft alle medewerkers die qua maandsalaris boven het gemiddelde van hun eigen afdeling zitten. Wat deze subquery bijzonder maakt is het feit dat er een verwijzing in voorkomt naar de tupelvariabele *m* van de hoofdquery.

We kunnen deze subquery dus niet zelfstandig (in isolement) uitvoeren; we moeten hem altijd zien in de context van een bepaalde rij uit de hoofdquery. De query [in figuur 9.6 wordt](#) dan ook als volgt verwerkt: De tupelvariable *m* loopt over de tabel MEDEWERKERS.

Voor iedere rij *m* wordt de subquery uitgevoerd, nadat voor M.AFD de huidige waarde is gesubstitueerd.

In de wiskunde spreken we in dit verband over *vrije* en *gebonden* variabelen; [in de subquery van figuur 9.6 is n](#) de vrije variabele, terwijl *m* gebonden is door de hoofdquery.

We bekijken nog een voorbeeld ([zie figuur 9.7](#)). [Deze query geeft](#) de op drie na jongste medewerker van het bedrijf, of, om iets nauwkeuriger te zijn: alle medewerkers waarvoor geldt dat er precies drie medewerkers jonger zijn (het resultaat kan namelijk uit meer dan één medewerker bestaan).

243

```
SQL> select m.*  
  2  from  medewerkers m  
  3  where (select count(*)  
  4      from  medewerkers n  
  5      where  n.gbdatum > m.gbdatum) = 3;  
  
MNR NAAM      VOORL FUNCTIE   CHEF GBDATUM      MAANDSAL    COMM     AFD  
----- -----  -----  -----  
7876 ADAMS      AA    TRAINER    7788 30-DEC-1966      1100          20  
  
SQL>
```

```

SQL> select u.*
  2  from uitvoeringen u
  3 where not exists
  4     (select i.*
  5      from inschrijvingen i
  6     where i.cursus      = u.cursus
  7     and   i.begindatum = u.begindatum);

CURS BEGINDATUM      DOCENT LOCATIE
----- -----
ERM  15-JAN-2001
PRO  19-FEB-2001      DE MEERN
RSO  24-FEB-2001      7788 UTRECHT
XML  18-SEP-2000      MAASTRICHT

SQL>

```

## Figuur 9.7

Dit soort queries kunnen we ook heel goed formuleren met behulp van vensters en analytische functies, die in een latere paragraaf van dit hoofdstuk nog aan bod zullen komen.

Gecorreleerde subqueries worden vaak gebruikt in combinatie met de EXISTS-operator. Laten we meteen maar weer een voorbeeld bekijken ([zie figuur 9.8](#)).

## Figuur 9.8

Dit geeft als resultaat alle cursusuitvoeringen waarvoor géén inschrijvingen bestaan. De EXISTS-operator is niet geïnteresseerd in rijen of kolomwaarden die uit een subquery komen; het gaat erom óf er rijen uit de subquery komen. Is dit het geval, dan levert de EXISTS

‘waar’ op; levert de subquery géén rijen op, dan is het resultaat ‘onwaar’. EXISTS is dus een soort ‘lege-verzameling-checker’.

```

SQL> select m.*
  2  from medewerkers m
  3 where exists (select u.*
  4                   from uitvoeringen u
  5                   where u.cursus = 'S02'
  6                   and u.docent = m.mnr);

MNR NAAM      VOORL FUNCTIE     CHEF GBDATUM      MAANDSAL COMM AFD
----- ----- -----
7369 SMIT      N      TRAINER    7902 17-DEC-1965      800   20
7902 SPIJKER   MG     TRAINER    7566 13-FEB-1959     3000   20

SQL>

```

Het maakt dan ook helemaal niets uit wat we in de SELECT-component van de subquery selecteren; vaak wordt daarom een willekeurige constante gebruikt. We hadden de query [in figuur 9.8 dus ook als](#) volgt kunnen formuleren:

**SQL> select u.\***

**2 from uitvoeringen u**

**3 where not exists**

**4 (select ‘x’**

**5 from ...**

Subqueries die volgen op de EXISTS-operator zijn vaak gecorreleerd.

Als ze dat namelijk niet zijn, is het resultaat van de hoofdquery ‘alles of niets’; de subquery levert dan voor iedere rij van de hoofdquery hetzelfde resultaat.

Pas op met de EXISTS-operator: een subquery die een null-waarde retourneert (één rij dus), is niet hetzelfde als een subquery die niets retourneert!

Nog een belangrijke opmerking: het resultaat van de EXISTS-operator is

altijd ‘waar’ of ‘onwaar’, en nooit ‘onbeslist’. Vooral in combinatie met nullwaarden en ontkenningen kan dit soms leiden tot verrassende resultaten.

Nog een voorbeeld tot slot. De vraag is ([zie figuur 9.9](#)): geef de personalia van alle medewerkers die ooit een S02-cursus hebben gegeven.

245

```
SQL> select m.*  
2   from medewerkers m  
3  where m.mnr in (select u.docent  
4                      from uitvoeringen u  
5                     where u.cursus = 'S02')
```

```
SQL> select DISTINCT m.*  
2   from medewerkers m  
3     join  
4       uitvoeringen u  
5      on m.mnr = u.docent  
6  where u.cursus = 'S02'
```

## Figuur 9.9

Dit probleem had ook als volgt opgelost kunnen worden (we laten de resultaten in [figuur 9.10](#) niet opnieuw zien): **Figuur 9.10**

Ook een joinconstructie zou hier kunnen worden toegepast – wellicht de meest voor de hand liggende oplossing. Let op: we moeten nu wel DISTINCT aan de SELECT-component toevoegen.

## Figuur 9.11

Onderzoek zelf wat er zou gebeuren als we [figuur 9.11](#) DISTINCT weglaten; het resultaat bestaat dan niet meer uit twee maar uit drie rijen.

9.2

Subqueries in de SELECT-component

Tot nu toe hebben we alleen subqueries bestudeerd in de WHERE-component. We mogen subqueries in SQL echter ook toepassen in andere componenten, zoals bijvoorbeeld de SELECT-component.

Kijk nog eens naar de figuren 5.29 en 5.30. De SQL-standaard biedt voor dat probleem een elegant alternatief met behulp van een subquery in de SELECT-component: zie [figuur 9.12](#).

246

```
SQL> select a.anr, a.naam, a.locatie
  2 ,      (select count(*)
  3       from medewerkers m
  4     where m.afd = a.anr) as m_aantal
  5 from afdelingen a;

ANR NAAM          LOCATIE      M_AANTAL
-----  -----
 10 HOOFDKANTOOR    LEIDEN        3
 20 OPLEIDINGEN    DE MEERN      5
 30 VERKOOP         UTRECHT       6
 40 PERSONEELSZAKEN GRONINGEN    0

SQL>
```

## Figuur 9.12

Eigenlijk is dit niet alleen een correcte maar ook een elegante oplossing.

Elegant, omdat we in deze query feitelijk geïnteresseerd zijn in gegevens over afdelingen. De eerste drie attributen (NUMMER, NAAM en LOCATIE) zijn ‘gewone’ attributen die als kolomwaarden in de tabel AFDELINGEN voorkomen; het vierde attribuut (het aantal medewerkers) is niet in de database opgeslagen, maar kan per afdeling worden afgeleid door de MEDEWERKERS-tabel te raadplegen. Zo laat zich deze query ook precies lezen; in de FROM-component bezoeken we de tabel AFDELINGEN, en in de SELECT-component selecteren we vier expressies. Geen OUTER

JOIN, zelfs geen gewone join, geen GROUP BY, en toch het juiste aantal medewerkers (0) voor de afdeling personeelszaken.

We zouden kunnen zeggen dat de GROUP BY-component bijna overbodig is geworden; vrijwel alle aggregatieproblemen kunnen we ook oplossen met behulp van een subquery in de SELECT-component, zonder gebruik te maken van GROUP BY.

Merk overigens op dat de subquery [in figuur 9.12 gecorreleerd](#) is; voor iedere rij  $a$  van de AFDELINGEN-tabel heeft A.ANR een andere waarde, en met die waarde wordt de subquery vier keer uitgevoerd. Het ligt ook nogal voor de hand om de subquery-expressie van een kolom-alias (M\_AANTAL [in figuur 9.12](#)) te voorzien; dit verhoogt de leesbaarheid van zowel de query als de resultaten.

In het Engels worden dit soort subqueries ook wel *scalar subquery expressions* genoemd. Deze naam geeft een belangrijke eigenschap van dit soort subqueries weer: het resultaat bestaat uit precies één rij, die 247

- 
- 
-

```

SQL> select m.naam, m.voort, m.maandsal
  2  from medewerkers m
  3      join
  4      (select x.afd
  5          , avg(x.maandsal) g_sal
  6      from medewerkers x
  7      group by x.afd
  8      ) g
  9  where m.maandsal > g.g_sal;

```

NAAM	VOORT	MAANDSAL
ALDERS	JAM	1600
JANSEN	JM	2975
BLAAK	R	2850
SCHOTTEN	SCJ	3000
DE KONING	CC	5000
SPIJKER	MG	3000

SQL>

precies één kolom bevat. Dit soort subqueries mogen we eigenlijk overal in SQL-commando's gebruiken, op plaatsen waar ook het gebruik van een constante is toegestaan. De subquery leidt als het ware de constante af.

Je zou dus kunnen zeggen dat SQL de volgende subquery-hiërarchie ondersteunt:

*Multi-row subqueries*: geen restricties met betrekking tot de resultaten.

*Single-row subqueries*: resultaat bestaat uit precies één rij.

*Scalar subqueries*: resultaat bestaat uit een constante (één rij, één kolom).

## 9.3

### Subqueries in de from-component

Een voorbeeld van een subquery in de FROM-component zien we in

[figuur 9.13. Dit](#) soort subqueries worden ook wel *inline views* genoemd.

Je zou kunnen zeggen dat we in plaats van een tabelnaam in de FROM-component een subquery gebruiken, waarvan de resultaten als een soort

‘afgeleide tabel’ in de hoofdquery worden gebruikt. De benaming inline view zal overigens vanzelf duidelijker worden als we views behandelen [in hoofdstuk 10.](#)

248

```
SQL> WITH g AS
  2      (select    x.afd
  3       ,        avg(x.maandsal) g_sal
  4       from      medewerkers x
  5       group by x.afd
  6     )
 7 select m.naam, m.voorl, m.maandsal
 8 from   medewerkers m
 9      join
10      g
11      using (afd)
12 where  m.maandsal > g.g_sal
```

### Figuur 9.13

Door over de subquery (zie regel 6 [in figuur 9.13](#)) de tupelvariable *g* te definiëren kunnen we aan resultaatkolommen van de inline view refereren, zoals bijvoorbeeld op de laatste regel (*G.G\_SAL*) gebeurt. De query [in figuur 9.13](#) is overigens een alternatief voor de query van

[figuur 9.6.](#)

Tot nu toe hebben we in dit hoofdstuk de subquery-mogelijkheden uitgebreid met de operatoren ANY, ALL en EXISTS. Bovendien hebben we kennisgemaakt met *gecorreleerde subqueries* en subqueries op minder voor de hand liggende plaatsen.

9.4

De WITH-component

Als we nog even terugkijken naar het laatste voorbeeld ([zie figuur 9.13](#)), dan hadden we ook de [syntax van figuur 9.14 kunnen gebruiken](#): **Figuur 9.14**

Nu is de subquery (regels 1-5) als het ware geïsoleerd van de hoofdquery (regels 7-12). Hierdoor wordt de structuur van de hoofdquery overzichtelijker. Deze constructie wordt vooral aantrekkelijk als we in de hoofdquery meerdere keren aan de subquery refereren. We kunnen desgewenst meerdere subqueries in de WITH-component definiëren, onderling gescheiden door komma's.

De Oracle optimizer heeft (achter de schermen) twee manieren om dit te doen:

- 
- 

soort queries aan te pakken:

De subquery-definities worden gesubstitueerd in de hoofdquery, waardoor ze zich gedragen als inline views.

De subqueries worden uitgevoerd, de resultaten worden in een tijdelijke tabel opgeslagen, en de tijdelijke tabel wordt vervolgens door de hoofdquery gebruikt.

Zie de Oracle-documentatie voor meer details en voorbeelden; de Engelse benaming voor deze WITH-syntaxis is ‘subquery factoring’.

## 9.5

### Hiërarchische queries

Tabellen zijn in principe ‘platte’ structuren; alle rijen zijn even belangrijk, en de volgorde van de rijen in een tabel is niet relevant. Dit kan problemen veroorzaken bij de raadpleging van hiërarchische gegevens in relationele databases. Bijvoorbeeld: de

managementstructuur in onze MEDEWERKERS-tabel.

In hiërarchische structuren spreken we over het algemeen van ‘parents’

en ‘siblings’; we zullen deze Engelse terminologie in deze paragraaf gebruiken.

Het in de vorige paragraaf geïntroduceerde concept subquery factoring, kan op een speciale manier gebruikt worden om queries op

hiërarchische gegevens te formuleren. Laten we eerst maar eens naar een voorbeeld kijken; zie [figuur 9.15](#).

```

SQL> with MDW (mnr, naam, nivo) as
  2  ( select m.mnr, m.naam, 1 as nivo
  3    from medewerkers m
  4   where m.chef is null
  5  UNION ALL
  6    select m.mnr, m.naam, c.nivo + 1
  7    from medewerkers m
  8    ,MDW c
  9   where m.chef = c.mnr )
10  SEARCH DEPTH FIRST BY NAAM SET VOLGORDE
11  select mnr, naam, nivo
12  from mdw
13  order by volgorde;

```

MNR	NAAM	NIVO
7839	DE KONING	1
7698	BLAAK	2
7499	ALDERS	3
7521	DE WAARD	3
7844	DEN DRAAIER	3
7900	JANSEN	3
7654	MARTENS	3
7782	CLERCKX	2
7934	MOLENAAR	3
7566	JANSEN	2
7788	SCHOTTEN	3
7876	ADAMS	4
7902	SPIJKER	3
7369	SMIT	4

14 rows selected.

SQL>

## Figuur 9.15

De hierboven gebruikte manier van de WITH-component wordt

‘recursive subquery factoring’ genoemd. De definitie van de subquery in de WITH-component is recursief omdat de naam die deze subquery heeft gekregen op regel 1 (MDW), in de tekst van die subquery zelf weer gebruikt wordt (zie regel 8). Met recursive subquery factoring kunnen we de hiërarchische verbanden (boomstructuur) in een tabel langslopen; dat doen we door een startpunt van de boomstructuur aan te geven, en de manier waarop je

in de boomstructuur vanuit iedere willekeurige rij naar beneden kunt wandelen.

Hier voor bevatt de WITH-component altijd twee subqueries die middels een UNION ALL gekoppeld worden. De eerste subquery geeft het startpunt van de boomstructuur aan; in dit geval zijn dat alle 251

■

■

medewerkers die zelf niet een chef hebben (*m.chef* is null). De tweede subquery geeft aan hoe we in de boomstructuur naar beneden kunnen wandelen. In dit geval gaat dat door te zoeken naar medewerkers wier CHEF-nummer overeenkomt met het MNR-nummer van de medewerker die één niveau hoger in de boomstructuur zit (zie regel 9).

**Let op:** We moeten er zelf voor zorgen het juiste startpunt van de hiërarchie aan te geven. We gebruiken ‘chef is null’ als conditie, omdat we weten dat de null-waarde in de CHEF-kolom een bepaalde betekenis heeft. Oracle behandelt *iedere* rij die resulteert uit de eerste subquery als de ‘wortel’ van een aparte boomstructuur; we kunnen dus meerdere boomstructuren in dezelfde query aanleggen. In ons voorbeeld zal dit er maar één zijn, omdat alleen medewerker De Koning een niet ingevulde CHEF-waarde heeft.

De eerste subquery wordt het ‘anchor member’ genoemd en de tweede subquery wordt het ‘recursive member’ genoemd. Oracle voert een hiërarchische query als volgt uit:

1

Voer het anchor member uit en bepaal daarmee de eerste laag in de hiërarchie.

2

Voer het recursive member uit en gebruik daarbij de vorige laag als input. Het resultaat van deze subquery geeft een (nieuwe) volgende laag in de hiërarchie.

3

Herhaal stap 2 hierboven totdat uitvoering van de recursive member leidt tot een leeg resultaat (i.e. de volgende laag bevat geen rijen meer).

4

Geef het hele resultaat terug: dit is de UNION ALL van alle gevonden lagen in de hiërarchie.

Door deze recursieve uitvoering van de tweede subquery zijn we in staat om zelf een nummer te genereren dat voor elke rij het ‘niveau’ in de boomstructuur aangeeft. Alle rijen van de eerste subquery hebben niveau 1 (zie regel 2, 1 as nivo), en voor rijen voortkomend uit volgende niveaus verhogen we dit nummer elke keer (zie regel 6, c.nivo + 1).

Op regel 10 (en 13) [in figuur 9.15 specificeren](#) we in welke volgorde we de rijen in het resultaat willen krijgen.

DEPTH FIRST geeft aan dat we per (parent-)rij op een niveau, eerst de 252

- 
- 

siblings van die rij willen rapporteren, alvorens naar de volgende parent rij op dat niveau te gaan. Het alternatief is BREADTH FIRST.

BY NAAM geeft aan dat we binnen één niveau de rijen gesorteerd willen hebben op NAAM.

SET VOLGORDE introduceert een nieuwe kolom die we kunnen

gebruiken om het resultaat conform de twee hierboven staande bullets te sorteren. In dit geval introduceren we een nieuwe kolom genaamd VOLGORDE en gebruiken deze in de ORDER BY-component van de hoofdquery. Oracle zal zorgen dat deze kolom gevuld wordt met een volgnummer dat de juiste sortering aangeeft.

Het resultaat van de query [in figuur 9.15 is](#) op het eerste gezicht niet erg imponerend; we krijgen ‘gewoon’ de nummers en namen van alle medewerkers te zien, gevuld door een getal. En als we NIVO hadden

weggelaten, was er helemaal niets bijzonders te zien geweest. Achter de schermen is echter van alles gebeurd, zoals we hierboven aangegeven hebben.

In figuur 9.16 zien we een voorbeeld van het gebruik van NIVO, gecombineerd met de LPAD-functie, om het resultaat van deze hiërarchische query leesbaarder te maken:

```

SQL> column naam format a30
SQL> with MDW (mnr, naam, nivo) as
  2  ( select m.mnr, m.naam, 1 as nivo
  3    from medewerkers m
  4   where m.mnr=7839
  5   UNION ALL
  6   select m.mnr, m.naam, c.nivo + 1
  7    from medewerkers m
  8      ,MDW c
  9   where m.chef = c.mnr )
10  SEARCH DEPTH FIRST BY NAAM SET VOLGORDE
11  select mnr, lpad(' ',2*nivo-1)||naam as naam, nivo
12  from mdw
13  order by volgorde;

```

MNR	NAAM	NIVO
7839	DE KONING	1
7698	BLAAK	2
7499	ALDERS	3
7521	DE WAARD	3
7844	DEN DRAAIER	3
7900	JANSEN	3
7654	MARTENS	3
7782	CLERCKX	2
7934	MOLENAAR	3
7566	JANSEN	2
7788	SCHOTTEM	3
7876	ADAMS	4
7902	SPIJKER	3
7369	SMIT	4

14 rows selected.

SQL>

## Figuur 9.16

Door in de recursive member handig gebruik te maken van

kolomwaarden uit het vorige niveau, zijn we in staat diverse interessante gegevens in het resultaat te tonen. Bijvoorbeeld: geef per medewerker de hoogste manager, inclusief het hiërarchisch pad naar deze manager.

Zie [figuur 9.17 voor een voorbeeld](#) hiervan. Merk op dat we de anchor

member hebben veranderd zodat we nu een aparte boomstructuur krijgen voor iedere manager.

254

```

SQL> with MDW (mnr, naam, manager, pad) as
  2  (select m.mnr, m.naam, m.naam as manager, ' > ' ||m.naam as pad
  3   from medewerkers m
  4  where m.functie = 'MANAGER'
  5  UNION ALL
  6  select m.mnr, m.naam, c.manager, c.pad||' > '||m.naam
  7   from medewerkers m
  8   ,MDW c
  9  where m.chef = c.mnr)
10 SEARCH DEPTH FIRST BY NAAM SET VOLGORDE
11 select mnr, naam, manager, pad
12 from mdw
13 order by volgorde;

```

MNR	NAAM	MANAGER	PAD
7698	BLAAK	BLAAK	> BLAAK
7499	ALDERS	BLAAK	> BLAAK > ALDERS
7521	DE WAARD	BLAAK	> BLAAK > DE WAARD
7844	DEN DRAAIER	BLAAK	> BLAAK > DEN DRAAIER
7900	JANSEN	BLAAK	> BLAAK > JANSEN
7654	MARTENS	BLAAK	> BLAAK > MARTENS
7782	CLERCKX	CLERCKX	> CLERCKX
7934	MOLENAAR	CLERCKX	> CLERCKX > MOLENAAR
7566	JANSEN	JANSEN	> JANSEN
7788	SCHOTTEN	JANSEN	> JANSEN > SCHOTTEN
7876	ADAMS	JANSEN	> JANSEN > SCHOTTEN > ADAMS
7902	SPIJKER	JANSEN	> JANSEN > SPIJKER
7369	SMIT	JANSEN	> JANSEN > SPIJKER > SMIT

13 rows selected.

SQL>

## Figuur 9.17

De hiërarchische structuur die in de MEDEWERKERS-tabel gevormd wordt door de MNR- en CHEF-kolommen, mag natuurlijk geen cyclisch verband bevatten: in dat geval zou de recursie namelijk oneindig diep doorgaan.

Om dit te voorkomen detecteert Oracle zo'n situatie. In [figuur 9.18](#) zien we dat er een foutmelding gegeven wordt indien we medewerker Martens de chef laten zijn van de directeur. We hebben het anchor member moeten aanpassen om de query nog steeds met de directeur te laten beginnen.



```

SQL> UPDATE medewerkers set chef = 7654
  2 WHERE chef is null;

1 row updated.

SQL> with MDW (mnr, naam, nivo) as
  2 ( select m.mnr, m.naam, 1 as nivo
  3   from medewerkers m
  4  where m.functie = 'DIRECTEUR'
  5  UNION ALL
  6  select m.mnr, m.naam, c.nivo + 1
  7   from medewerkers m
  8      ,MDW c
  9  where m.chef = c.mnr )
10  SEARCH DEPTH FIRST BY NAAM SET VOLGORDE
11  select mnr, lpad(' ',2*nivo-1)||naam as naam
12  from mdw
13  order by volgorde;

from mdw
*
ERROR at line 12:
ORA-32044: cycle detected while executing recursive WITH query

SQL>

```

## Figuur 9.18

In figuur 9.19 zien we dat deze situatie ook te detecteren is zonder dat er een foutmelding gegeven wordt. Middels de CYCLE-component kunnen we angeven dat er bij detectie van een cyclisch verband niet een foutmelding gegeven moet worden, maar dat de query gewoon door dient te gaan zonder dat de oneindige recursie ingedoken wordt.

```

SQL> with MDW (mnr, naam, nivo) as
  2  ( select m.mnr, m.naam, 1 as nivo
  3    from medewerkers m
  4   where m.functie = 'DIRECTEUR'
  5  UNION ALL
  6   select m.mnr, m.naam, c.nivo + 1
  7    from medewerkers m
  8      ,MDW c
  9   where m.chef = c.mnr )
10  SEARCH DEPTH FIRST BY NAAM SET VOLGORDE
11 CYCLE MNR SET IS_CYCLE TO 'Y' DEFAULT 'N'
12 select mnr, lpad(' ',2*nivo-1)||naam as naam, is_cycle
13 from mdw
14 order by volgorde;

```

More...

MNR	NAAM	I
7839	DE KONING	N
7698	BLAAK	N
7499	ALDERS	N
7521	DE WAARD	N
7844	DEN DRAAIER	N
7900	JANSEN	N
7654	MARTENS	N
7839	DE KONING	Y
7782	CLERCKX	N
7934	MOLENAAR	N
7566	JANSEN	N
7788	SCHOTTEM	N
7876	ADAMS	N
7902	SPIJKER	N
7369	SMIT	N

15 rows selected.

SQL>

## Figuur 9.19

In de CYCLE-component dienen we aan te geven welke kolom Oracle kan gebruiken om een cyclisch verband te detecteren. In figuur 9.19 is dat de MNR-kolom. En we introduceren een nieuwe kolom (middels de SET

IS\_CYCLE) waarin Oracle kan aangeven op welke plek in de

boomstructuur het cyclisch verband gedetecteerd is.

## 9.6

### Vensters en analytische functies

In deze paragraaf maken we kennis met het begrip *venster* (*window*), en 257

- 
- 
- 

zullen we vervolgens zien wat we met vensters kunnen doen door er analytische functies op toe te passen. Een venster is een

deelverzameling van de tabel die we aan het raadplegen zijn; voor iedere rij kan dit venster er verschillend uitzien.

Gesorteerde vensters zijn het meest interessant, omdat we (door het venster te sorteren) allerlei operaties op het venster kunnen toepassen die binnen het venster een bepaalde volgorde veronderstellen. Typische voorbeelden van zulke vensters zijn:

Alle rijen vanaf het begin van de tabel tot de huidige rij.

De huidige rij, aangevuld met de vorige en de volgende rij.

Alle rijen vanaf de huidige rij tot het eind van de tabel.

Merk op dat ‘het begin’ en ‘het eind’ van de tabel, de ‘vorige’ en de ‘volgende’ rij inderdaad alleen maar betekenis kunnen hebben binnen een *gesorteerd* venster; tabellen zijn immers van nature ongeordend. Dit betekent ook dat we voor onze vensters de *juiste* sortering moeten kiezen als we voorspelbare queryresultaten willen zien. Als we bijvoorbeeld sorteren op een kolom (of kolomcombinatie) die niet uniek is binnen het betreffende venster, dan wordt het begrip ‘vorige’ en ‘volgende’ rij binnen zo’n venster onduidelijk.

Laten we maar eens naar een voorbeeld kijken [in figuur 9.20.](#)

258

```

SQL> select chef, naam, maandsal
  2 ,      sum(maandsal) over
  3      ( order by chef, maandsal
  4        range unbounded preceding
  5      ) as cumulatief
  6 from medewerkers
  7 order by chef, maandsal;

```

CHEF	NAAM	MAANDSAL	CUMULATIEF
7566	SCHOTTEN	3000	6000
7566	SPIJKER	3000	6000
7698	JANSEN	800	6800
7698	DE WAARD	1250	9300
7698	MARTENS	1250	9300
7698	DEN DRAAIER	1500	10800
7698	ALDERS	1600	12400
7782	MOLENAAR	1300	13700
7788	ADAMS	1100	14800
7839	CLERCKX	2450	17250
7839	BLAAK	2850	20100
7839	JANSEN	2975	23075
7902	SMIT	800	23875
	DE KONING	5000	28875

SQL>

## Figuur 9.20

We krijgen een cumulatief maandsalaris te zien, omdat we per medewerker een venster definiëren vanaf het begin (UNBOUNDED

PRECEDING) tot aan de rij zelf. De huidige rij is in dit geval het standaardeinde van het venster; we hoeven dat niet expliciet aan te geven. Het cumulatieve resultaat gedraagt zich niet helemaal naar wens; kijk bijvoorbeeld naar de eerste twee rijen in [figuur 9.20. Dit](#) komt doordat de sortering niet precies genoeg is; SCHOTTEN en SPIJKER

hebben dezelfde chef en ook hetzelfde maandsalaris. We kunnen dit probleem verhelpen door de ORDER BY-component op regel 3 iets te veranderen; zie [figuur 9.21.](#)



```

SQL> select chef, naam, maandsal
  2 ,      sum(maandsal) over
  3      ( order by chef, maandsal, mnr
  4        range unbounded preceding
  5      ) as cumulatief
  6 from medewerkers
  7 order by chef, maandsal;

```

CHEF	NAAM	MAANDSAL	CUMULATIEF
7566	SCHOTTEN	3000	3000
7566	SPIJKER	3000	6000
7698	JANSEN	800	6800
7698	DE WAARD	1250	8050
7698	MARTENS	1250	9300
7698	DEN DRAAIER	1500	10800
7698	ALDERS	1600	12400
7782	MOLENAAR	1300	13700
7788	ADAMS	1100	14800
7839	CLERCKX	2450	17250
7839	BLAAK	2850	20100
7839	JANSEN	2975	23075
7902	SMIT	800	23875
	DE KONING	5000	28875

SQL>

## Figuur 9.21

We bouwen nog even op dit voorbeeld verder. Vensters kunnen we (net zoals bij outerjoins) partitioneren, zodat de analytische functies steeds opnieuw beginnen. In figuur 9.21 loopt de cumulatieve som gewoon door; in figuur 9.22 start de cumulatieve som voor iedere manager opnieuw, doordat we op de vierde regel PARTITION BY hebben toegevoegd.

```

SQL> select chef, naam, maandsal
  2 ,      sum(maandsal) over
  3      ( PARTITION BY chef
  4          order by chef, maandsal, mnr
  5          range unbounded preceding
  6      ) as cumulatief
  7 from medewerkers
  8 order by chef, maandsal;

```

CHEF	NAAM	MAANDSAL	CUMULATIEF
7566	SCHOTTEN	3000	3000
7566	SPIJKER	3000	6000
7698	JANSEN	800	800
7698	DE WAARD	1250	2050
7698	MARTENS	1250	3300
7698	DEN DRAAIER	1500	4800
7698	ALDERS	1600	6400
7782	MOLENAAR	1300	1300
7788	ADAMS	1100	1100
7839	CLERCKX	2450	2450
7839	BLAAK	2850	5300
7839	JANSEN	2975	8275
7902	SMIT	800	800
	DE KONING	5000	5000

SQL>

## Figuur 9.22

Overigens mogen we in plaats van RANGE ook ROWS gebruiken; ze hebben precies dezelfde betekenis, dus het is een kwestie van smaak.

Enkele typische voorbeelden van RANGE-definities, zonder er concrete voorbeelden van te bekijken, zijn:

ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING.

RANGE BETWEEN 3 PRECEDING AND 3 FOLLOWING.

Als analytische functies kunnen we de ‘gewone’ groepsfuncties toepassen, zoals SUM, MAX, MIN, AVG en COUNT. Oracle biedt daarnaast een heel scala aan overige analytische functies; het zijn er te veel om ze hier zelfs maar te noemen, laat staan ze te bespreken. Zie de Oracle-documentatie voor details en voorbeelden. Aanraders zijn RANK, DENSE\_RANK, LAG en LEAD.

We sluiten deze paragraaf af met een voorbeeld van de functies LAG en LEAD, toegepast op de HISTORIE-tabel; zie [figuur 9.23](#). We krijgen per medewerker en per begindatum het maandsalaris, het vorige

```

SQL> break on mnr

SQL> select mnr, begindatum, maandsal
  2 ,      LAG(maandsal) over
  3      ( partition by mnr
  4        order by mnr, begindatum
  5      ) as vorig_sal
  6 ,      LEAD(maandsal) over
  7      ( partition by mnr
  8        order by mnr, begindatum
  9      ) as volgend_sal
10 from historie
11 order by mnr, begindatum;

MNR BEGINNDATUM MAANDSAL VORIG_SAL VOLGEND_SAL
----- ----- -----
7369 01-JAN-2000    950          800
      01-FEB-2000    800          950
7499 01-JUN-1988   1000         1300
      01-JUL-1989   1300         1000
      01-DEC-1993   1500         1300
      01-OCT-1995   1700         1500
      01-NOV-1999   1600         1700
...
7902 01-SEP-1998   1400         1650
      01-OCT-1998   1650         1400
      15-MAR-1999   2500         1650
      01-JAN-2000   3000         2500
      01-AUG-2000   3000         3000
7934 01-FEB-1998   1275         1280
      01-MAY-1998   1280         1275
      01-FEB-1999   1290         1280
      01-JAN-2000   1300         1290

79 rows selected.

SQL>

```

maandsalaris, en het volgende maandsalaris te zien. Als het vorige of het volgende salaris niet van toepassing is krijgen we een null-waarde terug.

### Figuur 9.23

9.7

# Flashback queries

Deze paragraaf behandelt tot slot van dit hoofdstuk een specifieke SQL-uitbreiding van Oracle die te aardig is om in dit boek onbehandeld te blijven. In hoofdstuk 6 hebben we het over *read consistency* gehad. Dat 262

```
SQL> create table m as select * from medewerkers;

SQL> update m
   2 set     maandsal = maandsal + 10;
SQL> commit;

SQL> update m set     maandsal = maandsal - 20
   2 where    afd = 10;
SQL> commit;

SQL> delete from m
   2 where    afd <= 20;
SQL> commit;
```

bekent voor onze queries dat we altijd een consistent resultaat te zien krijgen, ongeacht wat er op hetzelfde moment door andere

databasegebruikers met de gegevens wordt gedaan. We krijgen gegarandeerd een ‘snapshot’ van de gegevens te zien zoals ze waren toen de query werd gestart. In datzelfde hoofdstuk hebben we ook gezien dat we onze sessie kunnen bestempelen als READ ONLY, waardoor queryresultaten zelfs worden bepaald op basis van de situatie aan het begin van onze sessie.

De manier waarop Oracle dat voor elkaar krijgt (zonder andere gebruikers te beletten dezelfde gegevens te veranderen) laten we hier buiten beschouwing. Feit is dat Oracle blijkbaar in staat is om een situatie uit het verleden te reconstrueren; in deze paragraaf zien we een aantal interessante manieren om van deze techniek gebruik te maken, door in onze queries explicet aan te geven dat we terug willen in de tijd.

Evenmin zullen we de manier behandelen waarop deze features door de databasebeheerder kunnen worden geconfigureerd. Zie de Oracle-documentatie; zoek in de index naar ‘flashback’ voor de details.

Voor onze flashback-experimenten maken we eerst een kopie van de tabel MEDEWERKERS, waarin we wijzigingen kunnen aanbrengen; zie

[figuur 9.24.](#)

### Figuur 9.24

**Tip:** Voer deze vier acties niet te snel achter elkaar uit, maar las steeds enige pauzes in. Daardoor wordt het gemakkelijker om naar een specifieke situatie in de tijd terug te gaan.

```

SQL> select mnr, naam, afd, maandsal
  2  from   m;

      MNR NAAM          AFD MAANDSAL
-----  -----
    7499 ALDERS        30    1610
    7521 DE WAARD      30    1260
    7654 MARTENS       30    1260
    7698 BLAAK         30    2860
    7844 DEN DRAAIER    30    1510
    7900 JANSEN         30     810

SQL> select mnr, naam, afd, maandsal
  2  from   m
  3      as of timestamp to_timestamp('&timestamp');

Enter value for timestamp: 05-FEB-12 09.24.51.266670 AM

      MNR NAAM          AFD MAANDSAL
-----  -----
    7369 SMIT           20     810
    7499 ALDERS        30    1610
    7521 DE WAARD      30    1260
    7566 JANSEN         20    2985
    7654 MARTENS       30    1260
    7698 BLAAK         30    2860
    7782 CLERCKX        10    2460
    7788 SCHOTTON       20    3010
    7839 DE KONING      10    5010
    7844 DEN DRAAIER    30    1510
    7876 ADAMS          20    1110
    7900 JANSEN         30     810
    7902 SPIJKER        20    3010
    7934 MOLENAAR       10    1310

SQL> /
Enter value for timestamp: 04-FEB-12 09.00.00.000000 AM
from   m
*
ERROR at line 2:
ORA-01466: unable to read data - table definition has changed

SQL>

```

In figuur 9.25 zien we een eerste voorbeeld. Eerst vragen we de huidige situatie op, en dan gebruiken we AS OF TIMESTAMP in de FROM-component om terug te gaan in de tijd. We gebruiken (zoals we al eerder

hebben gedaan) de ampersand (&) zodat we de query met verschillende waarden kunnen uitvoeren.

**Figuur 9.25**

264

```

SQL> break on mnr

SQL> select mnr, maandsal
  2 ,      versions_starttime
  3 ,      versions_endtime
  4 from   m
  5      versions between timestamp minvalue and maxvalue
  6 where  afd = 10
  7 order  by mnr, versions_starttime nulls first;

          MNR MAANDSAL VERSIONS_STARTTIME      VERSIONS_ENDTIME
-----  -----  -----
    7782      2450                      05-FEB-12 09.24.50 AM
              2460 05-FEB-12 09.24.50 AM 05-FEB-12 09.25.04 AM
              2440 05-FEB-12 09.25.04 AM 05-FEB-12 09.25.47 AM
              2440 05-FEB-12 09.25.47 AM
    7839      5000                      05-FEB-12 09.24.50 AM
              5010 05-FEB-12 09.24.50 AM 05-FEB-12 09.25.04 AM
              4990 05-FEB-12 09.25.04 AM 05-FEB-12 09.25.47 AM
              4990 05-FEB-12 09.25.47 AM
    7934      1300                      05-FEB-12 09.24.50 AM
              1310 05-FEB-12 09.24.50 AM 05-FEB-12 09.25.04 AM
              1290 05-FEB-12 09.25.04 AM 05-FEB-12 09.25.47 AM
              1290 05-FEB-12 09.25.47 AM

SQL>

```

Uiteraard zijn de gebruikte timestamp-waarden [in figuur 9.25](#)

afhankelijk van het tijdstip waarop de acties [in figuur 9.24 zijn](#) uitgevoerd. Uit de foutmelding [in figuur 9.25 blijkt](#) dat we terug willen naar een te vroeg tijdstip; in dit geval bestond de tabel nog niet.

[In figuur 9.26 gaan we nog een stapje](#) verder, waarbij we de volledige historie van rijen kunnen opvragen, voorzover ze te reconstrueren zijn.

## Figuur 9.26

Door de VERSIONS BETWEEN-operator in de FROM-component krijgen we de beschikking over een aantal extra pseudo-kolommen die we in onze queries kunnen gebruiken, waaronder VERSIONS\_STARTTIME en

VERSIONS\_ENDTIME.

Door de juiste ORDER BY-component te gebruiken (let op de NULLS

FIRST in figuur 9.26) krijgen we een compleet historisch overzicht. Van de oudste waarde is het starttijdstip niet bekend, en van de laatste (huidige) waarde is het eindtijdstip natuurlijk nog niet bekend.

265

```
SQL> drop table m;
Table dropped.

SQL> flashback table m to before drop;
Flashback complete.

SQL> select * from m;

MNR NAAM          VOORL FUNCTIE      CHEF GBDATUM      MAANDSAL COMM AFD
---- -----          ----- -----      ----- -----      -----
7499 ALDERS        JAM   VERKOPER    7698 20-FEB-1961      1610  300  30
7521 DE WAARD      TF    VERKOPER    7698 22-FEB-1962      1260  500  30
7654 MARTENS       P    VERKOPER    7698 28-SEP-1956      1260 1400  30
7698 BLAAK         R    MANAGER     7839 01-NOV-1963      2860      0  30
7844 DEN DRAAIER   JJ   VERKOPER    7698 28-SEP-1968      1510      0  30
7900 JANSEN        R   BOEKHOUDER 7698 03-DEC-1969      810      0  30

SQL>
```

In hoofdstuk 7 (paragraaf 7.9) hebben we al gezien dat we met behulp van het commando FLASHBACK TABLE een per ongeluk verwijderde tabel uit de ‘recycle bin’ kunnen redden; zie figuur 9.27.

## Figuur 9.27

9.8

### Opgaven

1

Het komt vaak voor dat een (junior) docent een cursus eerst bij een collega volgt voordat hij deze zelf verzorgt. Bij welke

docent/cursus-combinaties is dat gebeurd?

2

Sterker nog: als de beginnende docent de cursus dan voor het eerst geeft, zit de docent waarvan hij eerder de kunst heeft afgekeken, ter ondersteuning als deelnemer in het lokaal. Spoor dit soort cursus/junior/senior-combinaties op.

3

Welke medewerkers hebben nog nooit een cursus gegeven?

4

Welke werknemers hebben alle bouwcursussen (type BLD) gevolgd? Ze hebben namelijk recht op korting.

5

Wie hebben (ten minste) dezelfde cursussen gevolgd als medewerker 7788 gevolgd heeft?

6

Geef de gegevens van alle medewerkers waarvan het maandsalaris en de commissie overeenkomen met het maandsalaris en de commissie van (minstens) een medewerker van afdeling 30. We zijn natuurlijk alleen geïnteresseerd in medewerkers van andere afdelingen.

266

7

Zijn de twee queries van figuur 9.4 en 9.5 wel equivalent? Zoek eens ‘per ongeluk’ op een niet-voorkomende functie, en voer ze dan nog eens

allebei uit. Verklaar het resultaat...

8

Een serie voorbeelden in dit hoofdstuk betrof de personalia van alle medewerkers die ooit een S02-cursus hadden gegeven (zie de

[figuren 9.9 tot en met 9.12](#)). Hoe kunnen we deze queries aanpassen zodat ze antwoord geven op de ontkenning van dezelfde vraag (...

die *nooit* ...)

9

Kijk nog eens naar opgave 4 van het vorige hoofdstuk: geef van alle cursusuitvoeringen de cursuscode, de begindatum, en het aantal inschrijvingen. Sorteer op begindatum. Kunnen we nu een

elegantere oplossing bedenken, zonder een outerjoin te gebruiken?

10 Geef naam en voorletters van de medewerkers die ‘onder aan’ de hiërarchie hangen, met in een derde kolom het aantal

managementniveaus dat ze boven zich hebben.

11 Net zoals subqueries in de SELECT-component een GROUP BY-component overbodig kunnen maken, kunnen subqueries in de FROM-component de HAVING-component overbodig maken. Licht dit toe met een voorbeeld.

267

## **Hoofdstuk 10**

### **Views**

Views vormen een belangrijk onderdeel van het relationele model.

In de eerste paragraaf introduceren we het begrip view, en behandelen we het

CREATE VIEW-commando, waarmee we views

kunnen creëren.

In [paragraaf 10.2](#) worden de diverse toepassingsmogelijkheden van views toegelicht, op het gebied van raadpleging, logische gegevensonafhankelijkheid en beveiliging.

In een volgende paragraaf gaan we nader in op datamanipulatie via views: wat moeten we ons daarbij voorstellen, en wat zijn de beperkingen waar we rekening mee moeten houden?

Vervolgens wordt de CHECK OPTION behandeld. Als views

datamanipulatie toestaan, dan kunnen we met deze toevoeging integriteitscontrole afdwingen. De vijfde paragraaf gaat over datamanipulatie via inline views. Inline views zijn eigenlijk geen echte views; het zijn subqueries. Behandeling van dit onderwerp [in hoofdstuk 6](#) (over datamanipulatie) zou te vroeg zijn gekomen; bovendien sluit het onderwerp in dit hoofdstuk mooi aan bij de voorgaande twee paragrafen. Met datamanipulatie via inline views kunnen we diverse gecompliceerde datamanipulatie-operaties verrichten.

In paragraaf 6 komt performance aan de orde, en in de laatste paragraaf bespreken we materialized views. Materialized views worden vooral in data warehouse-omgevingen veel toegepast; ze stellen ons in staat om – met behulp van een gecontroleerde vorm van redundantie – de responsstijden te verkorten.

## 10.1 Wat zijn views?

Het resultaat van een query is altijd een tabel – iets preciezer geformuleerd: een *afgeleide* tabel. vergeleken met ‘echte’ tabellen in de database is het resultaat van een query vluchtig. Maar toch: het is en blijft een tabel. Het enige wat er eigenlijk aan ontbreekt is een naam. In principe is een view niets anders dan een query waaraan een naam is 268



gegeven.

Een view is een virtuele tabel met als ‘inhoud’ het resultaat van een vastgelegde view-query die bij iedere benadering opnieuw wordt uitgevoerd.

De eerste regel van deze definitie zegt twee dingen:

1

Een view is een *virtuele tabel*.

Dit wil zeggen dat een view zich in (bijna) alle opzichten gedraagt en laat benaderen als een tabel. Een view heeft een naam; vandaar dat men views ook wel ‘named queries’ noemt. Een view heeft kolommen met naam en datatype, een view kan met queries worden geraadpleegd, en de ‘inhoud’ van een view kan (onder bepaalde restricties) worden gemanipuleerd met behulp van insert, update en delete.

2

Een view is een *virtuele tabel*.

In werkelijkheid is er geen sprake van een echte tabel. Views hebben géén inhoud; vandaar de ‘aanhalingsstekens’ in de definitie. Ze zijn gedefinieerd in de vorm van view-queries, die in de datadictionary worden opgeslagen; vandaar dat men ook wel spreekt van ‘stored queries’. Telkens als we een beroep doen op de ‘inhoud’ van de view, wordt de view-query opgehaald en uitgevoerd. Het resultaat daarvan wordt vervolgens als inhoud van de view gebruikt.

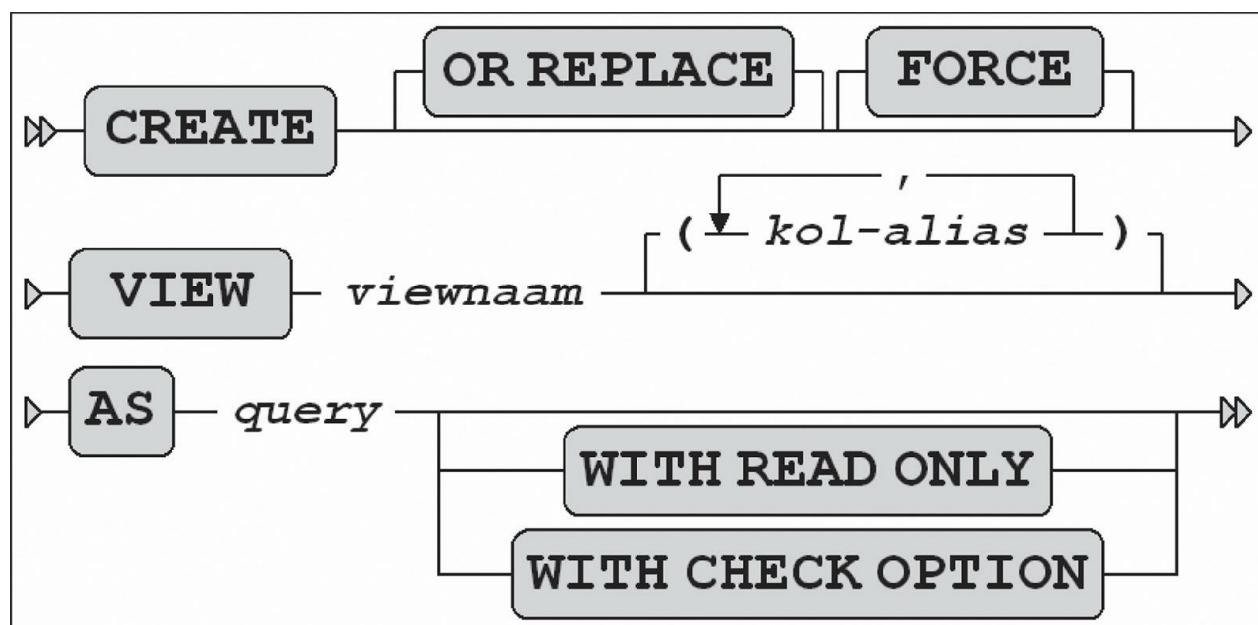
Views worden gecreëerd met het SQL-commando CREATE VIEW,

waarvan het syntax-diagram [in figuur 10.1 is](#) weergegeven. Met de optie OR REPLACE kunnen we een bestaande view-definitie vervangen. Dat is met name handig als aan de view allerlei privileges verbonden zijn; die zouden bij een DROP VIEW-commando verloren gaan, terwijl ze bij een CREATE OR REPLACE VIEW behouden blijven. De FORCE-optie controleert niet of de onderliggende tabellen bestaan en/of we daarop wel voldoende privileges

hebben. Uiteraard moet wel aan deze voorwaarden zijn voldaan tegen de tijd dat we de view ook willen gaan gebruiken.

Kolomnamen voor de view worden normaal gesproken van de query overgenomen. Daarbij kunnen zich echter problemen voordoen. In het resultaat van een query kunnen bijvoorbeeld twee kolommen

voorkomen met dezelfde naam, of er kunnen functies of berekeningen in 269



een kolomkop voorkomen.

**Figuur 10.1**

Een query-resultaat met dit soort problemen kan natuurlijk niet zonder meer worden gebruikt als basis voor een view-definitie; net als in een tabeldefinitie worden ook hier bepaalde eisen gesteld aan de kolomnamen. Ze moeten onderling verschillend zijn en mogen bijvoorbeeld geen haakjes ( ) bevatten. Deze problemen kunnen we op twee manieren oplossen:

1

We kunnen in de SELECT-component van de query kolom-aliassen opnemen, zodat de kolomtitels alsnog voldoen aan de normale conventies (deze methode zullen we zo veel mogelijk toepassen).

We kunnen in het CREATE VIEW-commando achter de view-naam tussen haakjes kolom-aliassen declareren ([zie figuur 10.1](#)).

De toevoegingen WITH CHECK OPTION en WITH READ ONLY hebben betrekking op datamanipulatie via views; hieraan besteden we later in dit hoofdstuk speciaal aandacht.

[In figuur 10.2](#) zien we een voorbeeld van een ‘gewone’ query, met het resultaat. Het is een join over drie tabellen, waarin van iedere medewerker informatie wordt gegeven over zijn of haar afdeling. Merk op dat we er met behulp van kolom-aliassen voor hebben gezorgd dat iedere kolom van het resultaat een andere naam heeft, terwijl we drie kolommen met dezelfde naam (NAAM) selecteren.

```

SQL> select m.mnr
  2 ,      m.NAAM
  3 ,      m.voort
  4 ,      a.NAAM   as afdeling
  5 ,      a.locatie
  6 ,      h.NAAM   as hoofd
  7 from medewerkers m
  8 ,      afdelingen a
  9 ,      medewerkers h
10 where m.afd = a.anr
11 and   h.mnr = a.hoofd;

```

MNR	NAAM	VOORT	AFDELING	LOCATIE	HOOFD
7902	SPIJKER	MG	OPLIEDINGEN	DE MEERN	JANSEN
7876	ADAMS	AA	OPLIEDINGEN	DE MEERN	JANSEN
7788	SCHOTTEN	SCJ	OPLIEDINGEN	DE MEERN	JANSEN
7566	JANSEN	JM	OPLIEDINGEN	DE MEERN	JANSEN
7369	SMIT	N	OPLIEDINGEN	DE MEERN	JANSEN
7900	JANSEN	R	VERKOOP	UTRECHT	BLAAK
7844	DEN DRAAIER	JJ	VERKOOP	UTRECHT	BLAAK
7698	BLAAK	R	VERKOOP	UTRECHT	BLAAK
7654	MARTENS	P	VERKOOP	UTRECHT	BLAAK
7521	DE WAARD	TF	VERKOOP	UTRECHT	BLAAK
7499	ALDERS	JAM	VERKOOP	UTRECHT	BLAAK
7934	MOLENAAR	TJA	HOOFDKANTOOR	LEIDEN	CLERCKX
7839	DE KONING	CC	HOOFDKANTOOR	LEIDEN	CLERCKX
7782	CLERCKX	AB	HOOFDKANTOOR	LEIDEN	CLERCKX

SQL>

```
SQL> create view empdept_v as
  2  select m.mnr
  3 ,      m.naam
  4 ,      m.voornl
  5 ,      a.naam   as afdeling
  6 ,      a.locatie
  7 ,      h.naam   as hoofd
  8 from  medewerkers m
  9 ,      afdelingen a
 10 ,      medewerkers h
 11 where m.afd = a.anr
 12 and   h.mnr = a.hoofd;
```

View created.

```
SQL>
```

## Figuur 10.2

Op deze query baseren we vervolgens een view, door een eerste regel aan het commando toe te voegen; zie [figuur 10.3](#).

271

```
SQL> select * from tab;
```

TNAME	TABTYPE	CLUSTERID
MEDEWERKERS	TABLE	
AFDELINGEN	TABLE	
SCHALEN	TABLE	
CURSUSSEN	TABLE	
UITVOERINGEN	TABLE	
INSCHRIJVINGEN	TABLE	
HISTORIE	TABLE	
M	TABLE	
EMPDEPT_V	VIEW	

```
SQL>
```

```

SQL> describe empdept_v
Name          Null?    Type
-----
MNR           NOT NULL NUMBER(4)
NAAM          NOT NULL VARCHAR2(12)
VOORL         NOT NULL VARCHAR2(5)
AFDELING      NOT NULL VARCHAR2(20)
LOCATIE        NOT NULL VARCHAR2(20)
HOOFD          NOT NULL VARCHAR2(12)

SQL> select * from empdept_v where hoofd = 'CLERCKX';

MNR  NAAM       VOORL  AFDELING    LOCATIE    HOOFD
-----  -----
7782 CLERCKX   AB     HOOFDKANTOOR LEIDEN    CLERCKX
7839 DE KONING CC     HOOFDKANTOOR LEIDEN    CLERCKX
7934 MOLENAAR  TJA    HOOFDKANTOOR LEIDEN    CLERCKX

SQL>

```

### Figuur 10.3

Deze view is nu toegevoegd aan onze verzameling databaseobjecten. In [figuur 10.4](#) zien we bijvoorbeeld hoe de view in de datadictionary kan worden aangetroffen.

### Figuur 10.4

In [figuur 10.5](#) blijkt dat we het SQL\*Plus-commando DESCRIBE op een view kunnen loslaten, net als op gewone tabellen, en er queries op kunnen formuleren.

```
SQL> create view dept20_v as
  2  select * from medewerkers where afd = 20;

View created.

SQL> create table dept20_t as
  2  select * from medewerkers where afd = 20;

Table created.

SQL>
```

## Figuur 10.5

Let goed op het verschil tussen de twee commando's van figuur 10.6.

## Figuur 10.6

De inhoud van de view DEPT20\_V zal te allen tijde afhankelijk blijven van de tabel MEDEWERKERS, voorzover je van een inhoud kunt spreken.

De tabel DEPT20\_T is een zelfstandige tabel (met een eigen inhoud) die een eigen leven zal gaan leiden, onafhankelijk van de tabel MEDEWERKERS.

Datamanipulatie op een view is in principe iets ongerijmds; er is immers geen sprake van een eigen inhoud. Toch zullen views zich zo veel mogelijk als tabellen gedragen, dus ook (zo mogelijk) datamanipulatie toestaan. Deze datamanipulatie wordt dan vertaald naar

corresponderende acties op de onderliggende tabellen. Dit is niet onder alle omstandigheden mogelijk: [in paragraaf 10.3 komen](#) deze beperkingen aan de orde.

Views zijn niet alleen gevoelig voor wijzigingen in de inhoud van de basistabellen waarop ze zijn gedefinieerd, maar ook op de

onderliggende structuur. Een view zal bijvoorbeeld niet meer functioneren als een onderliggende tabel wordt verwijderd.

We kunnen de view-definitie opvragen in de datadictionary, om de onderliggende query te bestuderen ([zie figuur 10.7](#)). Let even niet op de eerste twee SQL\*Plus-commando's; daar komen we in het volgende hoofdstuk nog op terug. Ze zorgen voor een iets leesbaarder resultaat.

273

```

SQL> set    long 999
SQL> column text format a40 word_wrapped

SQL> select view_name, text
  2  from  user_views;

VIEW_NAME          TEXT
-----
DEPT20_V           select "MNR", "NAAM", "VOORL", "FUNCTIE"
                   , "CHEF", "GBDATUM", "MAANDSAL", "COMM", "AFD"
                   from medewerkers where afd = 20

EMPDEPT_V          select m.mnr
                   , m.naam
                   , m.voorl
                   , a.naam   as afdeling
                   , a.locatie
                   , h.naam   as hoofd
                   from medewerkers m
                   afdelingen a
                   medewerkers h
                   where m.afd = a.anr
                   and h.mnr = a.hoofd

SQL>

```

## Figuur 10.7

Als views worden gedefinieerd met een query die begint met ‘SELECT \* FROM ...’, wordt het sterretje blijkbaar intern uitgewerkt tot (en opgeslagen als) een reeks kolomnamen; vergelijk de tekst van de V20-view in [figuur 10.7 en het](#) commando waarmee we die view hebben gemaakt [in figuur 10.6](#).

De definitie van een view kan niet worden gewijzigd. Views kunnen alleen worden verwijderd, of met de CREATE OR REPLACE-optie worden vervangen.

Views kunnen worden verwijderd met behulp van het volgende commando:

**SQL> drop view <view\_naam>;**

## 10.2 Toepassingsmogelijkheden

Views hebben zeer uiteenlopende toepassingsmogelijkheden. In deze 274

```

SQL> select m.mnr, m.naam
  2 ,      sum(c.lengte) as dagen
  3 from inschrijvingen i
  4 ,
  5 cursussen      c
  6 ,      medewerkers      m
  7 where m.mnr = i.cursist
  8 and   c.code = i.cursus
  9 group by m.mnr, m.naam;

```

MNR	NAAM	DAGEN
7900	JANSEN	3
7499	ALDERS	11
7521	DE WAARD	1
7566	JANSEN	5
7698	BLAAK	12
7782	CLERCKX	4
7788	SCHOTTEN	12
7839	DE KONING	8
7844	DEN DRAAIER	1
7876	ADAMS	9
7902	SPIJKER	9
7934	MOLENAAR	4

SQL>

paragraaf sommen we ze op en lichten we ze toe.

Views kunnen bij raadpleging van de database *vereenvoudigend* werken. Complexe queries kunnen stap voor stap worden opgezet, waardoor uiteindelijk de zekerheid groter is dat de query het correcte antwoord oplevert.

Bovendien kunnen vaak voorkomende *standaardqueries* in een view worden ondergebracht, wat het maken van onnodige fouten voorkomt.

Hierbij kan men bijvoorbeeld denken aan views gebaseerd op vaak gejoinde tabellen, een UNION, of een complexe GROUP BY-constructie.

Stel dat we geïnteresseerd zijn in een overzicht van alle medewerkers die meer cursusdagen hebben gevolgd dan het algemene gemiddelde.

Dit is beslist een lastige opgave, dus we gaan dit gefaseerd aanpakken.

Als eerste stap in de richting van het antwoord lossen we de volgende vraag op: hoeveel dagen heeft iedereen cursus gevuld? [Figuur 10.8](#)

geeft het antwoord.

**Figuur 10.8**

275

```

SQL> create or replace view cursusdagen as
  2  select m.mnr, m.naam
  3 ,      sum(c.lengte) as dagen
  4  from inschrijvingen i
  5 ,      cursussen c
  6 ,      medewerkers m
  7 where m.mnr = i.cursist
  8 and c.code = i.cursus
  9 group by m.mnr, m.naam;

```

View created.

```
SQL> select * from cursusdagen;
```

MNR	NAAM	DAGEN
7900	JANSEN	3
7499	ALDERS	11
7521	DE WAARD	1
7566	JANSEN	5
7698	BLAAK	12
7782	CLERCKX	4
7788	SCHOTTEN	12
7839	DE KONING	8
7844	DEN DRAATER	1
7876	ADAMS	9
7902	SPIJKER	9
7934	MOLENAAR	4

```
SQL>
```

Dit is op zich al een redelijk pittige query. Als we nu eens een tabel hadden met dit resultaat als inhoud, zou ons probleem al een stuk eenvoudiger op te lossen zijn. Welnu, daar kunnen we met een view-definitie voor zorgen. We voegen aan bovenstaande query als volgt een eerste regel toe ([zie figuur 10.9](#)).

## Figuur 10.9

Nu is ons probleem betrekkelijk eenvoudig geworden; de oplossing is te zien [in figuur 10.10](#).

```

SQL> select *
  2  from cursusdagen
  3 where dagen > (select avg(dagen)
                      from cursusdagen);

```

MNR	NAAM	DAGEN
7499	ALDERS	11
7698	BLAAK	12
7788	SCHOTΤEN	12
7839	DE KONING	8
7876	ADAMS	9
7902	SPIJKER	9

```
SQL>
```

## Figuur 10.10

Natuurlijk kan deze query ook rechtstreeks worden opgelost op de basistabellen, maar een foutje is snel gemaakt, en overzichtelijk zal de oplossing waarschijnlijk óók niet zijn. We hadden ook inline views kunnen gebruiken, of een subquery kunnen isoleren met de WITH-component (zie het vorige hoofdstuk). Een extra voordeel van het toepassen van een view is dat hij wellicht bij andere queries nog van pas kan komen, zolang we hem niet verwijderen. Ruimte kost een view nauwelijks, en van redundantie is geen sprake.

Door middel van views kan het aanzien van de database worden gewijzigd, zonder dat de onderliggende databasestructuur veranderd hoeft te worden. We noemen dat ook wel *logische*

*gegevensafhankelijkheid*. Dit is een tweede belangrijke toepassing van views. Bijvoorbeeld: verschillende gebruikers kunnen verschillende views op dezelfde gegevens hebben. Gegevens kunnen anders in kolommen worden geordend; entiteiten en/of attributen kunnen een andere naam krijgen.

In gedistribueerde databases wordt ook vaak van views gebruikgemaakt.

Een view die in de lokale database is gedefinieerd en opgeslagen kan bijvoorbeeld – geheel transparant voor de gebruiker of applicatie –

tabellen benaderen in andere databases op het netwerk.

Ook afleidbare gegevens kunnen via een view worden gepresenteerd; een soort *redundantie* op *logisch niveau*. De view die we zojuist hebben gemaakt, is daar een goed voorbeeld van, het aantal cursusdagen wordt daarin immers afgeleid.

277

```
SQL> create or replace view crs_uitvoeringen as
  2  select u.cursus, c.omschrijving, u.begindatum
  3  from   uitvoeringen u
  4 ,      cursussen c
  5 where  u.cursus = c.code;
```

View created.

```
SQL>
```

Views vormen ten slotte ook nog een krachtig hulpmiddel bij de *beveiliging* van gegevens. Door middel van views kunnen bepaalde gegevens voor gebruikers worden afgeschermd; de query die aan de view ten grondslag ligt, bepaalt namelijk precies welke rijen en/of kolommen van de basistabellen worden getoond.

Met behulp van de commando's GRANT en REVOKE kan vervolgens zeer gedetailleerd worden geregeld welke acties op deze gegevens toegestaan zijn. In een dergelijke situatie worden géén rechten gegeven op de basistabellen; die mogen natuurlijk niet rechtstreeks benaderd worden, anders heeft deze beveiliging weinig zin.

### 10.3 Datamanipulatie via views

Views zijn dus virtuele tabellen. De beperkingen van views manifesteren zich op het gebied van de datamanipulatie-mogelijkheden.

We bekijken de volgende drie view-definities (zie [de figuren 10.11 en 10.12](#)).

## Figuur 10.11

278

```
SQL> create or replace view emp as
  2  select mnr, naam, voorl
  3  from   medewerkers;

View created.

SQL> create or replace view avg_evaluaties as
  2  select cursus
  3 ,      avg(evaluatie) as beoordeling
  4  from   inschrijvingen
  5  group  by cursus;

View created.

SQL>
```

## Figuur 10.12

Het is niet altijd mogelijk om op dit soort views datamanipulatie te plegen, om de eenvoudige reden dat vaak niet eenduidig is vast te stellen op welke rij(en) of kolom(men) van de basistabel(len) een manipulatie betrekking heeft. In principe zouden we in staat moeten zijn rijen te verwijderen via view EMP, of wijzigingen aan te brengen in de geselecteerde kolomwaarden. Rijen toevoegen is uitgesloten omdat er NOT NULL-kolommen (zoals de geboortedatum) buiten de view liggen ([zie figuur 10.13](#)).

279

```

SQL> delete from emp
  2  where mnr = 7654;

1 row deleted.

SQL> update emp
  2  set naam = 'BLAKE'
  3  where mnr = 7698;

1 row updated.

SQL> insert into emp
  2  values(7999,'NIEUWMANS','NN');
insert into m
*
ERROR at line 1:
ORA-01400: cannot insert NULL into ("BOEK"."MEDEWERKERS"."GDDATUM")

SQL> rollback;

Rollback complete.

SQL>

```

```

SQL> delete from crs_uitvoeringen where cursus='ERM';

1 row deleted.

SQL> insert into crs_uitvoeringen (cursus, begindatum)
  2                      values ('OAG' , sysdate );

1 row created.

SQL> rollback;

Rollback complete.

SQL>

```

## Figuur 10.13

De view CRS\_UITVOERINGEN (zie figuur 10.11) is gebaseerd op een join van twee tabellen, UITVOERINGEN en CURSUSSEN. Toch zijn we in staat om zelfs via deze view datamanipulatie te plegen. Oracle ondersteunt

namelijk *updatable join-views*, waarvan CRS\_UITVOERINGEN een voorbeeld is. Dit betekent dat Oracle in staat is om de

datomanipulatiecommando's intern te vertalen naar corresponderende acties op de onderliggende tabellen. We komen dus steeds dichter in de buurt van de realisatie van regel 6 van Ted Codd ([zie paragraaf 1.7](#)).

280

### Figuur 10.14

Met betrekking tot updatable join-views bestaan er diverse regels.

Allereerst kan per commando bijvoorbeeld maar één basistabel tegelijk gemuteerd worden. Bovendien speelt het concept van de *key preserved* tabel een belangrijke rol. Het woord zegt het al: dit is een tabel waarmee op rij-nivo een één-op-één-verband bestaat met de view, via de primaire sleutel.

We kunnen bijvoorbeeld rijen via een join-view verwijderen, mits in de join exact één *key preserved* tabel voorkomt, en we kunnen via een join-view alleen kolommen updaten van een *key preserved* tabel.

In de view B ([zie figuur 10.12](#)) komt een GROUP BY-component voor. Dat betekent dat er geen één-op-één-verband meer bestaat tussen de rijen van de view en die van de tabel; via deze view is datomanipulatie daarom uitgesloten. Overigens heeft SELECT DISTINCT in een view-definitie hetzelfde effect.

De verzamelingsoperatoren UNION, MINUS en INTERSECT in een view-definitie hebben ook als effect dat datomanipulatie uitgesloten is. Als we bijvoorbeeld een rij zouden willen toevoegen via een view gebaseerd op een UNION, in welke basistabel zou die rij dan moeten worden ondergebracht?

Als een view met de optie WITH READ ONLY is gecreëerd, is datomanipulatie op die view vanzelfsprekend onmogelijk.

In de Oracle-documentatie staan de exacte regels wat betreft de

‘updatability’ van views. De meeste regels zijn nogal vanzelfsprekend, terwijl men – al experimenterend – vanzelf wel tegen verhelderende foutmeldingen aanloopt.

Ten slotte hebben we wat dit betreft nog een nuttige tabel in de datadictionary; [in figuur 10.15 zien](#) we bijvoorbeeld dat er met de kolom OMSCHRIJVING van de view U weinig toegestaan is. Het is dan ook een kolom die afkomstig is van een niet-key preserved tabel.

281

```
SQL> select column_name
  2 ,      updatable, insertable, deletable
  3 from   user_updatable_columns
  4 where  table_name = 'CRS_UITVOERINGEN';

COLUMN_NAME          UPD INS DEL
-----  -----  -----
CURSUS              YES YES YES
OMSCHRIJVING        NO  NO  NO
BEGINDATUM          YES YES YES

3 rows selected.

SQL>
```

### Figuur 10.15

Het is jammer dat we PL/SQL niet kunnen bespreken binnen het bestek van dit boek, maar hier moet deze taal toch weer even worden genoemd.

In Oracle kunnen op views namelijk *instead of*-triggers worden gedefinieerd, die als het ware de controle overnemen zodra er datomanipulatiecommando’s op de view worden losgelaten. Zodoende zijn we in staat om iedere view updatable te maken. De

verantwoordelijkheid ligt dan natuurlijk ook bij ons om in die triggers de juiste acties te ondernemen.

## 10.4 De CHECK OPTION

Als datamanipulatie via een view mogelijk is, kunnen zich twee vreemde situaties voordoen:

1

Via een INSERT-commando op een view wordt een rij aan de tabel toegevoegd, die daarna niet via dezelfde view te raadplegen is.

2

Via een UPDATE worden rijen in de onderliggende tabel zodanig gewijzigd dat ze uit het beeld van de view verdwijnen.

Hebben we de view DEPT20\_V nog ([zie figuur 10.6](#))? Kijk dan eens wat er gebeurt in de figuren 10.16 en 10.17.

282

```
SQL> update dept20_v
  2  set      afd = 30
  3  where   functie='TRAINER';

4 rows updated.

SQL> select * from dept20_v;

  MNR NAAM      VOORL FUNCTIE      CHEF GBDATUM      MAANDSAL COMM AFD
----- ----- -----
  7566 JANSEN    JM    MANAGER      7839 02-04-1967      2975       20
1 row selected.

SQL> rollback;

Rollback complete.

SQL>
```

```

SQL> insert into dept20_v
  2  values (9999,'BOS','D', null, null
  3          ,to_date('01-JAN-1939','DD-MON-YYYY')
  4          ,'10', null, 30);

1 row created.

SQL> select * from dept20_v;

MNR NAAM      VOORL FUNCTIE    CHEF GBDATUM    MAANDSAL COMM AFD
----- ----- -----
7369 SMIT       N     TRAINER    7902 17-12-1965      800    20
7566 JANSEN     JM    MANAGER    7839 02-04-1967     2975    20
7788 SCHOTTEN   SCJ   TRAINER    7566 26-11-1959     3000    20
7876 ADAMS      AA    TRAINER    7788 30-12-1966     1100    20
7902 SPIJKER    MG    TRAINER    7566 13-02-1959     3000    20

5 rows selected.

SQL> rollback;

Rollback complete.

SQL>

```

## Figuur 10.16

De wijzigingen worden blijkbaar doorgevoerd in de tabel medewerkers, hoewel de rijen niet (of niet meer) voldoen aan de voorwaarde om in het view-resultaat te verschijnen.

## Figuur 10.17

283

```

SQL> create or replace view dept20_v as
  2  select * from medewerkers where afd = 20
  3  with check option;

View created.

SQL> insert into dept20_v
  2  values (9999,'BOS','D', null, null
  3      ,to_date('01-JAN-1939','DD-MON-YYYY')
  4      ,'10', null, 30);
  5      ,to_date('01-JAN-1939','DD-MON-YYYY')

ERROR at line 3:
ORA-01402: view WITH CHECK OPTION where-clause violation

SQL>

```

```

SQL> create or replace view v_inschr as
  2  select i.*
  3  from inschrijvingen i
  4  where i.cursist in (select mnr
  5                      from medewerkers)
  6  and   i.cursus    in (select code
  7                      from cursussen)
  8  and   i.evaluatie in (1,2,3,4,5)
  9  with check option;

View created.

SQL>

```

Als een view wordt gedefinieerd met de toevoeging WITH CHECK OPTION, zullen bovenstaande datamanipulaties een foutmelding opleveren ([zie figuur 10.18](#)).

### Figuur 10.18

Nu lukt dus niet meer wat [in figuur 10.17 nog wel](#) mogelijk was.

In de tijd dat Oracle nog geen referentiële integriteit ondersteunde (dat is heel lang geleden, vóór Oracle 7) konden we gegevens toch gedeeltelijk door

middel van views met CHECK OPTION bewaken, door in de view-definitie een subquery op te nemen. Een voorbeeld daarvan is te zien in [figuur 10.19](#).

## Figuur 10.19

284

```
SQL> update ( select m.maandsal
  2           from medewerkers m
  3           , afdelingen a
  4           where m.afd = a.anr
  5           and locatie = 'DE MEERN')
  6   set maandsal = maandsal + 1;

5 rows updated.

SQL> rollback;
Rollback complete.

SQL>
```

Via deze view zullen alleen inschrijvingen doorgelaten worden van een bestaande medewerker voor een bestaande cursus, met een geldig evaluatiecijfer. Iedere vorm van datamanipulatie (in dit geval INSERT of UPDATE) die een van de drie regels schendt zal een foutmelding opleveren.

## 10.5 Datamanipulatie via inline views

Inline views zijn subqueries die in de rol van een tableexpressie in SQL-commando's voorkomen. Met andere woorden, we gebruiken een subquery (tussen haakjes) op een plaats in een SQL-commando waar we normaal gesproken een tabelnaam of view-naam zouden gebruiken.

Inline views hebben we in het vorige hoofdstuk al besproken (zie [paragraaf 9.3](#)) maar daarbij hebben we alleen queries in beschouwing genomen.

We kunnen inline views ook gebruiken voor datamanipulatie-

doeleinden, zoals we in deze paragraaf zullen zien. Datamanipulatie via inline views is vooral interessant in combinatie met updatable join-views. We zien daar een voorbeeld van [in figuur 10.20](#).

### Figuur 10.20

We kunnen dus datamanipulatie plegen via een inline join-view die de filtering voor ons doet; een filtering die anders met een ingewikkelde (gecorreleerde) subquery in de WHERE-component van het DML-commando zou moeten worden [ingebowd. In figuur 10.20 geven we](#)

alle medewerkers in De Meern een (symbolische) salarisverhoging.

285

Het lijkt wat vreemd om datamanipulatie te plegen via dit soort subqueries, maar het aantal mogelijkheden is vrijwel onbeperkt. De syntax is elegant, en de performance zal over het algemeen beter zijn dan de corresponderende commando's die gebruikmaken van een gecorreleerde subquery.

Vanzelfsprekend hebben we ook hier weer te maken met de [in paragraaf 10.3 genoemde beperkingen van updatable join-views](#).

## 10.6 Views en performance

In principe worden queries via views op de volgende manier verwerkt: 1

Oracle merkt op dat er sprake is van een view.

2

De view-definitie wordt uit de datadictionary opgehaald.

3

Deze view-definitie wordt met onze query samengevoegd.

4

Dit commando (op de basistabellen) wordt door de optimizer onder handen

genomen en daarna uitgevoerd.

Feitelijk vormen alleen de stappen 2 en 3 enige overhead. Het voordeel van deze benadering is dat we optimaal kunnen profiteren van bijvoorbeeld indexen op de onderliggende tabellen. De volgende query op view AVG\_EVALUATIES:

SQL> **select \***

2 **from avg\_evaluaties**

3 **where beoordeling >= 4**

zal bijvoorbeeld intern worden omgebouwd en uitgevoerd als: SQL>

**select**

**i.cursus**

2

,

**avg(i.evaluatie) as beoordeling**

3

**from**

**inschrijvingen i**

4

**group**

**by i.cursus**

5

**having**

**avg(i.evaluatie) >= 4**

Met name als het gaat om grotere tabellen zal het performance-verlies verwaarloosbaar klein zijn. Als we views op views op views gaan definiëren, wordt het natuurlijk een ander verhaal. En als we het dan toch niet helemaal vertrouwen, hebben we nog altijd diagnostic tools zoals AUTOTRACE ([zie paragraaf 7.6](#)) om performance-metingen te verrichten.

286

```
SQL> create view dept20_v as
  2  select * from medewerkers where afd=20;

View created.

SQL> create table dept20_t as
  2  select * from medewerkers where afd=20;

Table created.

SQL> create materialized view dept20_mv as
  2  select * from medewerkers where afd=20;

Materialized view created.
```

## 10.7 Materialized views

Hoewel de behandeling van materialized views eigenlijk buiten het bestek van dit boek valt, mag een korte introductie van materialized views in een hoofdstuk over views niet ontbreken.

Materialized views worden in het bijzonder toegepast in data warehouse-omgevingen, waarin zeer grote hoeveelheden gegevens voornamelijk worden geraadpleegd en zelden of nooit ad hoc worden gewijzigd, en waarbij het volume van de gegevens performance-problemen veroorzaakt.

We zullen met opzet een zo eenvoudig mogelijk voorbeeld kiezen. We halen [figuur 10.6 even terug](#), en voegen er een derde commando aan toe; zie [figuur](#)

## 10.21.

### Figuur 10.21

We kennen inmiddels het verschil tussen een tabel en een view, maar wat is een materialized view? Dat is een view waarvan de queryresultaten wel worden opgeslagen. Er is dus sprake van redundante opslag van gegevens. De materialized view DEPT20\_MV bevat nu alle medewerkers van afdeling 20.

Materialized views hebben twee belangrijke en bijzondere eigenschappen, respectievelijk op het gebied van hun onderhoud en hun gebruik.

287

*Onderhoud:* Materialized views zijn ‘snapshots’, dat wil zeggen dat ze op een bepaald moment in de tijd zijn gevuld (of ververst) op basis van ‘echte’ tabellen. Dat betekent dat de gegevens in een materialized view in de loop van de tijd kunnen gaan achterlopen, omdat de gegevens van de onderliggende tabellen worden gewijzigd. Gelukkig biedt Oracle diverse voorzieningen om het verversen van materialized views volledig (en zo efficiënt mogelijk) te automatiseren, zonder dat we daar nog naar hoeven om te kijken. Met andere woorden: er is sprake van redundantie, maar de redundantie kan betrekkelijk eenvoudig worden gecontroleerd.

*Gebruik:* De Oracle optimizer, het onderdeel van het DBMS dat beslist hoe SQL-commando’s worden uitgevoerd, is op de hoogte van het bestaan van materialized views. De optimizer weet ook of de gegevens van bepaalde materialized views ‘up-to-date’ zijn of dat ze achterlopen.

De optimizer kan deze kennis gebruiken om queries die tegen basistabellen zijn geschreven, te vervangen door queries tegen de materialized views, als de optimizer denkt dat dat sneller resultaat oplevert.

Met andere woorden, om het eenvoudige voorbeeld van het begin van deze paragraaf nog even voort te zetten: we geven het volgende SQL-commando:

SQL> **select \* from medewerkers where afd = 20 and functie = 'TRAINER'**

... en de optimizer voert achter de schermen in plaats daarvan het volgende commando uit:

SQL> **select \* from dept20\_mv where functie = 'TRAINER'**

Deze werkwijze wordt in de documentatie ‘query rewrite’ genoemd.

Omdat de materialized view DEPT20\_MV minder gegevens bevat dan de tabel MEDEWERKERS (en daarmee sneller in het geheugen kan worden ingelezen), is het een beter startpunt om tot het gewenste eindresultaat te

[komen. Zonder in te gaan op de details zien we dit gedrag in figuur](#)

[10.22 aan het werk:](#)

```
SQL> select * from medewerkers where afd=20 and functie='TRAINER';
```

MNR	NAAM	VOORL	FUNCTIE	CHEF	GBDATUM	MAANDSAL	COMM	AFD
7369	SMIT	N	TRAINER	7902	17-DEC-1965	800		20
7788	SCHOTDEN	SCJ	TRAINER	7566	26-NOV-1959	3000		20
7876	ADAMS	AA	TRAINER	7788	30-DEC-1966	1100		20
7902	SPIJKER	MG	TRAINER	7566	13-FEB-1959	3000		20

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		3	120	3 (0)	00:00:01
*1	MAT_VIEW REWRITE ACCESS FULL	DEPT20_MV	3	120	3 (0)	00:00:01

Predicate Information (identified by operation id):

```
1 - filter("DEPT20_MV"."FUNCTIE"='TRAINER')
```

```
SQL>
```

## Figuur 10.22

Hoewel we een query formuleren op de tabel MEDEWERKERS, laat het plan zien dat de materialized view DEPT20\_MV wordt geraadpleegd, en daarna wordt gefilterd op functie.

Nog twee opmerkingen over materialized views tot slot:

Het voorbeeld in deze paragraaf is natuurlijk veel te eenvoudig.

Normaal gesproken worden materialized views gemaakt met daarin relatief ‘dure’ operaties zoals aggregatie (GROUP BY), joins over diverse tabellen, verzamelingsoperatoren (UNION, MINUS,

INTERSECT); met andere woorden, operaties die bij herhaalde uitvoering erg tijdrovend zijn.

We krijgen via materialized views weliswaar sneller antwoord op onze

queries, maar we lopen daarbij het risico dat de resultaten zijn gebaseerd op verouderde gegevens omdat de materialized views niet up-to-date zijn. We kunnen zelf aangeven of we dat acceptabel vinden of niet, en daarmee het gedrag van de optimizer beïnvloeden.

Willen we exacte antwoorden, dan zal ‘query rewrite’ alleen plaatsvinden als de materialized views gegarandeerd up-to-date zijn.

Voor verdere details over materialized views, zie de *Data Warehousing Guide*.

```

SQL> select * from sal_historie;

      MNR D_IN_DIENST BEOORDELING SALARIS_VERHOGING
----- ----- -----
    7369 01-JAN-2000 01-JAN-2000
    7369 01-JAN-2000 01-FEB-2000          -150
    7499 01-JUN-1988 01-JUN-1988
    7499 01-JUN-1988 01-JUL-1989          300
    7499 01-JUN-1988 01-DEC-1993          200
    7499 01-JUN-1988 01-OCT-1995          200
    7499 01-JUN-1988 01-NOV-1999         -100
    ...
    7934 01-FEB-1998 01-FEB-1998
    7934 01-FEB-1998 01-MAY-1998          5
    7934 01-FEB-1998 01-FEB-1999          10
    7934 01-FEB-1998 01-JAN-2000          10

79 rows selected.

SQL>

```

## 10.8 Opgaven

1

Kijk nog eens terug naar het voorbeeld in de figuren 10.8, 10.9 en 10.10. Hoe zou je de query [in figuur 10.10](#) met behulp van de WITH-operator kunnen formuleren, zonder gebruik te maken van een view?

2

[Zie figuur 10.13. Hoe](#) is het mogelijk dat we de medewerker met nummer 7654 kunnen verwijderen via view EMP? Er zijn toch rijen in de HISTORIE-tabel die via een refererende sleutel verwijzen naar die medewerker?

3

Maak een view die het volgende resultaat oplevert, gebaseerd op de HISTORIE-tabel: Voor iedere medewerker zien we de datum van indiensttreding, de datums van de beoordelingsmomenten, en de salarisverhogingen (of verlagingen) als gevolg van die

beoordelingen.

290

## **Hoofdstuk 11**

### **SQL\*Plus en SQL Developer**

In [hoofdstuk 2](#) hebben we al kennismegemaakt met een aantal mogelijkheden van SQL\*Plus en SQL Developer. Toen hebben

we de meest noodzakelijke commando's behandeld om aan de

slag te kunnen, zoals:

- De editor-commando's (LIST, INPUT, CHANGE, APPEND, DEL, EDIT).
- Bestandsbeheer (SAVE, GET, START, SPOOL).
- Overige commando's (HOST, DESCRIBE, HELP).

In dit hoofdstuk gaan we wat dieper in op enkele mogelijkheden van SQL\*Plus, die het werken met deze tool verder zullen

veraangenamen.

Paragraaf 11.1 behandelt de mogelijkheden met betrekking tot SQL\*Plus-variabelen. Daarbij zijn SET, SHOW, DEFINE en ACCEPT

de voornaamste SQL\*Plus-commando's.

Vervolgens bespreken we hoe we SQL\*Plus niet alleen interactief kunnen gebruiken, maar ook kunnen aanspreken vanuit een

bestand. We zullen dit soort bestanden met de term 'script' aanduiden.

In de laatste paragraaf zullen we zien hoe we SQL\*Plus als een rapportage-tool kunnen gebruiken, door de lay-out van het

resultaat op te maken met SQL\*Plus-commando's zoals TTITLE, BTITLE en COLUMN. In de vierde paragraaf komen de commando's BREAK en COMPUTE aan de orde, waarmee we onze rapporten nog verder kunnen verfraaien.

## 11.1 SQL\*Plus versus SQL Developer

We hebben eerder [in hoofdstuk 2 al](#) kennis gemaakt met de SQL\*Plus en SQL Developer tools. Daar is toen ook al aangegeven dat het belangrijkste verschil tussen deze twee tools is, dat SQL\*Plus een regelgeoriënteerde commando-omgeving biedt, en SQL Developer een grafische omgeving. Er zijn nog enkele andere verschillen die we hier willen noemen.

291

- 
- 
- 
- 
- 
- 
- 
- 
- 
- 

SQL\*plus:

is een tool die al meer dan 20 jaar met de Oracle-database meegeleverd wordt;

is een tool die vaak door de DBA op de database-server zelf gebruikt wordt om database-beheeractiviteiten mee uit te voeren;

is een tool die gebruikt kan worden voor het uitvoeren van ad hoc SQL-commando's;

*i*s een tool die gebruikt kan worden om eenvoudige rapportages te produceren.

SQL Developer:

is een vrij nieuwe tool;

is een tool die vaak door developers gebruikt wordt om SQL (en PL/SQL) mee te ontwikkelen;

is een tool die veel meer functionaliteiten biedt dan het

uitvoeren/ontwikkelen van ad hoc SQL en het produceren van eenvoudige rapportages.

Omdat SQL\*Plus al zo lang bestaat, is het in ‘het Oracle-wereldje’ een bijna de facto standaardtool waarmee veel DBA’s en developers bekend zijn, en waarmee ook veel scripts gemaakt zijn die dagelijks in productie-Oracle-databaseomgevingen gerund worden. In dit hoofdstuk zullen we dieper ingaan op de mogelijkheden van SQL\*Plus en met name hoe je *scripts* kunt maken die door SQL\*Plus uitgevoerd kunnen worden. Toen Oracle SQL Developer is gaan bouwen, heeft men zich gerealiseerd dat één van de vereisten van deze nieuwe tool moest zijn dat alles wat men in SQL\*Plus kan doen, ook mogelijk moet zijn in SQL Developer.

De additionele mogelijkheden van SQL\*Plus die we in dit hoofdstuk gaan behandelen, zullen we steeds illustreren in figuren waarin we ook SQL\*Plus gebruiken. Alle voorbeelden kunnen echter ook in SQL

Developer uitgevoerd worden door de commando’s in de SQL

Worksheet te typen en vervolgens op het kleinere groene driehoekje (Run Script) bovenaan in het worksheet window te klikken.

## 11.2 SQL\*Plus-variabelen

SQL\*Plus kent drie soorten variabelen:

1

substitutievariabelen;

```

SQL> select * from afdelingen
2 where naam like upper('%&letter%');

Enter value for letter: a
old  2: where naam like upper('%&letter%')
new  2: where naam like upper('%a%')

ANR NAAM          LOCATIE      HOOFD
-----  -----
10 HOOFDKANTOOR   LEIDEN        7782
40 PERSONEELSZAKEN GRONINGEN    7839

SQL>

```

2

gebruikersvariabelen;

3

systeemvariabelen.

Substitutievariabelen worden in SQL-commando's gebruikt. Bij het uitvoeren van het SQL-commando wordt dan om een waarde gevraagd.

Substitutievariabelen onderscheiden zich van de andere twee soorten doordat hun waarden nergens worden vastgelegd: als hetzelfde SQL-commando nogmaals wordt uitgevoerd, wordt opnieuw om een waarde gevraagd. Sterker nog: als een substitutievariabele binnen één SQL-commando tweemaal voorkomt, wordt ook tweemaal om een waarde gevraagd (zie bijvoorbeeld [figuur 11.4](#)).

### 11.2.1 Substitutievariabelen

Het karakter waarmee SQL\*Plus verwijzingen naar een

substitutievariabele herkent, is de ampersand (& ), ook wel het DEFINE-karakter genoemd. Bekijk eens wat er gebeurt in [figuur 11.1](#).

## Figuur 11.1

De punt (.) kunnen we gebruiken om het einde van de naam van een variabele in een commando duidelijk te markeren, zoals bijvoorbeeld in

[figuur 11.3. De punt](#) (.) heet in SQL\*Plus het CONCAT-karakter. Zowel het DEFINE- als het CONCAT-karakter kunnen we overigens opvragen met het commando SHOW (en eventueel wijzigen met het commando SET); zie

[figuur 11.2.](#)

293

```
SQL> show define
define "&" (hex 26)

SQL> show concat
concat "." (hex 2e)
```

```
SQL> select '&drank.glas' from dual;

Enter value for drank: champagne
old   1: select '&drank.glas' from dual
new   1: select 'champagneglas' from dual

'CHAMPAGNEGLA
-----
champagneglas

SQL>
```

```
SQL> select naam from medewerkers
  2  where mnr between &x and &x+100;

Enter value for x: 7500
Enter value for x: 7500
old   2: where mnr between &x and &x+100
new   2: where mnr between 7500 and 7500+100

NAAM
-----
DE WAARD
JANSEN

SQL>
```

**Figuur 11.2**

**Figuur 11.3**

**Figuur 11.4**

Als we het niet nodig vinden om steeds te zien hoe de

substitutievariabelen zijn vervangen (zoals in de figuren 11.1, 11.3 en 11.4) kunnen we dat onderdrukken met de VERIFY-setting van SQL\*Plus; zie [figuur 11.5](#).

294

```
SQL> set verify on
SQL> set verify off
SQL> show verify
verify OFF
```

```
SQL> select naam from medewerkers
  2  where mnr between &x and &x+100;
```

```
Enter value for x: 7500
```

```
Enter value for x: 7500
```

```
NAAM
```

```
-----
```

```
DE WAARD
```

```
JANSEN
```

```
SQL>
```

```
SQL> define x=7500
```

```
SQL> select naam from medewerkers
  2  where mnr between &x and &x+100;
```

```
NAAM
```

```
-----
```

```
DE WAARD
```

```
JANSEN
```

```
SQL>
```

## Figuur 11.5

Als we de VERIFY-setting aldus hebben veranderd en het commando in [de SQL-buffer met het commando RUN nogmaals uitvoeren \(zie figuur 11.6\)](#), krijgen we de ‘old:’ en ‘new:’ regels niet meer te zien.

## Figuur 11.6

### 11.2.2 Gebruikersvariabelen

Als we de waarde van een variabele tijdelijk willen vastleggen om er daarna meer dan één keer gebruik van te kunnen maken, komen we in de volgende categorie terecht: de *gebruikersvariabelen*.

[We gebruiken daarbij het SQL\\*Plus-commando DEFINE \(zie figuur](#)

[11.7\).](#)

295

```
SQL> def x
DEFINE X          = "7500"  (CHAR)
SQL> def
DEFINE _DATE      = "08-FEB-2004" (CHAR)
DEFINE _CONNECT_IDENTIFIER = "orcl" (CHAR)
DEFINE _USER       = "BOEK"   (CHAR)
DEFINE _PRIVILEGE = ""      (CHAR)
DEFINE _SQLPLUS_RELEASE = "1001000200" (CHAR)
DEFINE _EDITOR     = "vi"    (CHAR)
DEFINE _O_VERSION  = "Oracle Database 10g Enterprise Edition Release
10.1.0.2.0 - Production
With the Partitioning, OLAP and Data Mining options" (CHAR)
DEFINE _O_RELEASE  = "1001000200" (CHAR)
DEFINE X          = "7500"  (CHAR)
SQL> undefine x
SQL>
```

## Figuur 11.7

Door het DEFINE-commando hebben we de gebruikersvariabele X met zijn waarde vastgelegd. Daarom wordt er [in figuur 11.7 na het](#)

commando RUN niet meer om een waarde voor X gevraagd.

Met het commando DEFINE kunnen we niet alleen nieuwe waarden aan een variabele toewijzen, maar ook bestaande waarden opvragen. We kunnen één specifieke variabele opvragen, of een overzicht krijgen van alle gedefinieerde variabelen. Met het UNDEFINE-commando kunnen we een gedefinieerde variabele verwijderen ([zie figuur 11.8](#)).

## Figuur 11.8

Er is een tussenvorm mogelijk tussen *substitutievariabelen* en *gebruikersvariabelen*. Dit zijn variabelen die in een SQL-commando voorkomen, waarbij alleen de eerste keer om een waarde wordt gevraagd. Dit bereiken we door een dubbele ampersand te gebruiken (&&). Door wat te experimenteren zien we dat er impliciet een DEFINE

wordt uitgevoerd, zoals in [figuur 11.9](#).

296

```
SQL> define mnr
SP2-0135: symbol mnr is UNDEFINED

SQL> select * from medewerkers
  2 where mnr between &&mnr and &mnr+100;

Enter value for mnr: 7500

      MNR  NAAM        VOORL  FUNCTIE      CHEF  GBDATUM      MAANDSAL  COMM  AFD
-----  -----  -----  -----  -----  -----  -----  -----  -----
    7521  DE WAARD    TF    VERKOPER    7698  22-FEB-1962    1250    500    30
    7566  JANSEN     JM    MANAGER    7839  02-APR-1967    2975          20

SQL> define mnr
DEFINE MNR                  = "7500"  (CHAR)
SQL>
```

```
SQL> prompt Dit is een demonstratie.
Dit is een demonstratie.
SQL> pause Geef Enter...
Geef Enter...

SQL> accept x number -
> prompt "Geef een waarde voor x: "
Geef een waarde voor x: 42

SQL> define x
DEFINE X                  =          42  (NUMBER)
SQL>
```

## Figuur 11.9

Er wordt nu nog maar één keer naar een waarde gevraagd, en als we de inhoud van de SQL-buffer nogmaals uitvoeren (met / of RUN) wordt zelfs helemaal niets meer gevraagd.

Naast het gebruik van ampersands is er ook een vriendelijker manier om gebruikersvariabelen te definiëren, waarbij we tevens invloed kunnen uitoefenen op de manier waarop naar een waarde wordt gevraagd. Dit is

vooral plezierig vanuit SQL\*Plus-scripts, die in de volgende paragraaf aan de orde komen. We gebruiken daartoe de commando's PROMPT, PAUSE en ACCEPT (zie [figuur 11.10](#)).

## Figuur 11.10

Met het commando PROMPT kunnen we tekst naar het scherm schrijven, met PAUSE kunnen we de verwerking even onderbreken, en met ACCEPT

297

```
SQL> show pagesize
pagesize 36
SQL> show pause
pause is OFF
SQL> set pause '[Enter]... '
SQL> set pause on
SQL> set pagesize 10

SQL> select * from medewerkers;
...
SQL> show all
...
```

hebben we invloed op het datatype van de variabele en op de tekst waarmee om een waarde wordt gevraagd. Probeer in [figuur 11.10](#) maar eens een niet-numerieke waarde voor X in te geven.

Merk overigens ook op hoe we een SQL\*Plus-commando over twee regels kunnen verdelen: door de eerste regel met een minteken (–) te laten eindigen. Als we aan het einde van een regel als laatste karakter een minteken gebruiken, dan verliest de [Enter] zijn bijzondere betekenis. Let wel: we hebben het hier over SQL\*Plus-commando's die normaal gesproken slechts één regel beslaan, in tegenstelling tot SQL-commando's.

### 11.2.3 Systeemvariabelen

De derde soort variabelen in SQL\*Plus zijn de *systeemvariabelen*. De waarden van deze voorgedefinieerde variabelen bepalen voor een groot deel het gedrag van SQL\*Plus. We hebben er al diverse voorbeelden van gezien,

zoals PAGESIZE en PAUSE [in hoofdstuk 2](#).

Bij het werken met de SQL\*Plus-systeemvariabelen zijn de commando's SET en SHOW van belang, respectievelijk om waarden toe te wijzen of op te vragen.

### **Figuur 11.11**

Als we het laatste commando van figuur 11.11 (SHOW ALL) uitvoeren, zien we dat het aantal SQL\*Plus-systeemvariabelen indrukwekkend is.

In het volgende overzicht is dan ook geen enkele poging gedaan om volledig te zijn; alleen de meest gebruikte systeemvariabelen komen erin voor. Waar dat van toepassing is, zijn de default-waarden in de 298

<b>concat</b>	Het einde van de naam van een variabele	(.)
<b>define</b>	Referentie naar waarde van variabelen	(&)
<b>echo</b>	(alleen van toepassing voor scripts)	
	De commando's wel of niet weergeven	(off)
<b>feedback</b>	Geeft de melding '... rows selected', vanaf een bepaald aantal rijen,	(6)
<b>heading</b>	Kolomnamen boven resultaattabel	(on)
<b>linesize</b>	Regel- of schermbreedte in kolommen	(80)
<b>long</b>	Default-breedte LONG kolommen	(80)
<b>newpage</b>	Aantal lege regels na pagina-overgang	(1)
<b>null</b>	Weergave null-waarden op scherm	
<b>numformat</b>	Default-formaat getallenweergave	
<b>numwidth</b>	Default-breedte numerieke kolommen	(10)
<b>pagesize</b>	Aantal regels op een pagina of scherm	(14)
<b>pause</b>	Dosering per pagina	(off)
<b>scan</b>	Al of niet reageren op &-substitutie	(on)
<b>colsep</b>	Aantal spaties tussen kolommen	(1)
<b>sqlprompt</b>	De prompt van SQL*Plus	(SQL>)
<b>sqlterminator</b>	Afsluiten en uitvoeren commando	(;)
<b>timing</b>	Gebruikte tijd na ieder commando	(off)
<b>trimspool</b>	Onderdruk spaties in spool bestanden	(off)
<b>user</b>	Naam waaronder is aangelogd	
<b>verify</b>	Commandoregels voor/na substitutie van variabelen weergeven	(on)

derde kolom tussen haakjes opgenomen.

### Figuur 11.12

Tot slot van deze paragraaf geven we enkele experimenten met deze systeemvariabelen weer in de figuren 11.13 tot en met 11.16. De resultaten van de queries worden niet steeds herhaald; we kunnen zelf aan de resultaten op het scherm zien wat het effect is van de systeemvariabelen.

```

SQL> select * from afdelingen;

ANR NAAM          LOCATIE      HOOFD
-----
10 HOOFDKANTOOR    LEIDEN       7782
20 OPLEIDINGEN    DE MEERN    7566
30 VERKOOP         UTRECHT     7698
40 PERSONEELSZAKEN GRONINGEN  7839

SQL> set feedback 4
SQL> /

ANR NAAM          LOCATIE      HOOFD
-----
10 HOOFDKANTOOR    LEIDEN       7782
20 OPLEIDINGEN    DE MEERN    7566
30 VERKOOP         UTRECHT     7698
40 PERSONEELSZAKEN GRONINGEN  7839

4 rows selected.

SQL> select * from medewerkers;
...
SQL> set feedback off
SQL> show feedback
feedback OFF
SQL> /
...
SQL> set feedback 10
SQL>

```

Figuur 11.13

```
SQL> select * from schalen;
```

SNR	ONDERGRENNS	BOVENGRENNS	TOELAGE
1	700	1200	0
2	1201	1400	50
3	1401	2000	100
4	2001	3000	200
5	3001	9999	500

```
SQL> set colsep 4
```

```
SQL> set numwidth 10
```

```
SQL> /
```

SNR	ONDERGRENNS	BOVENGRENNS	TOELAGE
1	700	1200	0
2	1201	1400	50
3	1401	2000	100
4	2001	3000	200
5	3001	9999	500

```
SQL>
```

```

SQL> set numwidth 5
SQL> set null N.V.T

SQL> select naam, chef, comm
  2  from medewerkers
  3  where afd = 10;

NAAM          CHEF    COMM
-----  -----  -----
CLERCKX      7839  N.V.T
DE KONING    N.V.T  N.V.T
MOLENAAR     7782  N.V.T

SQL> set numformat 09999.99
SQL> select * from schalen;

      SNR ONDERGRENSENTAGE BOVENGRENSENTAGE
-----  -----  -----  -----
 00001.00 00700.00 01200.00 00000.00
 00002.00 01201.00 01400.00 00050.00
 00003.00 01401.00 02000.00 00100.00
 00004.00 02001.00 03000.00 00200.00
 00005.00 03001.00 09999.00 00500.00

SQL>

```

**Figuur 11.14**

**Figuur 11.15**

```
SQL> select 'Jansen&Co' from dual;
Enter value for co: Janssen

'JANSENJANSSE
-----
JansenJanssen

SQL> set define off
SQL> run
 1* select 'Jansen&Co' from dual

'JANSEN&C
-----
Jansen&Co

SQL> set define !
SQL> select 'Jansen&Co' from !tabel;
Enter value for tabel: dual

'JANSEN&C
-----
Jansen&Co

SQL> set define &
SQL>
```

## Figuur 11.16

Als we weer een ‘schone start’ willen maken, kunnen we het beste SQL\*Plus even verlaten en het opnieuw starten; we zijn dan van alle gevolgen van onze experimenten af.

### 11.3 SQL\*Plus-scripts

We weten al dat we SQL-commando’s kunnen bewaren door SAVE te gebruiken. Tot nog toe hebben we steeds slechts één SQL-commando vanuit de buffer naar een bestand geschreven. We kunnen echter ook bestanden aanleggen met een willekeurig aantal SQL-commando’s erin; tussen deze SQL-commando’s kunnen we eventueel ook nog SQL\*Plus-commando’s opnemen. Dit soort bestanden noemen we SQL\*Plus-scripts.

[Dergelijke scripts kunnen we met START of @ aanroepen \(zie de figuren\)](#)

[11.17 en 11.18\).](#)

302

```
SQL> select *
  2  from medewerkers
  3  where afd = &&afdelingsnummer
  4  and functie = upper('&&functie');

...
SQL> save testscript replace
Wrote file testscript.sql
SQL> clear buffer
SQL> start testscript
...
SQL> @testscript
...
```

```
SQL> select *
  2  from afdelingen
  3  where anr = &afdelingsnummer;

...
SQL> save testscript append
Appended file to testscript.sql
SQL> @testscript
...
SQL> get testscript
...
SQL> /
```

**Figuur 11.17**

**Figuur 11.18**

We krijgen nu een foutmelding ([zie figuur 11.19](#)). Dat komt doordat de buffer als gevolg van de laatste GET-opdracht meerdere commando's bevat, en dat kan niet meer met / of RUN worden uitgevoerd. Wat START

(of @) feitelijk doet, is het script regel voor regel lezen en uitvoeren, alsof de regels één voor één interactief werden ingetikt.

303

```
SQL> get testscript
 1 select *
 2 from medewerkers
 3 where afd = &&afdelingsnummer
 4 and functie = upper('&&functie')
 5 /
 6 select *
 7 from afdelingen
 8* where anr = &afdelingsnummer
SQL> /
select *
*
ERROR at line 6:
ORA-00936: missing expression

SQL>
```

## Figuur 11.19

Vandaar dat de slash (/) die altijd door SAVE aan de inhoud van de buffer wordt toegevoegd, essentieel is; let maar op wat er gebeurt als we hem weghalen, met een editor bijvoorbeeld. Het script zal dan wachten op invoer vanaf het toetsenbord, alsof het commando nog niet af is.

Overigens kunnen we SQL\*Plus-scripts niet alleen met START of met @ uitvoeren, maar ook met @@. Het verschil tussen @ en @@ speelt alleen een rol als we scripts vanuit andere scripts aanroepen. In dat geval betekent @@ dat het te starten (sub)script altijd wordt gezocht in dezelfde directory waar het (hoofd)script staat vanwaar het (sub)script wordt aangeroepen.

We gaan nu een extra mogelijkheid van scripts uitproberen: het werken met *parameters*, die we met het aanroepen van het script op de commandoregel kunnen meegeven. We kunnen maximaal negen

parameters gebruiken, die in het script respectievelijk worden aangesproken als &1, &2, ..., &9. Zorg voor de volgende wijzigingen in het testscript dat we net hebben gemaakt ([zie figuur 11.20](#)).

```

REM =====
REM gewijzigde inhoud testscript.sql
REM =====
select *
from medewerkers
where afd = &&1
and functie = upper('&2')
/
select *
from afdelingen
where anr = &1
/
undefine 1

```

```

SQL> @testscript
Enter value for 1: 10
Enter value for 2: manager

      MNR NAAM        VOORL FUNCTIE      CHEF GBDATUM      MAANDSAL COMM AFD
----- ----- -----
    7782 CLERCKX     AB     MANAGER    7839 09-JUN-1965      2450 N.V.T   10

      ANR NAAM          LOCATIE      HOOFD
----- -----
    10 HOOFDKANTOOR      LEIDEN       7782

SQL>

```

## Figuur 11.20

[We kunnen het script nu op twee manieren aanroepen \(zie de figuren 11.21 en 11.22\).](#)

## Figuur 11.21

Hieruit blijkt dat we niet verplicht zijn om parameters op de commandoregel mee te geven; als we ze achterwege laten, worden ze als gewone substitutievariabelen behandeld. Tenminste, als van een eerdere aanroep van een script geen waarden zijn achtergebleven (gebruik het UNDEFINE-commando).

[In figuur 11.22 zien we wat er gebeurt als we bij de aanroep van het script](#)

direct twee waarden meegeven.

305

```
SQL> @testscript 30 verkoper

      MNR NAAM        VOORL FUNCTIE   CHEF    GBDATUM     MAANDSAL COMM   AFD
----- ----- -----
7499 ALDERS       JAM    VERKOPER  7698 20-FEB-1961      1600   300   30
7521 DE WAARD    TF     VERKOPER  7698 22-FEB-1962      1250   500   30
7654 MARTENS     P      VERKOPER  7698 28-SEP-1956     1250  1400   30
7844 DEN DRAAIER JJ    VERKOPER  7698 28-SEP-1968     1500     0   30
ANR NAAM          LOCATIE           HOOFD
-----
30 VERKOOP        UTRECHT          7698
```

```
SQL>
```

```
SQL> set buffer BLA
SQL> input
1  clear screen
2  set verify off
3  set pause off
4  accept afdeling number -
5  prompt "Geef een afdelingsnummer: "
6  select *
7  from afdelingen
8  where anr = &afdeling;
9  select naam, functie, maandsal
10 from medewerkers
11 where afd = &afdeling;
12 undefined afdeling
13 set pause on
14 set verify on
15
```

```
SQL>
```

## Figuur 11.22

We kunnen zoals gezegd ook nog SQL\*Plus-commando's opnemen in het script. Een probleem hierbij is dat SQL\*Plus-commando's niet in de SQL-buffer kunnen worden opgeslagen. Er zijn twee manieren om dit toch voor elkaar te krijgen:

1

Door het gebruik van een externe editor.

2

Door het gebruik van een aparte buffer.

De tweede methode leggen we hier uit ([zie figuur 11.23](#)), omdat die volledig onafhankelijk is van het onderliggende operating system; in een Windows-omgeving is het werken met een editor zoals Notepad zonder meer de handigste oplossing.

### **Figuur 11.23**

306

Sluit het invoeren van tekst in de buffer af door op regel 15 nogmaals een [Enter] te geven. Nu kunnen we het script bewaren en testen: SQL> **save testscript2**

Created file testscript2.sql

SQL> **@testscript2**

...

Door het commando SET BUFFER (de naam BLA is naar keuze) maken we naast de SQL-commandobuffer een extra buffer aan. Hierop kunnen we alle SQL\*Plus-editorcommando's toepassen, en SAVE en GET om bestandsbeheer uit te voeren.

De inhoud van extra buffers kan niet rechtstreeks worden uitgevoerd; RUN en / voeren namelijk altijd de inhoud van de default SQL-buffer uit.

Vandaar dat we het script eerst veilig moeten stellen met behulp van het commando SAVE.

SQL\*Plus-commando's horen normaal gesproken op één regel te staan.

Als dat niet lukt, moeten we expliciet aangeven dat het commando op de volgende regel verdergaat. Dat doen we met een minteken (-), zoals ook in het commando [ACCEPT in figuur 11.23 gebeurt](#).

Vanaf nu geven we alleen nog de inhoud van scripts, en wordt het aan uzelf overgelaten om de weergegeven inhoud in het script te krijgen.

Nogmaals: beschikt u over een editor zoals Notepad onder Windows, gebruik die dan vooral.

Een bijzonder script kan in deze paragraaf niet onvermeld blijven; dat is het script met de naam *login.sql*. Dit script wordt automatisch uitgevoerd bij het starten van SQL\*Plus, als het tenminste in de directory staat van waaruit u SQL\*Plus start ([zie figuur 2.25\) of](#)

anderszins door SQL\*Plus kan worden gevonden, via de SQLPATH-omgevingsvariabele (onder Linux) of registry-setting (onder Windows).

*Let op:* Het script *login.sql* wordt ook uitgevoerd als we zonder SQL\*Plus te verlaten een CONNECT-commando uitvoeren. Het script *login.sql* kunnen we met name gebruiken om allerlei SQL\*Plus-systeemvariabelen een waarde te geven en kolomdefinities te regelen.

Zorg voor de volgende inhoud van het bestand *login.sql*, en test de werking door SQL\*Plus te verlaten en opnieuw te starten:

307

```
REM voorbeeld van een LOGIN.SQL script:  
=====  
set pause      "Enter... "  
set pause      on  
set numwidth 6  
set pagesize 24  
alter session set nls_date_format='dd-mm-yyyy'  
REM define_editor=Notepad    /* voor Windows */  
REM define_editor=vi          /* voor UNIX      */  
clear screen
```

<b>SET LINESIZE</b>	
<b>SET PAGESIZE</b>	Bladspiegel regelen
<b>SET NEWPAGE</b>	
<b>SET TRIMSPPOOL ON</b>	Onderdruk aanvullen regels met spaties
<b>COLUMN</b>	Kolommen opmaken
<b>TTITLE en BTITLE</b>	Kop- en voetregels definiëren
<b>BREAK</b>	Rijen indelen in groepen, en
<b>COMPUTE</b>	Tussenberekeningen toevoegen
<b>SPOOL</b>	Resultaten in een bestand zetten

**Figuur 11.24**

#### 11.4 Rapportage met SQL\*Plus

Met behulp van SQL kunnen we queries formuleren. Deze queries leveren een resultaattabel op. Vaak laat de lay-out van deze ‘platte’ tabellen te wensen over.

SQL\*Plus beschikt over een aantal mogelijkheden om resultaattabellen enigszins te verfraaien tot een toonbaar rapport. Dat zijn (onder andere) de commando’s van figuur 11.25.

**Figuur 11.25**

Het SET-commando kennen we al, en van het COLUMN-commando hebben we al eerder een toepassing gezien. Met dit commando is echter veel meer mogelijk, zoals hierna zal blijken. De algemene gedaante van het COLUMN-commando is als volgt:

SQL> **column [ <kolomnaam> | <expressie> ] [ <optie>...]**

<b>alias</b>	Kolom-alias, vooral nuttig in verdere <b>break-</b> en <b>compute-</b> commando's
<b>clear</b>	Reset alle kolominstellingen
<b>format</b>	Formaat kolomweergave
<b>heading</b>	Definitie van een (andere) kolomtitel
<b>justify</b>	Uitlijning kolomtitel: L(eft), C(enter) of R(ight)
<b>like</b>	Overnemen van instellingen van een andere kolom
<b>newline</b>	Regelovergang vóór deze kolom
<b>new_value</b>	Variabele om laatste kolomwaarde vast te houden
<b>noprint</b>	Onderdrukken van de kolom in de uitvoer
<b>null</b>	Weergave van null-waarden in een kolom
<b>on   off</b>	Instellingen (de)activeren
<b>wrapped</b>	Te lange kolomwaarden gaan door op de volgende regel
<b>word_wrapped</b>	Idem, waarbij tussen twee woorden wordt afgebroken
<b>truncated</b>	Te lange kolomwaarden worden afgekapt

Als we helemaal geen argumenten meegeven, krijgen we een overzicht van alle actuele kolominstellingen. Geven we alleen een kolomnaam, dan zien we de instellingen van die specifieke kolom.

Met behulp van expressie kunnen we kolommen aansturen die in de SELECT-component van de query als een expressie zijn geformuleerd. In dat geval moet de expressie letterlijk van de query worden overgenomen.

Een selectie uit de geldige opties van het COLUMN-commando: **Figuur 11.26**

De laatste drie opties – die overigens net als alle andere opties mogen worden afgekort – sluiten elkaar natuurlijk uit.

Enkele voorbeelden van COLUMN-commando's zien we in de figuren 11.27 tot en met 11.29.

```
SQL> select mnr,naam,gbdatum  
2 ,      maandsal          as salaris  
3 ,      comm            as commissie  
4 from medewerkers;  
...  
SQL> col naam      format a20 hea achternaam jus c  
SQL> col salaris   format $9999.99  
SQL> col commissie like salaris  
SQL> col salaris   heading maand|salaris  
SQL> /  
...
```

```
SQL> col comm noprint    -- NB: let op de kolomnaam!  
SQL> /  
...  
SQL> col commissie nopri  -- NB: let op de kolomnaam!  
SQL> /  
...  
SQL> col commissie off  
SQL> /  
...  
SQL> col commissie  
SQL> col commissie on  
SQL>
```

```
SQL> col mnr new_value BLA  
SQL> /  
...  
SQL> def BLA  
  
SQL> I  
5 where afd = 30;  
  
SQL> def BLA  
SQL> undefined BLA
```

Figuur 11.27

Figuur 11.28

Figuur 11.29

Op deze manier kunnen we de lay-out per kolom regelen. De opmaak per pagina kan in eerste instantie worden geregeld met SET PAGESIZE en SET LINESIZE. Het resultaat kan vervolgens met de commando's TTITLE en BTITLE van een kop- of voetregel worden voorzien. Met dezelfde twee commando's kunnen we het mechanisme ook aan- of 310

```
SQL> set      pagesize 22
SQL> set      linesize 80
SQL> tttitle "OVERZICHT|medewerkers van afdeling 30"
SQL> btitle   "vertrouwelijk"
SQL> /
SQL> btitle off
SQL> btitle
SQL> tttitle off
```

uitzetten, of de huidige instellingen opvragen.

### Figuur 11.30

Op deze manier krijgen we een standaardkopregel, met links de datum en rechts het paginanummer. Het commando TTITLE heeft vele mogelijkheden om hierop te variëren; experimenten daarmee worden aan de lezer overgelaten.

SQL\*Plus kent naast BTITLE en TTITLE ook nog de commando's REPHEADER en REPFOOTER. Zie de documentatie voor details.

Uit het gegeven voorbeeld blijkt dat we de verticale streep (|) kunnen gebruiken om een regelovergang te forceren. In een van de vorige voorbeelden kwam dit karakter ook al voor binnen het COLUMN-commando. Het wordt in SQL\*Plus ook wel het HEADSEP-karakter genoemd, en is in principe instelbaar, net als alle andere karakters die binnen SQL\*Plus een bijzondere betekenis hebben.

Als we nog eens naar [figuur 11.23](#) kijken, zien we dat dat script eigenlijk al een vrijwel volledig rapport produceert. Met behulp van het SPOOL-commando kunnen we het resultaat in een bestand wegschrijven, zodat het bijvoorbeeld kan worden geprint. Pas daartoe het script testscript2.sql aan zoals in [figuur 11.31](#) is weergegeven.

```

...
4 accept afdeling number -
5 prompt "Geef een afdelingsnummer: "
  set trimspool on
  spool rapport.txt
6 select *
7 from afdelingen
...
11 where afd = &afdeling;
  spool off
12 undefine afdeling
13 set pause on
14 set verify on

```

### Figuur 11.31

## 11.5 BREAK en COMPUTE

We kunnen het resultaat van een rapport voorzien van zogeheten

'breaks' met behulp van het BREAK-commando. Breaks zijn punten in het rapport die we in het eindresultaat kunnen accentueren door repeterende kolomwaarden te onderdrukken, een extra regel over te slaan, of geforceerd naar een nieuwe pagina te gaan.

Dit zijn tevens de plaatsen in het rapport waar we subtotalen kunnen berekenen. Dit laatste bereiken we met behulp van het COMPUTE-commando. Maar laten we eerst de mogelijkheden van BREAK maar eens onderzoeken.

De algemene gedaante van het commando BREAK is als volgt:

SQL> **break on <element> [ <actie> ... ] [on <element> [ <actie> ]] ...**

Als <element> kan een kolomnaam of kolomexpressie worden opgegeven, en als <actie> bijvoorbeeld:

**SKIP n**

Sla n regels over

## [SKIP] PAGE

Forceer een paginaovergang

Laten we eens een voorbeeld bekijken ([zie figuur 11.32](#)).

312

```
SQL> clear columns
SQL> select afd,functie,mnr,naam,maandsal,comm
  2  from medewerkers
  3  order by afd,functie;
...
SQL> break on afd skip 2
SQL> /
...
SQL> break
SQL> break on afd page
SQL> set pause on
SQL> /
...
```

```
SQL> break on afd skip page on functie skip 1
SQL> /
...
```

## Figuur 11.32

Let op de ORDER BY-component; die is voor het gewenste effect beslist nodig, omdat het BREAK-commando niet zelf zal zorgen voor sortering.

Er kan steeds maar één break-definitie actief zijn. Iedere definitie van een nieuwe break overschrijft de oude. Willen we een break op verschillende niveaus, dan moeten we dat doen in één BREAK-commando, en wel zoals in [figuur 11.33](#).

## Figuur 11.33

Merk op dat we in dit commando geen komma's gebruiken als

scheidingstekens tussen de BREAK-definities, maar spaties.

Er zijn nog twee bijzondere elementen in een rapport waarop een BREAK kan worden gedefinieerd, namelijk:

## **ROW**

Forceer een break op iedere rij van het resultaat

## **REPORT**

Break aan het einde van het rapport

Nu gaan we op onze breaks berekeningen toevoegen aan het rapport.

Dat doen we met het COMPUTE-commando, waarvan de algemene syntax er als volgt uitziet:

313

<b>AVG</b>	Het gemiddelde
<b>COUNT</b>	Het aantal NOT NULL-waarden in de kolom
<b>MAX</b>	Het maximum
<b>MIN</b>	Het minimum
<b>NUMBER</b>	Het aantal rijen
<b>STD</b>	De standaarddeviatie
<b>SUM</b>	De som
<b>VAR</b>	De variantie

```
SQL> compute sum label totaal of maandsal on afd
SQL> compute count number      of comm      on afd

SQL> /
...
SQL> compute
...
SQL> clear computes
SQL> clear breaks
```

SQL> **compute** <*functie*> [**label** <*tekst*>]... -

> **of** <*kolomexpr*> -

> **on** <*breakspec*>

De toegestane functies van COMPUTE zijn:

### Figuur 11.34

De *kolomexpr* geeft aan op welke kolom de berekening moet worden toegepast; de *breakspec* geeft aan op welk moment dat moet gebeuren.

De *breakspec* moet een kolom (of kolomexpressie) zijn waarop we eerder al een BREAK hebben gedefinieerd.

### Figuur 11.35

Zoals uit dit voorbeeld blijkt, kunnen we wél meerdere COMPUTE-opdrachten na elkaar definiëren, met afzonderlijke commando's. Met CLEAR COMPUTES verwijderen we de definities, en met CLEAR BREAKS verwijderen we de breaks.

Als we tevreden zijn over ons rapport, zetten we alle SQL- en SQL\*Plus-commando's in een script. We kunnen ook nog zorgen voor 314

eventuele uitvoer op papier door gebruik te maken van het SPOOL-commando, zoals [in figuur 11.31](#).

315

## Hoofdstuk 12

### Object-relationele features

In dit laatste hoofdstuk komt zoals beloofd de object-relationele benadering van Oracle aan bod. Hierbij beperken we ons tot de consequenties voor SQL, hoewel object-relationele databases voor een goed begrip eigenlijk moeten

worden beschouwd in de

context van een objectgeoriënteerde ontwikkelomgeving. De

eerste stap in het opzetten van een object-relationele omgeving is het definiëren van de juiste verzameling objecttypes.

Deze objecttypes kunnen we vervolgens gebruiken om er

objecttabellen op te definiëren, waardoor we een volledig object-relationele omgeving creëren; we kunnen echter ook met behulp van object-views een object-relationele laag leggen over een

‘standaard’ relationele omgeving. Behandeling van objecttabellen en object-views zou voor dit boek te ver voeren; in dit hoofdstuk zullen we objecttypes gebruiken als een basis voor

zelfgedefinieerde (user defined) datatypes in relationele tabellen.

We zullen in dit hoofdstuk ook aandacht besteden aan twee

verzamelingswaardige datatypes: arrays en geneste tabellen.

Overigens speelt PL/SQL een belangrijke rol bij het opzetten van een object-relationele omgeving; het is de taal die we moeten gebruiken in de definitiefase van de object-relationele features.

Omdat PL/SQL in dit boek verder niet aan de orde komt,

veronderstellen we enige kennis van deze taal, en gaan we

bijvoorbeeld niet in op het creëren van methoden (methods) voor onze objecttypes. Willen we meer weten over de object-relationele features van Oracle, dan is de “Application

Developer’s Guide - Object-Relational Features” een uitstekend startpunt voor nadere studie.

## 12.1 Nog meer datatypes

Tot nu toe hebben we uitsluitend gebruik gemaakt van de standaard datatypes die door Oracle worden ondersteund, zoals NUMBER, DATE, TIMESTAMP, CHAR en VARCHAR2. Dat betekent dat we twee onderwerpen 316

- 
- 
- 
- 

hebben laten liggen:

verzamelingswaardige datatypes (arrays en geneste tabellen); zelfgedefinieerde datatypes.

Verzamelingswaardige datatypes zijn datatypes waarbij een attribuut niet meer uit een enkele waarde bestaat, maar uit een verzameling van waarden. We kunnen bijvoorbeeld per medewerker een lijst van telefoonnummers opslaan in een enkele kolom, of we kunnen

bijvoorbeeld een verzameling errata toevoegen aan een cursus. Arrays kennen we waarschijnlijk wel uit andere programmeertalen, en geneste tabellen zijn tabellen binnen een tabel.

Het eerstgegeven voorbeeld (de lijst van telefoonnummers) leent zich heel goed voor het gebruik van een array, en wel om twee redenen: We weten over het algemeen vrij redelijk hoe groot zo'n lijst maximaal gaat worden.

We willen waarschijnlijk een betekenis toekennen aan de volgorde van de telefoonnummers, zoals: zakelijk, prive, mobiel of fax.

Het tweede voorbeeld (de errata) kunnen we beter met een geneste tabel oplossen, omdat we van tevoren geen idee hebben hoeveel errata we uiteindelijk per cursus zullen moeten opslaan, en de fysieke volgorde van de errata interesseert ons niet.

Zelfgedefinieerde datatypes stellen ons (zoals de naam al aangeeft) in staat om onze eigen complexe datatypes te definiëren. We kunnen bijvoorbeeld een datatype ADRES maken, met de elementen STRAATNAAM,

## HUISNUMMER, POSTCODE en PLAATSNAAM.

Overigens hebben we zelfgedefinieerde datatypes altijd nodig om er geneste tabellen op te kunnen baseren. Desgewenst kunnen we arrays van zelfgedefinieerde datatypes creëren. We zouden bijvoorbeeld een array van adressen kunnen toevoegen aan de tabel UITVOERINGEN, in plaats van de LOCATIE-kolom, zodat we voor de cursus-uitvoeringen een aantal alternatieve locaties kunnen opnemen. We kunnen natuurlijk ook volstaan met het toevoegen van een enkel adres, in welk geval we geen array nodig hebben.

We kunnen aan zelfgedefinieerde datatypes ook methoden (*methods*) toevoegen, bijvoorbeeld om aan te geven hoe we twee adressen met elkaar willen vergelijken of hoe we adressen willen sorteren. Methoden 317

maken zelfgedefinieerde datatypes krachtiger en rijker aan semantiek; voor het definiëren van methoden is echter nogal wat PL/SQL-programmering nodig, dus laten we methoden in dit boek buiten beschouwing.

Opmerking 1: Alle tot nu toe in deze paragraaf genoemde voorbeelden zijn ook heel goed op te lossen met de standaard relationele technieken die we tot nu toe hebben gebruikt: uitsplitsen van verschillende telefoonnummers over aparte kolommen, een separate ERRATA-tabel maken met een refererende sleutel vanuit de CURSUSSEN-tabel, enzovoort. Waarom we voor de ene of de andere benadering zouden moeten kiezen is voor een deel een kwestie van smaak; daar gaan we verder niet op in. We gebruiken deze voorbeelden in dit hoofdstuk alleen om de object-relationele mogelijkheden van Oracle te illustreren.

De laatste paragraaf van dit hoofdstuk behandelt multiset-operatoren, die alleen maar kunnen worden toegepast op geneste tabellen.

Opmerking 2: Voor het geval we twijfelen of de in dit hoofdstuk beschreven technieken al of niet in strijd zijn met de eerste normaalvorm als een van de fundamenten van het relationele model: dat is niet zo. Het relationele model verbiedt op geen enkele manier het opslaan van complexe of verzamelingswaardige attributen. Trouwens, een datum is eigenlijk ook een complex (of samengesteld) datatype, want het heeft componenten zoals jaar,

maand en dag. Verdere discussie van dit onderwerp valt buiten het bestek van dit boek.

## 12.2 Arrays

We beginnen deze paragraaf met het implementeren van het telefoonlijst-voorbeeld uit de vorige paragraaf. Om de bestaande MEDEWERKERS-tabel niet te verminken, maken we voor de experimenten in dit laatste hoofdstuk een kopie van die tabel, waarbij we voor het gemak ook een aantal kolommen weglaten. [Zie figuur 12.1](#).

318

```
SQL> create table m
  2  as
  3  select mnr, naam, voorl, chef, afd
  4  from medewerkers;
```

Table created.

```
SQL>
```

```
SQL> create or replace type nummerlijst_t
  2  as varray(4) of varchar2(20);
  3  /
```

Type created.

```
SQL> describe nummerlijst_t
nummerlijst_t VARRAY(4) OF VARCHAR2(20)
```

```
SQL> select type_name, typecode
  2  from user_types;
```

TYPE_NAME	TYPECODE
NUMMERLIJST_T	COLLECTION

```
SQL>
```

Figuur 12.1

Voordat we aan deze tabel een lijst van telefoonnummers kunnen toevoegen, moeten we eerst het bijbehorende type definiëren; zie figuur 12.2.

## Figuur 12.2

Merk op dat we dit zelfgedefinieerde type nu zo vaak als we willen kunnen gebruiken; het type is bekend in de database, en opgeslagen in de datadictionary. Bovendien valt in figuur 12.2 op dat het CREATE

TYPE-commando wordt afgesloten met een slash (/) op regel drie, hoewel de tweede regel is afgesloten met een puntkomma. Dit komt omdat we hier niet een SQL- of een SQL\*Plus-commando uitvoeren, maar een PL/SQL-commando.

In figuur 12.3 voegen we een kolom aan de tabel M toe waarbij we gebruik maken van het NUMMERLIJST-type.

```

SQL> alter table m add (nummer nummerlijst_t);

Table altered.

SQL> describe m
Name          Null?    Type
----- -----
MNR           NUMBER(4)
NAAM          NOT NULL VARCHAR2(12)
VOORL          NOT NULL VARCHAR2(5)
CHEF          NUMBER(4)
AFD            NUMBER(2)
NUMMER        NUMMERLIJST_T

SQL> select mnr, nummer from m;

MNR  NUMMER
-----
7369
7499
7521
7566
7654
7698
7782
7788
7839
7844
7876
7900
7902
7934

14 rows selected.

SQL>

```

### ■ ■ ■

### Figuur 12.3

De resultaten spreken nog niet echt tot de verbeelding; natuurlijk is de NUMMER-kolom nog leeg. We moeten dus nog twee problemen overwinnen:

Hoe krijgen we gegevens in de NUMMER-kolom?

Als we dat gedaan hebben, hoe kunnen we ze dan raadplegen?

Ieder zelfgedefinieerd objecttype heeft automatisch een gelijknamige functie waarmee we waarden van dat type kunnen genereren. Deze functie wordt ook wel een constructor genoemd. Als we een objecttype creëren, krijgen we dus een gelijknamige constructor methode cadeau.

In figuur 12.4 wijzen we een nummerlijst toe aan een vijftal 320

```

SQL> update m
  2  set     nummer = nummerlijst_t('1234','06-78765432','029-8765432')
  3  where   mnr = 7839;
1 row updated.

SQL> update m
  2  set     nummer = nummerlijst_t('4231','06-12345678')
  3  where   mnr = 7782;
1 row updated.

SQL> update m
  2  set     nummer = nummerlijst_t('2345')
  3  where   mnr = 7934;
1 row updated.

SQL> update m
  2  set     nummer = nummerlijst_t('')
  3  where   mnr = 7698;
1 row updated.

SQL> update m
  2  set     nummer = nummerlijst_t()
  3  where   mnr in (7566,7844);
2 rows updated.

SQL>

```

```

SQL> select mnr, nummer
  2  from   m
  3  where  mnr in (7839, 7782, 7566, 7698, 7844);

      MNR    NUMMER
----- -----
7566  NUMMERLIJST_T()
7698  NUMMERLIJST_T(NULL, '06-23456789')
7839  NUMMERLIJST_T('1234', '06-78765432', '029-8765432')
7934  NUMMERLIJST_T('2345')

SQL>

```

medewerkers. Merk op dat we elementen kunnen overslaan, en ook een lege nummerlijst kunnen toevoegen.

**Figuur 12.4**

Het raadplegen gaat als volgt. Als we de NUMMER-kolom zonder enige nadere specificatie raadplegen, dan zien we de waarden terug zoals we ze hebben geconstrueerd; zie [figuur 12.5](#).

321

```
SQL> break on mnr

SQL> select m.mnr, n.*
  2  from   m
  3 ,      table(m.nummer) n;

MNR COLUMN_VALUE
-----
7698 06-23456789
7782 4231
      06-12345678
7839 1234
      06-78765432
      029-8765432
7934 2345

SQL>
```

## **Figuur 12.5**

Willen we nu specifieke nummers uit de array selecteren, dan moeten we de gegevens eerst “ontnesten”. Daartoe kunnen we de TABLE-functie gebruiken. Zonder op de details in te gaan laat [figuur 12.6 een voorbeeld](#) van deze functie zien:

## **Figuur 12.6**

Willen we specifieke nummers selecteren, bijvoorbeeld het tweede nummer van de array, dan kan dat niet rechtstreeks in de taal SQL; we moeten dan met behulp van PL/SQL een functie bouwen die het tweede element van de array retourneert. [In hoofdstuk 5 hebben we al](#) een voorbeeld van een opgeslagen functie gezien; [figuur 12.7](#) laat zien hoe we met een opgeslagen functie het eerste telefoonnummer (het interne toestelnummer) kunnen ophalen.

De WHERE-component [in figuur 12.7](#) is “met zorg” gekozen; de opgeslagen functie is namelijk zo eenvoudig mogelijk gehouden, en heeft daarom geen ingebouwde voorzieningen voor het geval een medewerker geen telefoonnummerlijst heeft, of als die lijst leeg is. Als we zelf wat variëren met het voorbeeld dan krijgen we de bijbehorende foutmeldingen. Dit is in principe vrij eenvoudig op te lossen met wat extra PL/SQL in de functiedefinitie, maar het gaat ons hier alleen maar om het voorbeeld.

```

SQL> create or replace function intern
  2      (p_varray_in nummerlijst_t)
  3      return varchar2
  4  is
  5      v_telefoonnummer varchar2(20);
  6  begin
  7      v_telefoonnummer := p_varray_in(1);
  8      return v_telefoonnummer;
  9 end;
10 /
Function created.

```

```

SQL> select naam, voorl, intern(nummer)
  2  from   m
  3  where  afd = 10;

```

NAAM	VOORL	INTERN (NUMMER)
CLERCKX	AB	4231
DE KONING	CC	1234
MOLENAAR	TJA	2345

```
SQL>
```

Het is onmogelijk om specifieke elementen van een array te wijzigen; we kunnen alleen een bestaande array-waarde in zijn geheel vervangen door een nieuwe array-waarde.

## Figuur 12.7

### 12.3 Geneste tabellen

In deze paragraaf zien we een voorbeeld van een geneste tabel; we gaan het voorbeeld van de cursus-errata implementeren. Er zijn veel overeenkomsten tussen arrays en geneste tabellen; een verschil is dat we nu een extra stap moeten uitvoeren om een tabeltype te definiëren. Een voordeel van geneste tabellen is dat ze ons meer flexibiliteit bieden dan

[arrays. In figuur 12.8 maken we eerst de twee types die we voor ons voorbeeld nodig hebben.](#)

```

SQL> create or replace type erratum_t as object
  2  ( code varchar2(4)
  3 , hst number(2)
  3 , blz number(3)
  4 , opm varchar2(40)
  5 ) ;
  6 /


Type created.

SQL> create or replace type errata_tab_t as table of erratum_t;
  2 /


Type created.

SQL> describe errata_tab_t

errata_tab_t TABLE OF ERRATUM_T
Name          Null?    Type
----- -----
CODE          VARCHAR2(4)
HST           NUMBER(2)
BLZ           NUMBER(3)
OPM           VARCHAR2(40)

SQL>

```

```

SQL> create table c
  2  as
  3  select * from cursussen;

Table created.

SQL> alter table c
  2  add (errata errata_tab_t)
  3  nested table errata store as errata_tab;

Table altered.

SQL> update c
  2  set      errata = errata_tab_t();

10 rows updated.

SQL>

```

## Figuur 12.8

Nu maken we een kopie van de CURSUSSEN-tabel, voegen er vervolgens een kolom met een geneste tabel aan toe, en zorgen er voor dat er een lege geneste tabel aan iedere rij wordt toegevoegd; zie [figuur 12.9](#).

324

```
SQL> insert into table ( select errata
  2           from   c
  3           where  code = 'S02')
 4 values ('S02'
  5         , 3
  6         , 45
  7         , 'spelfout in laatste regel');

1 row created.

SQL>
```

```
SQL> update table ( select errata
  2           from   c
  3           where  code = 'S02') e
 4 set     e.hst = 7;

1 row updated.

SQL>
```

## Figuur 12.9

Nu kunnen we rijen toevoegen aan de geneste tabellen; zie [figuur 12.10](#).

Merk op dat we geneste tabellen alleen maar kunnen benaderen in de context van de tabel waarvan ze deel uitmaken; we kunnen ze niet als zelfstandige tabellen benaderen. We gebruiken weer de TABLE-functie, net als in [figuur 12.6](#).

## Figuur 12.10

We hebben nu een erratum ingevoerd voor de S02-[cursus, hoofdstuk 3](#),

bladzijde 45. Op vergelijkbare wijze kunnen we ook rijen van een geneste tabel verwijderen. Geneste tabellen bieden meer flexibiliteit dan arrays; we kunnen namelijk individuele kolomwaarden wijzigen, terwijl we arrays alleen maar in zijn geheel kunnen vervangen. Stel dat we in

[figuur 12.10 een fout](#) hebben gemaakt in het hoofdstuknummer, dan kunnen we dat met een UPDATE-commando corrigeren; zie [figuur 12.11](#).

Merk op dat we op de derde regel de tupelvariabele e introduceren, om hem op de vierde regel te kunnen gebruiken.

## Figuur 12.11

325

```
SQL> select code, omschrijving
  2 ,      hst, blz, opm
  3 from   c
  4 join
  5      table(c.errata) e
  6 using (code);

CODE      OMSCHRIJVING
----- -----
HST BLZ OPM
-----
S02      Introductiecursus SQL
7 45 spelfout in laatste regel

SQL>
```

Als we alle errata voor de S02-cursus willen raadplegen, dan kan dat bijvoorbeeld zoals aangegeven in [figuur 12.12](#).

## Figuur 12.12

Zoals [figuur 12.12](#) laat zien, blijven we eigenlijk vrij dicht bij de syntax voor “gewone” joins; de TABLE-functie converteert zijn argument (in dit geval is

dat c.errata) tot een tabel. Dit is mogelijk omdat we in de FROM-component eerst de tabel C hebben opgenomen; daardoor kunnen we naar c.errata refereren.

In de laatste paragraaf van dit hoofdstuk komen we nog op geneste tabellen terug, als we multiset-operatoren op geneste tabellen gaan toepassen. Deze multiset-operatoren kunnen een belangrijke reden zijn om geneste tabellen te gebruiken, in plaats van “gewone” tabellen met een refererende sleutel.

## 12.4 Zelfgedefinieerde types

Het derde voorbeeld dat in de eerste paragraaf van dit hoofdstuk werd genoemd was het ADRES-type, als kandidaat voor een zelfgedefinieerd datatype. [Figuur 12.13 geeft](#) aan hoe we dat type kunnen creëren, en vervolgens aan een kopie van de UITVOERINGEN-tabel kunnen toevoegen.

```

SQL> create or replace type adres_t as object
  2  ( straatnaam varchar2(20)
  3 , huisnummer varchar2(5)
  4 , postcode    varchar2(6)
  5 , plaatsnaam varchar2(20)
  6 ) ;
  7 /


Type created.

SQL> describe adres_t
Name          Null?    Type
-----
STRAATNAAM           VARCHAR2 (20)
HUISNUMMER          VARCHAR2 (5)
POSTCODE            VARCHAR2 (6)
PLAATSNAAM          VARCHAR2 (20)

SQL> select type_name, typecode
  2  from user_types;

TYPE_NAME          TYPECODE
-----
ERRATUM_T          OBJECT
ERRATA_TAB_T        COLLECTION
NUMMERLIJST_T       COLLECTION
ADRES_T            OBJECT

SQL> create table u
  2  as
  3  select cursus, begindatum, docent
  4  from uitvoeringen;

Table created.

SQL> alter table u add (adres adres_t);

Table altered.

SQL> update u
  2  set   u.adres =
  3      adres_t('','','',
  4                  (select initcap(uv.locatie)
  5                   from uitvoeringen uv
  6                   where uv.cursus      = u.cursus
  7                     and uv.begindatum = u.begindatum)
  8      )
  9  ;

13 rows updated.

SQL>

```

**Figuur 12.13**

327

```

SQL> describe u
      Name          Null?       Type
----- -----
CURSUS           NOT NULL  VARCHAR2(4)
BEGINDATUM      NOT NULL  DATE
DOCENT          NUMBER(4)
ADRES            ADRES_T

SQL> set describe depth 2
SQL> describe u
      Name          Null?       Type
----- -----
CURSUS           NOT NULL  VARCHAR2(4)
BEGINDATUM      NOT NULL  DATE
DOCENT          NUMBER(4)
ADRES            ADRES_T
  STRAATNAAM    VARCHAR2(20)
  HUISNUMMER    VARCHAR2(5)
  POSTCODE      VARCHAR2(6)
  PLAATSNAAM    VARCHAR2(20)

SQL>

```

We gebruiken in het laatste commando van figuur 12.13 de ADRES\_T -

functie om adressen te genereren, waarbij we de eerste drie velden leeg laten en de plaatsnaam ophalen uit de oorspronkelijke UITVOERINGEN-tabel.

Als we zelfgedefinieerde datatypes gebruiken, dan kunnen we de DEPTH van het DESCRIBE-commando zodanig wijzigen dat de componenten van het datatype ook worden weergegeven; zie figuur 12.14.

## Figuur 12.14

### 12.5 Multiset-operatoren

In paragraaf 12.3 hebben we gezien hoe we errata per cursus kunnen opslaan in een geneste tabel. Omdat we pas één rij in één geneste tabel hebben ingevoerd (zie figuur 12.10) voeren we eerst een paar extra rijen in (zie figuur 12.15) zodat we daarna met de multiset-operatoren van SQL aan de slag kunnen.



```

SQL> insert into table ( select errata
  2                      from   c
  3                      where  code = 'S02' )
  4      values ('S02'
  5                  , 3
  6                  , 46
  7                  , 'layout illustratie' );
1 row created.

SQL> insert into table ( select errata
  2                      from   c
  3                      where  code = 'S02' )
  4      values ('S02'
  5                  , 5
  6                  , 1
  7                  , 'introductie ontbreekt' );
1 row created.

SQL> insert into table ( select errata
  2                      from   c
  3                      where  code = 'XML' )
  4      values ('XML'
  5                  , 5
  6                  , 1
  7                  , 'introductie ontbreekt' );
1 row created.

SQL> insert into table ( select errata
  2                      from   c
  3                      where  code = 'XML' )
  4      values ('XML'
  5                  , 7
  6                  , 3
  7                  , 'regel 5: "slaagt" moet zijn "faalt"' );
1 row created.

SQL>

```

## Figuur 12.15

Als we de tabel C “gewoon” raadplegen ([zie figuur 12.16](#)) dan wordt de structuur van de ERRATA-kolom met de geneste tabel duidelijk.

```

SQL> col errata format a80 word

SQL> select errata
2  from   c
3  where  code = 'S02';

ERRATA(CODE, HST, BLZ, OPM)
-----
ERRATA_TAB_T(ERRATUM_T('S02', 7, 45, 'spelfout in laatste regel'),
ERRATUM_T('S02', 3, 46, 'layout illustratie'), ERRATUM_T('S02', 5, 1,
'introductie ontbreekt'))

SQL>

```

## Figuur 12.16

Het resultaat van de query [in figuur 12.16 bestaat](#) uit één rij met één kolom; er staat dus één waarde op het scherm. Als we het resultaat van binnen naar buiten lezen, dan zien we dat de ERRATUM\_T-functie driemaal is gebruikt om een individueel erratum op te bouwen, waarna de drie resultaten daarvan door de ERRATA\_TAB\_T-functie worden omgezet naar een (geneste) tabel.

Als we in onze tabellen gebruik maken van geneste tabellen, dan kunnen we op deze tabellen multiset-operatoren loslaten. De taal SQL gebruikt overigens de term “multiset” om aan te geven dat we het niet over échte verzamelingen hebben, maar over verzamelingen waarin duplicates zijn toegestaan (en ook een verschil maken).

Met multiset-operatoren kunnen we geneste tabellen met elkaar vergelijken, of er een nieuwe tabel van afleiden (bijvoorbeeld met de doorsnede, het verschil, of de vereniging). [Figuur 12.17 geeft](#) een overzicht van de Oracle multiset-operatoren, die overigens ook onderdeel uitmaken van de SQL-standaard. Voor de volledigheid zijn niet alleen de multiset-operatoren maar ook de andere toegestane bewerkingen op geneste tabellen in deze figuur opgenomen.

<code>nt1 MULTISET EXCEPT [distinct] nt2</code>	De verschilverzameling
<code>nt1 MULTISET intersecT [distinct] nt2</code>	De doorsnede
<code>nt1 MULTISET UNION [DISTINCT] nt2</code>	De vereniging
<code>CARDINALITY(nt)</code>	Het aantal rijen van een geneste tabel
<code>nt IS [NOT] EMPTY</code>	Is een geneste tabel leeg?
<code>nt IS [NOT] A SET</code>	Bevat een geneste tabel duplicaten?
<code>SET(nt1)</code>	Verwijder duplicaten uit <code>nt1</code>
<code>nt1 = nt2</code>	Zijn twee geneste tabellen gelijk?
<code>nt1 IN (nt2, nt3, ...)</code>	Komt <code>nt1</code> in een lijst van geneste tabellen voor?
<code>nt1 [NOT] SUBMULTISET OF nt2</code>	Is <code>nt1</code> een deelverzameling van <code>nt2</code> ?
<code>r [not] MEMBER of nt</code>	Komt rij <code>r</code> in tabel <code>nt</code> voor?
<code>CAST(COLLECT(col))</code>	Produceer een geneste tabel op basis van kolom <code>col</code>
<code>POWERMULTISET(nt)</code>	De machtsverzameling van een geneste tabel: alle niet-lege deelverzamelingen
<code>POWERMULTISET_BY_CARDINALITY(nt,c)</code>	Idem, maar met een gegeven cardinaliteit

```
SQL> select code, cardinality(errata)
  2  from   c
  3  where  errata is not empty;

CODE      CARDINALITY (ERRATA)
-----  -----
S02                  3
XML                  2
```

## Figuur 12.17

De Oracle-documentatie geeft van al deze operatoren voorbeelden; de volgende figuren laten zien hoe we een paar van deze operatoren kunnen gebruiken.

## Figuur 12.18

In figuur 12.18 gebruiken we de operator IS NOT EMPTY om de cursussen te selecteren waarvoor minstens één erratum voorkomt, en we gebruiken de CARDINALITY-functie om van die cursussen het aantal errata te tellen.

In figuur 12.19 produceren we de machtsverzameling (powerset) van de errata voor de S02-cursus. Om de leesbaarheid te verhogen plaatsen we een BREAK op iedere rij; daardoor wordt het duidelijk dat het resultaat uit 331

```

SQL> break on row page

SQL> select *
  2  from  table ( select powermultiset(errata)
  3                  from   c
  4                  where  code = 'S02' );

COLUMN_VALUE(CODE, HST, BLZ, OPM)
-----
ERRATA_TAB_T(ERRATUM_T('S02', 7, 45, 'spelfout in laatste regel'))

COLUMN_VALUE(CODE, HST, BLZ, OPM)
-----
ERRATA_TAB_T(ERRATUM_T('S02', 3, 46, 'layout illustratie'))

COLUMN_VALUE(CODE, HST, BLZ, OPM)
-----
ERRATA_TAB_T(ERRATUM_T('S02', 7, 45, 'spelfout in laatste regel'),
              ERRATUM_T('S02', 3, 46, 'layout illustratie'))

COLUMN_VALUE(CODE, HST, BLZ, OPM)
-----
ERRATA_TAB_T(ERRATUM_T('S02', 5, 1, 'introductie ontbreekt'))

COLUMN_VALUE(CODE, HST, BLZ, OPM)
-----
ERRATA_TAB_T(ERRATUM_T('S02', 7, 45, 'spelfout in laatste regel'),
              ERRATUM_T('S02', 5, 1, 'introductie ontbreekt'))

COLUMN_VALUE(CODE, HST, BLZ, OPM)
-----
ERRATA_TAB_T(ERRATUM_T('S02', 3, 46, 'layout illustratie'),
              ERRATUM_T('S02', 5, 1, 'introductie ontbreekt'))

COLUMN_VALUE(CODE, HST, BLZ, OPM)
-----
ERRATA_TAB_T(ERRATUM_T('S02', 7, 45, 'spelfout in laatste regel'),
              ERRATUM_T('S02', 3, 46, 'layout illustratie'),
              ERRATUM_T('S02', 5, 1, 'introductie ontbreekt'))

7 rows selected.

SQL>

```

zeven rijen bestaat, waarbij elk van die zeven rijen een verzameling is.

## Figuur 12.19

Het resultaat bestaat uit zeven rijen omdat we drie errata hebben; er zijn drie deelverzamelingen mogelijk met cardinaliteit 1, drie met cardinaliteit twee, en één met cardinaliteit 3 (de verzameling zelf). In de wiskunde zouden we ook nog de lege verzameling in het resultaat 332

```
SQL> select c1.errata
  2      multiset union
  3      c2.errata
  4      as resultaat
  5  from  c c1,
  6        c c2
  7 where c1.code = 'S02'
  8 and   c2.code = 'XML';

RESULTAAT(CODE, HST, BLZ, OPM)
-----
ERRATA_TAB_T(ERRATUM_T('S02', 7, 45, 'spelfout in laatste regel'), ERRATUM_T('S02', 3, 46, 'layout illustratie'), ERRATUM_T('S02', 5, 1, 'introduction ontbreekt'), ERRATUM_T('XML', 5, 1, 'introduction ontbreekt'), ERRATUM_T('XML', 7, 3, 'regel 5: "slaagt" moet zijn "faalt"'))
SQL>
```

```
SQL> create type nummer_tab_t
  2  as table of nummerlijst_t;
  3  /

Type created.

SQL> select cast(collect(nummer) as nummer_tab_t) as resultaat
  2  from  m
  3  where  mnr in (7839, 7782);

RESULTAAT
-----
NUMMER_TAB_T(NUMMERLIJST_T('4231', '06-12345678'),
              NUMMERLIJST_T('1234', '06-78765432', '029-8765432'))
```

SQL>

verwachten, maar die deelverzameling wordt expliciet in de definitie van de POWERMULTISET-operator uitgesloten ([zie figuur 12.17](#)).

In [figuur 12.20 gebruiken](#) we de MULTISET UNION-operator om twee

geneste tabellen samen te voegen:

### **Figuur 12.20**

[In figuur 12.21 vallen we terug op de tabel M met de array \(zie paragraaf 12.2\) en gebruiken we de COLLECT-operator om de array te converteren](#)

naar een geneste tabel. Om het resultaat op te vangen creëren we eerst een nieuw type nummer\_tab\_t met gebruikmaking van het bestaande nummerlijst\_t-type.

333

### **Figuur 12.21**

334

## **Appendix A**

### **De casus**

Deze bijlage biedt – in diverse vormen – een overzicht van de casus die we in dit boek gebruiken.

Allereerst geven we een ERM-diagram, waarin de entiteiten van het datamodel worden aangegeven met hun unieke identifiers, alsmede hun onderlinge relaties.

Daarna volgen beschrijvingen van de zeven casustabellen, met per kolom het datatype en zo nodig een korte toelichting.

Dan volgt een tabeldiagram, waaruit de primaire en in het bijzonder de refererende sleutels zijn op te maken.

Vervolgens bieden we de inhoud van de zeven tabellen.

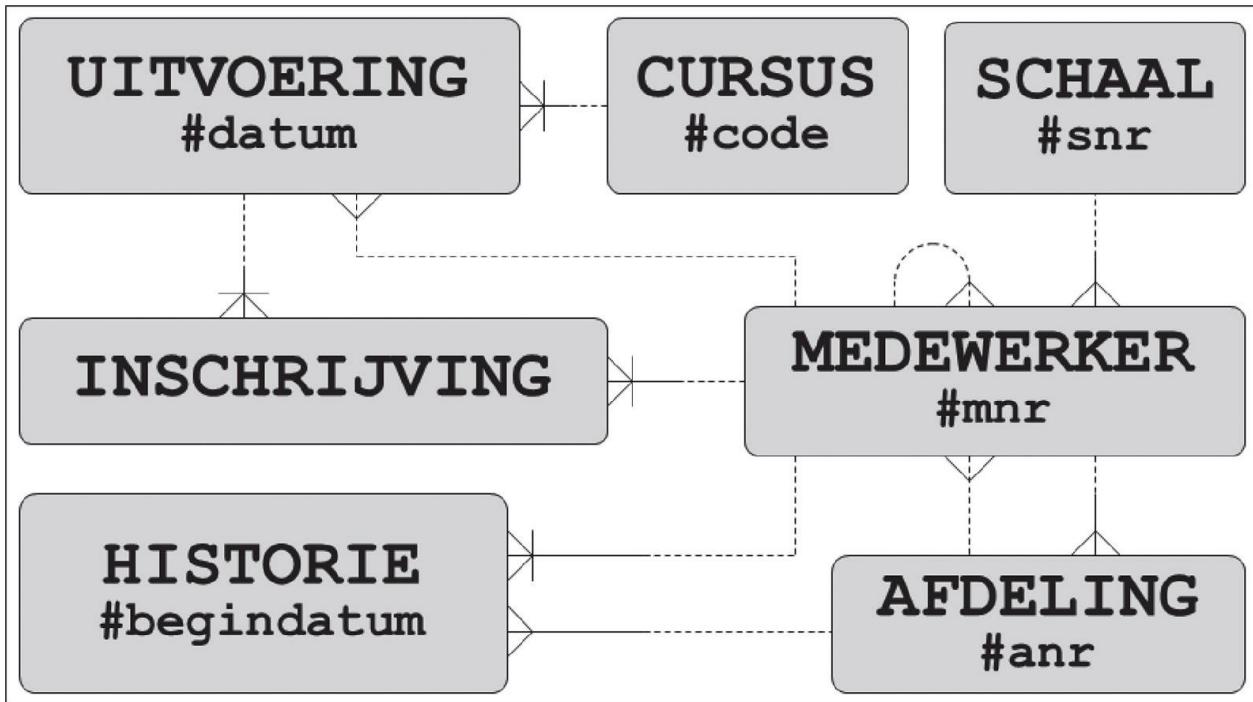
Ten slotte zijn twee illustraties opgenomen die de casusinhoud zo compact

mogelijk weergeven. Deze illustraties kunnen bijvoorbeeld prettig zijn bij het controleren van onze queryresultaten. Het eerste diagram geeft een overzicht van alle medewerkers, waaruit we de hiërarchische verhoudingen en de afdelingspopulaties eenvoudig kunnen aflezen. Het tweede plaatje geeft een matrixoverzicht van alle cursusuitvoeringen, met datum, locatie, cursisten (C) en docenten (D).

Overigens wordt de casus ook elders in dit boek beschreven; verwijzingen naar de betreffende bladzijden zijn in deze bijlage opgenomen.

ERM-diagram

335



- 
- 
- 
- 
- 
- 
- 

### Uitleg

In dit ERM-diagram (een herhaling van figuur 1.2) zien we de zeven entiteiten met de unique identifiers en de onderlinge relaties. De kraaienpoten geven een-op-veelrelaties aan; een hekje (#) voor een attribuut betekent dat het attribuut onderdeel is van de unique identifier, en een dwarsstreepje door een relatie betekent dat die relatie onderdeel is van de unique identifier.

De relaties zijn als volgt te lezen:

Iedere medewerker heeft hoogstens één chef (en een medewerker kan meerdere ondergeschikten hebben).

Iedere medewerker valt qua maandsalaris in een bepaalde schaal en hoort bij hoogstens één afdeling.

Iedere afdeling heeft precies één chef.

Ieder uitvoering is van een bestaande cursus, met hoogstens één medewerker als docent.

Ieder inschrijving (als deelnemer) is van een bestaande medewerker voor een bestaande uitvoering.

Ieder historie-record heeft betrekking op precies één medewerker en precies één afdeling.

Tabelstructuren

336

\*: NOT NULL

P: Primaire sleutel

## **MEDEWERKERS**

MNR

N(4)

uniek medewerkersnummer

NAAM

VC(12)

achternaam (evt. voorvoegsels

ervoor)

VOORL

VC(5)

P

voorletters (zonder  
interpunctie)

FUNCTIE

VC(10)

\*

taakomschrijving

CHEF

N(4)

\*

nummer van de chef

| ref.

sleutel

GBDATUM

DATE

\*

geboortedatum

MAANDSAL

N(6,2)

\*

maandsalaris (exclusief netto

toelage)

COMM

N(6,2)

commissie (op jaarbasis, voor

verkopers)

AFD

N(2)

afdeling

| ref.

sleutel

## **AFDELINGEN**

ANR

N(2)

P

uniek afdelingsnummer

NAAM

VC(20)

\*

naam van de afdeling

**LOCATIE**

VC(20)

\*

plaats van vestiging

**HOOFD**

N(4)

nummer afdelingshoofd

| ref.

sleutel

**SCHALEN**

SNR

N(2)

P

uniek salarisschaalnummer

**ONDERGRENS**

N(6,2)

\*

laagste salaris in deze schaal

**BOVENGRENS**

N(6,2)

\*

hoogste salaris in deze schaal

**TOELAGE**

N(6,2)

\*

netto toelage op het

maandsalaris

**CURSUSSEN**

**CODE**

VC(4)

P

unieke code

**OMSCHRIJVING**

VC(50)

\*

cursus-omschrijving

**TYPE**

C(3)

\*

cursustype (ALG, BLD, of DSG)

**LENGTE**

N(2)

\*

cursuslengte in dagen

**UITVOERINGEN**

**CURSUS**

VC(4)

cursuscode

| ref.

337

sleutel

**BEGINDATUM**

**DATE**

P

datum eerste cursusdag

**DOCENT**

N(4)

P

persoon die de cursus geeft

| ref.

sleutel

LOCATIE

VC(20)

daar vindt de cursus plaats

**INSCHRIJVINGEN**

CURSIST

N(4)

P nummer medewerker

|

samengestelde

| ref.

sleutel

CURSUS

VC(4)

P cursuscode

BEGINDATUM

DATE

P datum eerste cursusdag

EVALUATIE

N(1)

waardering (op een

schaal 1 - 5)

## **HISTORIE**

MNR

N(4)

P nummer medewerker

| ref.

sleutel

## **BEGINJAAR**

N(4)

\* jaartal van begindatum

BEGINDATUM DATE

P begindatum interval

EINDDATUM

DATE

einddatum interval

AFD

N(2)

\* afdeling gedurende het

| ref.

interval

Sleutel

**MAANDSAL**

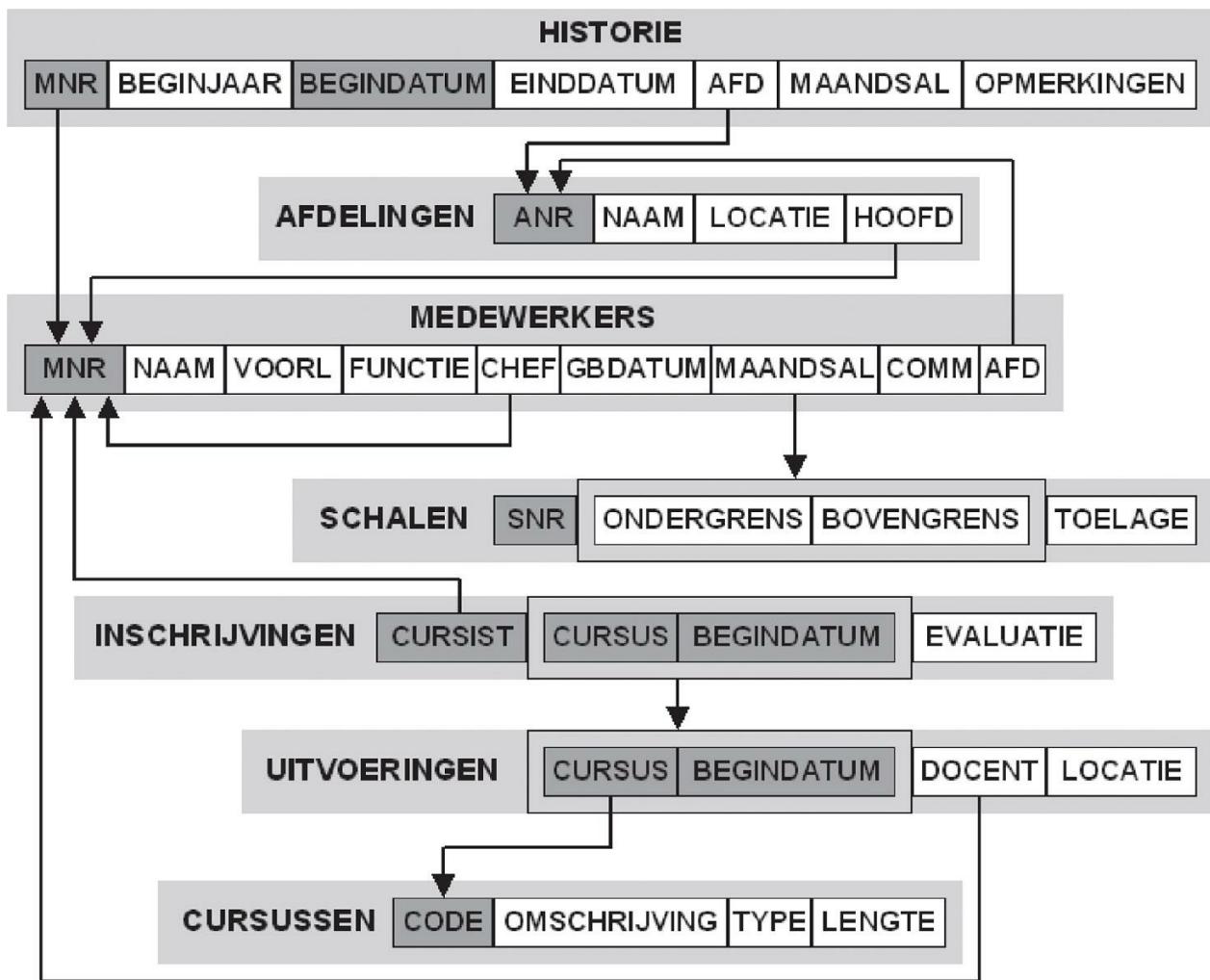
N(6,2) \* maandsalaris gedurende het

interval

**OPMERKINGEN VC(60)** ruimte voor vrije tekst

Kolommen en refererende sleutels

338



**MEDEWERKERS**

MNR	NAAM	VOORL	FUNCTIE	CHEF	GBDATUM	MAANDSAL	COMM	AFD
7369	SMIT	N	TRAINER	7902	17-DEC-65	800		20
7499	ALDERS	JAM	VERKOPER	7698	20-FEB-61	1600	300	30
7521	DE WAARD	TF	VERKOPER	7698	22-FEB-62	1250	500	30
7566	JANSEN	JM	MANAGER	7839	02-APR-67	2975		20
7654	MARTENS	P	VERKOPER	7698	28-SEP-56	1250	1400	30
7698	BLAAK	R	MANAGER	7839	01-NOV-63	2850		30
7782	CLERCKX	AB	MANAGER	7839	09-JUN-65	2450		10
7788	SCHOTTEN	SCJ	TRAINER	7566	26-NOV-59	3000		20
7839	DE KONING	CC	DIRECTEUR		17-NOV-52	5000		10
7844	DEN DRAAIER	JJ	VERKOPER	7698	28-SEP-68	1500	0	30
7876	ADAMS	AA	TRAINER	7788	30-DEC-66	1100		20
7900	JANSEN	R	BOEKHOUDER	7698	03-DEC-69	800		30
7902	SPIJKER	MG	TRAINER	7566	13-FEB-59	3000		20
7934	MOLENAAR	TJA	BOEKHOUDER	7782	23-JAN-62	1300		10

14 rows selected.

**Inhoud van de tabellen**

339

**AFDELINGEN****ANR****NAAM****LOCATIE****HOOFD**

10

**HOOFDKANTOOR****LEIDEN**

7782

20

OPLEIDINGEN

DE MEERN

7566

30

VERKOOP

UTRECHT

7698

40

PERSONEELSZAKEN

GRONINGEN

7839

4 rows selected.

**SCHALEN>**

SNR

ONDERGRENS

BOVENGRENS

TOELAGE

1

700

1200

0

2

1201

1400

50

3

1401

2000

100

4

2001

3000

200

5

3001

9999

500

5 rows selected.

CURSUSSEN

CODE

OMSCHRIJVING

TYPE

LENGTE

S02

Introductiecursus SQL

ALG

4

OAG

Oracle voor applicatiegebruikers

ALG

1

JAV

Java voor Oracle ontwikkelaars

BLD

4

PLS

Introductie PL/SQL

BLD

1

XML

XML voor Oracle ontwikkelaars

BLD

2

ERM

Datamodellering met ERM

DSG

3

PMT

Procesmodelleringstechnieken

DSG

1

RSO

Relationeel systeemontwerp

DSG

2

PRO

Prototyping

DSG

5

GEN

Systeemgeneratie

DSG

4

10 rows selected.

UITVOERINGEN

CURSUS

BEGINDATUM

DOCENT

LOCATIE

S02

12-04-1999

7902

DE MEERN

OAG

10-08-1999

7566

UTRECHT

S02

04-10-1999

7369

MAASTRICHT

S02

13-12-1999

7369

DE MEERN

340

JAV

13-12-1999

7566

MAASTRICHT

XML

03-02-2000

7369

DE MEERN

JAV

01-02-2000

7876

DE MEERN

PLS

11-09-2000

7788

DE MEERN

XML

18-09-2000

MAASTRICHT

OAG

27-09-2000

7902

DE MEERN

ERM

15-01-2001

PRO

19-02-2001

DE MEERN

RSO

24-02-2001

7788

UTRECHT

13 rows selected.

INSCHRIJVINGEN

CURSIST

CURSUS

BEGINDATUM

EVALUATIE

7499

S02

12-04-1999

4

7499

JAV

13-12-1999

2

7499

XML

03-02-2000

5

7499

PLS

11-09-2000

7521

OAG

10-08-1999

4

7566

JAV

01-02-2000

3

7566

PLS

11-09-2000

7698

S02

12-04-1999

4

7698

S02

13-12-1999

7698

JAV

01-02-2000

5

7782

JAV

13-12-1999

5

7788

S02

04-10-1999

7788

JAV

13-12-1999

5

7788

JAV

01-02-2000

4

7839

S02

04-10-1999

3

7839

JAV

13-12-1999

4

7844

OAG

27-09-2000

5

7876

S02

12-04-1999

2

7876

JAV

13-12-1999

5

7876

PLS

11-09-2000

7900

OAG

10-08-1999

4

7900

XML

03-02-2000

4

7902

OAG

10-08-1999

5

7902

S02

04-10-1999

4

7902

S02

13-12-1999

7934

S02

12-04-1999

5

26 rows selected.

341

MNR	JAAR	BEGIN		MAAND		OPMERKINGEN
		BEGINDATUM	EINDDATUM	AFD	SAL	
7369	2000	01-01-2000	01-02-2000	40	950	
7369	2000	01-02-2000		20	800	Overgang naar opleidingen, met ...
7499	1988	01-06-1988	01-07-1989	30	1000	
7499	1989	01-07-1989	01-12-1993	30	1300	
7499	1993	01-12-1993	01-10-1995	30	1500	
7499	1995	01-10-1995	01-11-1999	30	1700	
7499	1999	01-11-1999		30	1600	Targets al weer niet gehaald; salaris-verlaging
7521	1986	01-10-1986	01-08-1987	20	1000	
7521	1987	01-08-1987	01-01-1989	30	1000	Overgang naar afdeling verkoop op eigen verzoek
7521	1989	01-01-1989	15-12-1992	30	1150	
7521	1992	15-12-1992	01-10-1994	30	1250	
7521	1994	01-10-1994	01-10-1997	20	1250	
7521	1997	01-10-1997	01-02-2000	30	1300	
7521	2000	01-02-2000		30	1250	
7566	1982	01-01-1982	01-12-1982	20	900	
7566	1982	01-12-1982	15-08-1984	20	950	
7566	1984	15-08-1984	01-01-1986	30	1000	Niet zo geschikt als docent; dan maar naar ...
7566	1986	01-01-1986	01-07-1986	30	1175	Verkoop is ook al niet zo'n succes...
7566	1986	01-07-1986	15-03-1987	10	1175	
7566	1987	15-03-1987	01-04-1987	10	2200	
7566	1987	01-04-1987	01-06-1989	10	2300	
7566	1989	01-06-1989	01-07-1992	40	2300	Van hoofdkantoor naar personeelszaken;
7566	1992	01-07-1992	01-11-1992	40	2450	...
7566	1992	01-11-1992	01-09-1994	20	2600	Terug naar afdeling opleidingen, als hoofd
7566	1994	01-09-1994	01-03-1995	20	2550	
7566	1995	01-03-1995	15-10-1999	20	2750	
7566	1999	15-10-1999		20	2975	
7654	1999	01-01-1999	15-10-1999	30	1100	Senior verkoper; zou wel eens een aan-winst ...
7654	1999	15-10-1999		30	1250	Valt toch een beetje tegen.
7698	1982	01-06-1982	01-01-1983	30	900	
7698	1983	01-01-1983	01-01-1984	30	1275	
7698	1984	01-01-1984	15-04-1985	30	1500	
7698	1985	15-04-1985	01-01-1986	30	2100	
7698	1986	01-01-1986	15-10-1989	30	2200	

HISTORIE

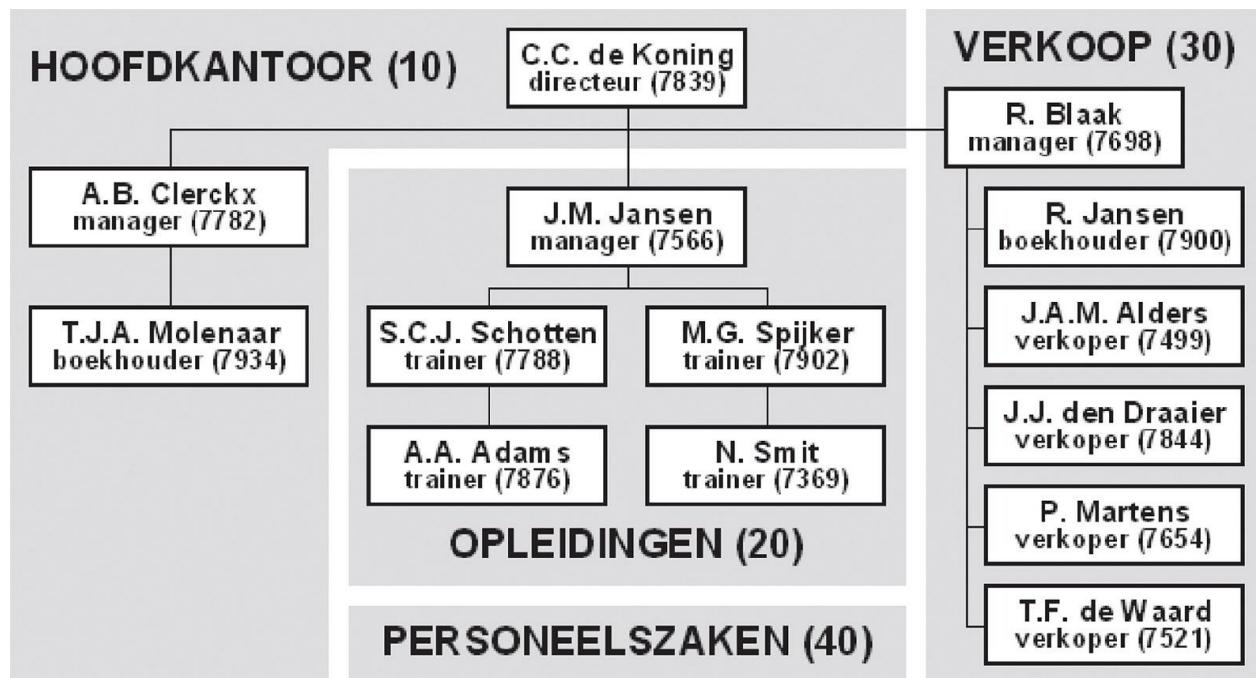
342

7698	1989	15-10-1989	30	2850	Gepromoveerd tot hoofd van de afdeling verkoop
7782	1988	01-07-1988	10	2450	Aangenomen als manager voor het hoofdkantoor
7788	1982	01-07-1982	01-01-1983	20	900
7788	1983	01-01-1983	15-04-1985	20	950
7788	1985	15-04-1985	01-06-1985	40	950 Overgang naar personeelszaken, zonder ...
7788	1985	01-06-1985	15-04-1986	40	1100
7788	1986	15-04-1986	01-05-1986	20	1100
7788	1986	01-05-1986	15-02-1987	20	1800
7788	1987	15-02-1987	01-12-1989	20	1250 Salarisverlaging met 550, vanwege ...
7788	1989	01-12-1989	15-10-1992	20	1350
7788	1992	15-10-1992	01-01-1998	20	1400
7788	1998	01-01-1998	01-01-1999	20	1700
7788	1999	01-01-1999	01-07-1999	20	1800
7788	1999	01-07-1999	01-06-2000	20	1800
7788	2000	01-06-2000		20	3000
7839	1982	01-01-1982	01-08-1982	30	1000 Oprichter en eerste werknemer van het bedrijf
7839	1982	01-08-1982	15-05-1984	30	1200
7839	1984	15-05-1984	01-01-1985	30	1500
7839	1985	01-01-1985	01-07-1985	30	1750
7839	1985	01-07-1985	01-11-1985	10	2000 Hoofdkantoor als nieuwe zelfstandige ...
7839	1985	01-11-1985	01-02-1986	10	2200
7839	1986	01-02-1986	15-06-1989	10	2500
7839	1989	15-06-1989	01-12-1993	10	2900
7839	1993	01-12-1993	01-09-1995	10	3400
7839	1995	01-09-1995	01-10-1997	10	4200
7839	1997	01-10-1997	01-10-1998	10	4500
7839	1998	01-10-1998	01-11-1999	10	4800
7839	1999	01-11-1999	15-02-2000	10	4900
7839	2000	15-02-2000		10	5000
7844	1995	01-05-1995	01-01-1997	30	900
7844	1998	15-10-1998	01-11-1998	10	1200 Project (van een halve maand) voor het hoofd...
7844	1998	01-11-1998	01-01-2000	30	1400
7844	2000	01-01-2000		30	1500
7876	2000	01-01-2000	01-02-2000	20	950
7876	2000	01-02-2000		20	1100
7900	2000	01-07-2000		30	800 Junior verkoper -- moet nog veel le- ren...
7902	1998	01-09-1998	01-10-1998	40	1400
7902	1998	01-10-1998	15-03-1999	30	1650
7902	1999	15-03-1999	01-01-2000	30	2500
7902	2000	01-01-2000	01-08-2000	30	3000

343

7902	2000	01-08-2000		20	3000
7934	1998	01-02-1998	01-05-1998	10	1275
7934	1998	01-05-1998	01-02-1999	10	1280
7934	1999	01-02-1999	01-01-2000	10	1290
7934	2000	01-01-2000		10	1300

79 rows selected.



Hiërarchisch overzicht medewerkerstabel

Casusinformatie elders in het boek

**Waar:**

## **Toelichting:**

1.9

De casus

Eerste introductie van de casus

3.4

De casustabellen

Eenvoudige create table-commando's (zonder constraints)

7.4

Constraints

Volledige create table-commando's  
(met constraints)

Overzicht cursusuitvoeringen

344

	cursus:	<b>S02</b>	OAG	<b>S02</b>	JAV	<b>S02</b>	JAV
	datum:	12/04/99	10/08/99	04/10/99	13/12/99	13/12/99	01/02/00
	locatie:	de Meern	Utrecht	Maastr	Maastr	de Meern	de Meern
N. Smit	7369	.	.	<b>D</b>	.	<b>D</b>	.
J.A.M. Alders	7499	C	.	.	C	.	.
T.F. de Waard	7521	.	C	.	.	.	.
J.M. Jansen	7566	.	<b>D</b>	.	<b>D</b>	.	C
P. Martens	7654	.	.	.	.	.	.
R. Blaak	7698	C	.	.	.	C	C
A.B. Clerckx	7782	.	.	.	C	.	.
S.C.J. Schotten	7788	.	.	C	C	.	C
C.C. de Koning	7839	.	.	C	C	.	.
J.J. den Draaier	7844	.	.	.	.	.	.
A.A. Adams	7876	C	.	.	C	.	<b>D</b>
R. Jansen	7900	.	C	.	.	.	.
M.G. Spijker	7902	<b>D</b>	C	C	.	C	.
T.J.A. Molenaar	7934	C	.	.	.	.	.

		<b>cursus:</b>	<b>XML</b>	<b>PLS</b>	...	<b>OAG</b>	...	<b>RSO</b>
		<b>datum:</b>	03/02/00	11/09/00	...	27/09/00	...	24/02/01
		<b>locatie:</b>	de Meern	de Meern	...	de Meern	...	Utrecht
N. Smit	7369		<b>D</b>	.	...	.	...	.
J.A.M. Alders	7499		C	C	...	.	...	.
T.F. de Waard	7521	.	.	.	...	.	...	.
J.M. Jansen	7566	.	.	C	...	.	...	.
P. Martens	7654	.	.	.	...	.	...	.
R. Blaak	7698	.	.	.	...	.	...	.
A.B. Clerckx	7782	.	.	.	...	.	...	.
S.C.J. Schotten	7788	.		<b>D</b>	...	.	...	<b>D</b>
C.C. de Koning	7839	.	.	.	...	.	...	.
J.J. den Draaier	7844	.	.	.	...	C	...	.
A.A. Adams	7876	.		C	...	.	...	.
R. Jansen	7900	C	.	.	...	.	...	.
M.G. Spijker	7902	.	.	.	...	<b>D</b>	...	.
T.J.A. Molenaar	7934	.	.	.	...	.	...	.
<hr/>								
	<b>cursus:</b>	<b>XML</b>	<b>ERM</b>	<b>PRO</b>	wel gepland; géén docent; géén inschrijving			
	<b>datum:</b>	18/09/00	15/01/01	19/02/01				
	<b>locatie:</b>	Maastr		deMeern				

346

## Index

### Symbolen

[82](#)

[;35](#)

@ [44](#), [243](#)

\* [35](#), [68](#), [168](#)

/ [37](#)

& [234](#)

&& [238](#)

# [17](#)

% [82](#)

+ [162](#)

| [251](#)

\_editor [36](#)

A

ABS [98](#)

accept [239](#)

ACOS [98](#)

ADD [133](#)

ADD\_MONTHS [109](#)

afleidbare gegevens [3](#)

aggregatie [164](#)

alfanumeriek [27](#)

alfanumerieke operatoren [29](#)

algebra [8](#)

algemene functies [111](#)

ALIAS [249](#)

aliassen [152](#)

all [189, 190](#)

ALTER [R 23, 26](#)

ALTER SESSION [45, 58](#)

ALTER TABLE [133](#)

analytische functies [203](#)

AND [30, 77](#)

ANSI/ISO [14, 23](#)

SQL-standaard [95](#)

any [189](#)

347

ANY [190](#)

append [42](#)

APPEND [39](#)

applicable [91](#)

arrays [255](#)

A [S 130](#)

ASC 75

ASCII101

ASIN98

AS OF208

ATAN98

attribuut 2, 3, 10

-constraint 19

-waarde10

auditing 27

AUTOCOMMIT125

autorisatie25

AUTOTRACE142

AVG166, 206, 253

B

backslash107

basisgegevens 3

berekeningen 70

bestandsspecificatie44

between80

beveiliging 2, 21, 24, 221

bewerkingen 11

bitmap indexen 141

boomstructuur 198

BREAK 248, 252

btitle 250

BTITLE 248

C

calculus 8

CARDINALITY 269

Cartesiaans produkt 153, 172

Cartesisch produkt 12

CASCADE CONSTRAINTS 149

CASE-expressie 83, 111

case-insensitive 106

case-sensitive 106

348

CAST 269

casus 16

hiërarchisch overzicht medewerkerstabell 281

inhoud van de tabellen 276

kolommen en refererende sleutels [276](#)

overzicht cursusuitvoeringen [282](#)

tabelstructuren [274](#)

[casustabellen](#) [137](#)

CAT [64](#)

CEIL [98](#)

CHANGE [37](#)

CHAR [59](#), [131](#)

CHECK [59](#)

childeren [197](#)

CHR [101](#)

Chris Date [11](#), [91](#)

CLEAR [249](#)

CLEAR BREAKS [254](#)

clear buffer [48](#)

clear COMPUTE [S 254](#)

CLEAR SCREEN [48](#)

CLOB [59](#)

COALESCE [111](#)

Codd [8](#), [13](#), [62](#), [91](#)

**COLLECT** [269, 271](#)

**COLS** [64](#)

[column 46](#)

**COLUMN** [46, 87, 249](#)

[comment 149](#)

commentaar [32](#)

**COMMIT** [23, 124](#)

[compatibiliteit 59](#)

[complexiteit 1](#)

**COMPUTE** [248, 252](#)

**CONCAT** [101, 240](#)

concateneren [28](#)

CONCAT-karakter [235](#)

concurrency [126](#)

[conditie 30, 73](#)

[conditionele uniciteit 150](#)

**CONNECT** [58, 248](#)

[consistent 4](#)

[consistentie 124](#)

[constante 27](#)

349

constraint [5, 7, 8, 14, 59, 122, 134](#)

constructor [259](#)

conversiefuncties [113](#)

correctheid [10, 16, 79, 90, 183](#)

correlatie-namen [152](#)

COS [98](#)

COSH [98](#)

COUNT [166, 206, 253](#)

COUNT(\*) [169](#)

create index [141](#)

CREATE SCHEMA [139](#)

create synonym [145](#)

create table [129](#)

CREATE TABLE [58](#)

CREATE VIEW [214](#)

CROSS join [160](#)

CUBE [176](#)

cumulatief [205](#)

CURRENT [206](#)

CURRENT\_DATE [29](#)

current\_schema [58](#)

CURRVAL [145](#)

D

data

-definitie [21](#)

-dictionary [7, 13, 14, 62, 216](#)

-manipulatie [13, 21, 142, 222, 227](#)

-model [5, 6](#)

-modelleren [10](#)

database [7](#)

-applicaties [8](#)

integriteit [10](#)

links [148](#)

datatype [58, 59, 131](#)

-conversie [116](#)

DATE [108](#)

datumformaten [109](#)

datumfuncties [108, 109](#)

datums [27, 60](#)

[day](#) [109](#)

[DAY](#) [28](#)

[DAY TO MINUTE](#) [108](#)

350

[D](#)[BMS](#) [7, 9, 22, 126](#)

[DECODE](#) [111](#)

deelver[zameling](#) [203](#)

deferrable constraints [140](#)

DEFERRED [140](#)

def[ine](#) [237](#)

DEFINE [36, 240](#)

-karakter [234](#)

[del](#) [40](#)

[DEL](#) [39](#)

[DELETE](#) [23, 26, 40, 121](#)

DENSE\_RANK [206](#)

DEPTH [266](#)

[desc](#) [75](#)

describe [48, 62, 64, 118, 216](#)

DESCRIBE [266](#)

Developer/[2000](#) [15](#)

[diagnostic tools](#) [142](#)

diagram [6](#)

[dict\\_columns](#) [64](#)

DISABLE [135](#)

[distinct](#) [70](#), [167](#), [183](#)

DISTINCT [92](#), [269](#)

domein [10](#)

door[snede](#) [12](#), [182](#)

driewaar[dige logica](#) [11](#), [88](#)

drijvende-[komma getallen](#) [27](#)

DROP [23](#)

DROP INDEX [142](#)

[drop table](#) [121](#)

DUAL [72](#)

dubbele rijen [70](#)

dummy-[tabel](#) [72](#)

duplicaatrijen [182](#)

dynamic performance view[s](#) [64](#)

E

ECHO [240](#)

EDIT [37](#)

editor [36](#)

een-op-[veel](#) [16](#)

[ellipsis](#) [41](#)

[embedded](#) [21](#)

351

EMPTY [269](#)

ENABLE [135](#)

Enterprise Manager [21](#)

[entiteit](#) [2, 3, 10](#)

entiteitsintegr[iteit](#) [5](#)

equi-[join](#) [155](#)

E**RM** [6, 16](#)

ERM diagr[am](#) [274](#)

ESCAPE [83](#)

EXCEPT [269](#)

exclusief [77](#)

EXECUTE [26](#)

EXIST**S** [192](#)

[exit](#) 34

[EXP](#) 98

[EXPLAIN](#) 142

[expliciet](#) 124

[expressie](#) 30

externe editor [37](#)

[EXTRACT](#) 109

F

[fill mode](#) 115

[flashback](#) 208

[FLASHBACK TABLE](#) 148, 210

[flexibiliteit](#) 10

[floating point](#) 27, 132

FLOOR [98](#)

[FOLLOWING](#) 206

[FORCE](#) 214

[foreign key](#) 10, 135

[FOREIGN KEY](#) 59

[format](#) 46

[FORMAT](#) 249

**FORTRAN** [21](#)

**foutmelding** [38](#)

**FROM** [M 24, 68](#)

**FULL OUTER JOIN** [162](#)

**functie** [31, 97](#)

**-gebaseerde indexen** [141](#)

**fysiek ontwerp** [p 16](#)

**G**

352

**gebonden variabelen** [191](#)

**gebruiker** [57](#)

**gebruikersvariabelen** [237](#)

**gecorreleerde subqueries** [191](#)

**gedistribueerde database** [14, 148, 221](#)

**gegevensonafhankelijkheid** [13, 14](#)

**generiek** [19](#)

**geneste tabellen** [255, 262](#)

**geordende paren** [10](#)

**gepartitioneerde outerjoins** [179](#)

**geserveerde woorden** [32](#)

[gesloten](#) 11

[gesorteerde vensters](#) 204

[GET](#) 44

[getallen](#) 27

[grant](#) 222

[GRANT](#) 25, 26

[GREATEST](#) 111

[groepsfuncties](#) 166

[GROUP BY](#) 24, 68, 164, 224

[grouping](#) 177

[sets](#) 179

[GROUPING\\_ID](#) 177

H

[haakjes](#) 31, 78

[having](#) 68

[HAVING](#) 24, 170

[heading](#) 70

[HEADING](#) 240, 249

[HEADSEP-karakter](#) 251

[hiërarchie](#) 197

histogram [103](#)

historische gegevens [19](#)

HOUR [109](#)

Human Resources [15](#)

I

IMMEDIATE [140](#)

impliciet [124](#)

impliciete datatype-conversie [97](#)

IN [81, 91, 269](#)

inapplicable [91](#)

353

inclusief [77](#)

IND [64](#)

index [141](#)

INDEX [26](#)

informatie [1](#)

-systeem [1, 6, 7](#)

inhoud [22](#)

INITCAP [101](#)

INITIALLY IMMEDIATE [140](#)

inline constraint [130, 135](#)

inline views [195, 197, 227](#)

[input](#) 39

INSERT [26, 39, 119](#)

instead of-triggers [225](#)

INSTR [101](#)

integriteit [4](#)

integriteitsonafhankelijkheid [14](#)

interactief [21](#)

InterOffice [15](#)

intersect [183, 269](#)

INTERSECT [224](#)

intersection [12](#)

interval [108](#)

INTERVAL [28, 60, 131](#)

IS NOT EMPTY [269](#)

[is null](#) 88

ISO/ANSI [132](#)

J

[join](#) 12, 155

[JUSTIFY](#) [249](#)

K

[kandidaatsleutel](#) [10](#)

ker[nel](#) [7, 14](#)

key preser[ved table](#) [224](#)

kolom

-[alias](#) [70, 75, 215](#)

-[kop](#) [69](#)

-[naam](#) [58](#)

-[specificatie](#) [58, 130](#)

-waard[den](#) [9](#)

[kraaienpoten](#) [16](#)

354

L

[LABEL](#) [253](#)

[LAG](#) [206](#)

[LAST\\_DAY](#) [109](#)

[LEAD](#) [206](#)

[LEAST](#) [111](#)

[LEFT outer](#) [join](#) [162](#)

lege string [87](#)

lege verzameling [192](#)

LENGTH [101](#)

LEVEL [199](#)

LIKE [82, 249](#)

linesize [250](#)

LINESIZE [240, 248](#)

list [35](#)

LN [98](#)

LOCALTIMESTAMPA [MP 29](#)

locking [27, 126, 128](#)

LOG [98](#)

LOGIN.SQL [46, 248](#)

logische gegevensonafhankelijkheid [22, 221](#)

logische operatoren [30](#)

logisch niveau [12](#)

logisch ontwerp [16](#)

LONG [240](#)

LOWER [101](#)

LPAD [101, 103, 200](#)

LTRIM[101](#)

M

machtsver[zameling](#) [269](#)

materialized views [229](#)

MAX [166, 206, 253](#)

MEMBER [269](#)

MERGE [123](#)

meta-[gegevens](#) [7, 62](#)

methoden [6, 256](#)

MIN [166, 206, 253](#)

minteken [239, 247](#)

minus [224](#)

MINUS[S](#) [183](#)

minute [109](#)

MOD [98](#)

355

model [1](#)

MODIFY [133](#)

MONTH [109](#)

MONTHS\_BETWEEN [109](#)

MULTISET [268](#)

[mutaties](#) [124](#)

N

named query [214](#)

[namen](#) [31](#)

natuurlijke join [13](#)

[nesting](#) [94, 167](#)

NEWLINE [249](#)

NEWPAGE [240, 248](#)

NEW\_TIME [109](#)

NEW\_VALUE [249](#)

NEXT\_DAY [109](#)

[nextval](#) [145](#)

NLS [45](#)

NLS\_CURRENCY [45](#)

NLS\_DATE\_FORMAT [45, 109, 113](#)

[nls\\_language](#) [114](#)

NLS\_LANGUAGE [45](#)

NLS\_NUMERIC\_CHARACTERS [27, 45](#)

NLS\_SESSION\_PARAMETERS [65, 109](#)

NLS\_SORT [106](#)

NLS\_TIME\_FORMAT [45](#)

NLS\_timestamp\_format [109](#)

non-padded [131](#)

NOPRINT [249](#)

normaalvor**m** [256](#)

normaliser**en** [6](#)

NOSORT [142](#)

not [79](#)

NOT [30](#)

not null [59](#)

NOT NULL [133](#)

NULL [87, 120, 133, 240, 249](#)

NULLIF [111](#)

NULLS FIRST [76, 210](#)

NULLS LAST [76](#)

NULL-waard**e** [13](#)

null-waarden [11, 73, 76, 86, 177, 193](#)

356

NUMBER [59, 166, 253](#)

**NUMFORMAT** [240](#)

**NUMWIDTH** [240](#)

**nvl** [167](#)

**NVL** [89, 111](#)

**NVL2** [89, 111](#)

**O**

**OBJ** [64](#)

**occurrence** [3, 10, 17](#)

**omgevings-variable** [248](#)

**ON** [159](#)

**ontbrekende informatie** [11, 13, 18](#)

**operatorand** [29](#)

**operator** [11, 24, 29](#)

**opgeslagen functie** [117, 260](#)

**optimizer** [14, 74, 142, 143, 197, 228](#)

**or** [77](#)

**OR** [30](#)

**ORDER BY** [24, 74, 97, 183](#)

**ordinaalgetal** [115](#)

**or replace** [214](#)

outerjoin [117](#), [161](#), [169](#), [184](#)

P

[padded](#) [131](#)

PAGE [252](#)

[pagesize](#) [250](#)

PAGESIZE [240](#), [248](#)

parameters [245](#)

parents [197](#)

partities [182](#)

PARTITION BY [205](#)

PAUSE [239](#), [240](#)

performance [4](#), [229](#)

[plan\\_table](#) [143](#)

PL/SQL [117](#), [256](#)

POSIX [105](#)

POWER[R](#) [98](#)

POWERMULTISET [269](#), [270](#)

powerset [269](#)

precedentie [31](#), [78](#)

PRECEDING [206](#)

357

[precisie 27, 131](#)

[predikaat 11](#)

[primaire sleutel 10, 13, 94, 135](#)

[primary key 59](#)

[privé-synoniemen 146](#)

[privileges 25, 57, 214](#)

[produkt 12](#)

[projectie 12](#)

[prompt 239](#)

[propositie 11](#)

[prototyping 6](#)

[prullenmand 148](#)

[pseudo-kolom 29, 72](#)

[publieke synoniemen 146](#)

[PURGE 148](#)

Q

[query 68](#)

[rewrite 230](#)

[quit 34](#)

R

[raadpleging](#) 21

[RAD](#) 6

[RANGE](#) 206

[RANK](#) 206

rapport[ten](#) 8

RA[W](#) 131

RD[BMS](#) 11, 12, 13, 14

[read consistency](#) 127, 207

READ ONLY [207](#)

reconstrueren[en](#) 208

recursieve relatie [16](#)

[recycle bin](#) 148, 210

[redundante opslag](#) 230

[redundantie](#) 4, 221

REFERENCE[S](#) 26

referentiële integriteit [5, 10](#)

refererende sleutels [10](#)

regelnummering [39](#)

regelnummers [35](#)

[REGEXP\\_INSTR](#) [104](#)

[REGEXP\\_LIKE](#) [83](#), [104](#)

358

[REGEXP\\_SUBSTR](#) [105](#)

[register](#) [248](#)

reguliere expressies [105](#)

rekenfuncties [98](#)

rekenkundige operatoren [29](#)

[relatie](#) [9](#)

[relationeel model](#) [9](#)

[relationele databases](#) [8](#)

[RENAME](#) [149](#)

[REPFOOTER](#) [251](#)

[REPHEADER](#) [251](#)

[replace](#) [104](#)

[REPLACE](#) [42](#), [101](#)

[REPORT](#) [253](#)

[responstijd](#) [141](#)

[restrictie](#) [12](#)

[REVOKE](#) [25](#), [27](#), [222](#)

right outer [join](#) 162, 180

[rijen](#) 9

[rollback](#) 121

ROLLBACK [23](#)

[rollen](#) 25

[rollup](#) 176

ROUND [98, 109](#)

ROW [253](#)

ROWS [206](#)

[rpad](#) 103

RPAD [101](#)

RTRIM [101](#)

RUN [38](#)

S

[save](#) 243

SAVE [42, 44](#)

[savepoints](#) 125

scalar subquery expr[essions](#) 195

SCAN [240](#)

[schema](#) 57

schrikkeljaar [110](#)

script [46, 243](#)

SECOND [109](#)

[select](#) 68

SELECT [23, 24, 26, 97](#)

359

SELECT\_CATALOG\_ROLE [62](#)

[selectie](#) 12

self-join [157](#)

[semantiek](#) 6

[sequences](#) 144

[set](#) 239

SET [121, 235](#)

set buffer [247](#)

[settings](#) 44

SET TRANSACTION [128](#)

SHOW [45, 235, 239](#)

SIGN [98](#)

SIN [98](#)

single row subquery [188](#)

SINH [98](#)

SKIPI [252](#)

[snapshot](#) [127](#), [207](#), [230](#)

soft box [16](#)

sorteren [76](#)

SPACE [240](#)

specifiek [3](#), [19](#)

SPOOL [47](#), [248](#), [251](#), [254](#)

spreadsheets [8](#)

SQL [3](#) [14](#)

SQL [89](#) [14](#)

SQL [92](#) [13](#)

SQL-buffer [34](#), [246](#)

SQL\*Plus [32](#)

SQLPROMPT [240](#)

SQL-standaard [128](#), [268](#)

SQLTERMINATOR [240](#)

SQRT [98](#)

start [44](#), [243](#)

STD [253](#)

**STDDEV** [166](#)

**STORAGE** [130](#)

stored query [214](#)

**STORE SET** [47](#)

**string** [27](#)

**structuur** [22](#)

**SUBMULTISET** [269](#)

subprocess [37](#)

subqueries [187, 194](#)

subquery [22, 92, 120, 195](#)

360

factoring [197](#)

**SUBST**[R](#) [101](#)

**SUM** [166, 206, 253](#)

**synoniemen** [145](#)

**syntax** [69](#)

**SYSDATE** [29, 72, 100](#)

**systeemvariabelen** [29, 234, 239](#)

**SYSTIMESTAMP** [29](#)

T

[tabel 9](#)

[-constraint 130, 135](#)

[TABLE-functie 264](#)

[TAN 98](#)

[TANH 98](#)

[tautologie 90](#)

[technieken 6](#)

[tegenstrijdigheden 4](#)

[tekst 27](#)

[-functies 101](#)

[THEN 84](#)

[thèta-join 155](#)

[tijdelijke tabellen 197](#)

[tijdsduur 27](#)

[tijdzone 131](#)

[timestamp 28, 209](#)

[TIMESTAMPA 60, 108](#)

[TIME ZONE 60](#)

[timezone\\_abbr 109](#)

[TIMING 240](#)

[tkprof](#) [142](#)

[TO\\_CHAR](#) [113](#)

[TO\\_DATE](#) [28](#), [113](#)

[toegangsrechten](#) [8](#)

[toegankelijkheid](#) [2](#), [13](#)

[TO\\_NUMBER](#) [R](#) [113](#)

[tools](#) [8](#)

[TRACE](#) [142](#)

[traceonly](#) [143](#)

[transactie](#) [23](#), [124](#)

[TRANSLATE](#) [101](#), [104](#)

[TRIMSPOOL](#) [248](#)

[TRUNC](#) [C](#) [98](#), [109](#)

361

[TRUNCATE](#) [124](#), [149](#)

[TRUNCATED](#) [249](#)

[TTITLE](#) [248](#), [250](#)

[tupels](#) [9](#)

[tupelvariabele](#) [152](#), [156](#), [191](#)

[tweewaardige logica](#) [90](#)

U

UML [6](#)

UNBOUNDED PRECEDING [205](#)

UNDEFINE [237, 246](#)

underscore [31](#)

[union](#) 12

UNION [183, 224, 269, 271](#)

UNIQUE [59](#)

unique identifier [17, 274](#)

updatable join views [223](#)

UPDATE [23, 26, 120](#)

UPPER [101](#)

UPSERT [123](#)

USER [R 29, 240](#)

USER\_CONS\_COLUMN [S 136](#)

user\_constraints [136](#)

USER\_RECYCLEBIN [148](#)

USING [159](#)

V

values [120](#)

[VALUES](#) [22](#)

[VAR](#) [253](#)

[VARCHAR](#) [131](#)

[VARCHAR](#) [259](#)

[variable](#) [28, 29, 234](#)

[VARIANCE](#) [166](#)

[vensters](#) [203](#)

[vereniging](#) [12, 182](#)

[vergelijgingsoperator](#) [30, 189](#)

[VERIFY](#) [236, 240](#)

[verschil](#) [12](#)

[-verzameling](#) [182](#)

[VERSIONS BETWEEN](#) [210](#)

[VERSIONS\\_ENDTIME](#) [210](#)

[VERSIONS\\_STARTTIME](#) [210](#)

362

[verzameling](#) [9](#)

[verzamelingenleer](#) [8](#)

[verzamelingsoperator](#) [en](#) [182](#)

[verzamelingswaardige attributen](#) [165](#)

verzamelingswaardige datatypes 255

vierwaardige logica 91

view 13, 22, 213

volgnummer 144

volgorde 9

volume 2, 229

voorrang 31

voorwaarde 30, 73

vraagtaLEN 8

vrije variabelen 191

W

wachtwoord 34, 57

WHEN 84

WHERE 24, 68, 97

wildcarDs 82

windowS 203

with 197

WITH 221

with check option 215, 226

WITH READ ONLY 215, 224

witruimte 69, 78

WORD\_WWRAPPED [249](#)

WRAPPED [249](#)

Y

year [108, 109](#)

YEAR [28](#)

Z

zelfgedefinieerde datatypes [264](#)

zoekpatroon [82, 105](#)

363



Leerboek Oracle SQL is in eerste instantie bedoeld voor het hoger onderwijs en is geschikt voor alle studierichtingen die Oracle als softwareomgeving gebruiken voor het leren omgaan met SQL. Het boek is met name geschikt als ondersteuning bij zelfstudie en/of practicum en als zodanig ook heel goed individueel te gebruiken buiten het reguliere onderwijs.

Deze vierde herziene druk is gebaseerd op Oracle Database 11g. Het boek is echter ook heel goed te gebruiken met de aankomende release Oracle12c.

De opzet van het boek is vrijwel geheel gehandhaafd. Hierdoor is een soepele overgang naar deze nieuwe editie mogelijk. De indeling van de hoofdstukken is niet veranderd, wel zijn in alle hoofdstukken op diverse plaatsen

tekstuele aanpassingen doorgevoerd om deze weer in overeenstemming te brengen met de huidige stand van de techniek. Verwijzingen naar inmiddels niet meer gangbare tools, zijn vervangen door hun opvolgers, en enkele nieuw geïntroduceerde SQL features zijn toegevoegd: het INSERT-ALL commando en een uitbreiding op de REGEXP functies zijn hiervan voorbeelden. De paragrafen over Constraints en over Hiërarchische Queries zijn geheel herschreven.

#### Online support

Scripts om de casustabellen te maken, alle voorbeelden en uitwerkingen van de opgaven zijn beschikbaar via de pagina bij het boek op [www.academicservice.nl](http://www.academicservice.nl).

#### Over de auteurs

Ir. Toon Koppelaars (1965) studeerde Informatica aan de Technische Universiteit van Eindhoven, en heeft meer dan twintig jaar ervaring met Oracle databases en ontwikkeltools. Hij is een veelgevraagd spreker op Oracle-bijeenkomsten, lid van het OakTable netwerk en Oracle ACE.

Ir. Lex de Haan (1954-2006) was als Oracle expert bekend als presentator en organisator van seminars. Hij was lid van de ISO-standaardisatiecommissie voor de taal SQL en hij was betrokken bij de oprichting van het internationale OakTable netwerk van Oracle-experts.

Van beide auteurs verscheen eerder *Applied Mathematics for Database Professionals* (Apress, New York, 2007).

ISBN 978 90 395 2681 1

NUR 123 / 991



## **Table of Contents**

Voorblad

2

Titelpagina

3

Copyright

4

Voorwoord

6

Inhoud

11

1. Inleiding relationele databasesystemen en Oracle

15

1.1 Informatiebehoefte en informatiesystemen

15

1.2 Databaseontwerp

17

1.3 Database-managementsysteem

23

## 1.4 Relationale databases

25

## 1.5 Relationale gegevensstructuur

26

## 1.6 Relationale operatoren

29

## 1.7 Hoe relationeel is mijn DBMS?

30

## 1.8 De Oracle-software

32

## 1.9 De casus

34

# 2. Kennismaking met SQL, SQL\*Plus en SQL Developer 40

## 2.1 Overzicht SQL

40

### 2.1.1 Databeeld

41

### 2.1.2 Datomanipulatie

41

### 2.1.3 Raadpleging

43

#### 2.1.4 Beveiliging

45

### 2.2 Enkele basisbegrippen

49

### 2.3 Kennismaking met SQL\*Plus

55

#### 2.3.1 De SQL-buffer

58

#### 2.3.2 Het gebruik van een externe editor

60

#### 2.3.3 De SQL\*Plus-editor

61

#### 2.3.4 Commando's bewaren

67

#### 2.3.5 SQL\*Plus-instellingen

69

#### 2.3.6 Nog een paar nuttige SQL\*Plus-commando's

73

### 2.4 Kennismaking met SQL Developer

75

365

3. Datadefinitie – deel I

84

3.1 Schema's en gebruikers

84

3.2 Tabellen maken

85

3.3 Datatypes

86

3.4 De casustabellen

88

3.5 De datadictionary

90

4. Raadpleging – de basis

97

4.1 Overzicht van de SELECT-componenten

97

4.2 De SELECT-component

99

4.3 De WHERE-component

105

4.4 De ORDER BY-component

106

4.5 AND, OR, NOT

109

4.6 BETWEEN, IN, LIKE

114

4.7 CASE-expressies

117

4.8 NULL-waarden

120

4.9 Subqueries

126

4.10 Opgaven

131

5. Raadpleging – functies

133

5.1 Inleiding

133

5.2 Rekenfuncties	
	135
5.3 Tekstfuncties	
	138
5.4 Reguliere expressies	
	141
5.5 Datumfuncties	
	146
5.6 Algemene functies	
	149
5.7 Conversiefuncties	
	151
5.8 Opgeslagen functies	
	155
5.9 Opgaven	
	157
6. Datamanipulatie	
	158
6.1 Het INSERT-commando	
	158

6.2 Het UPDATE-commando

160

6.3 Het DELETE-commando

161

6.4 Transactieverwerking

164

6.5 Read consistency en locking

167

366

7. Datadefinitie – deel II

171

7.1 CREATE TABLE

171

7.2 Datatypes

173

7.3 ALTER TABLE

175

7.4 Constraints

177

7.5 Indexen

186

7.6 Performance

189

7.7 Sequences

191

7.8 Synoniemen

192

7.9 DROP TABLE

195

7.10 Overige commando's

196

7.11 Opgaven

197

8. Raadpleging – meerdere tabellen en aggregatie

199

8.1 Tuple-variabelen

199

8.2 Joins

201

8.3 De ANSI/ISO standaard join syntax

206	
8.4 De outerjoin	
210	
8.5 De GROUP BY-component	
213	
8.6 Groepsfuncties	
215	
8.7 De HAVING-component	
220	
8.8 Extra mogelijkheden van de GROUP BY-component	
224	
8.9 Verzamelingsoperatoren	
233	
8.10 Opgaven	
236	
9. Raadpleging – enkele geavanceerde mogelijkheden	
238	
9.1 Subqueries: vervolg	
238	
9.2 Subqueries in de SELECT-component	

246	
9.3 Subqueries in de from-component	
248	
9.4 De WITH-component	
249	
9.5 Hiërarchische queries	
250	
9.6 Vensters en analytische functies	
257	
9.7 Flashback queries	
262	
9.8 Opgaven	
266	
10. Views	
268	
367	
10.1 Wat zijn views?	
268	
10.2 Toepassingsmogelijkheden	
274	

10.3 Datamanipulatie via views	
278	
10.4 De CHECK OPTION	
282	
10.5 Datamanipulatie via inline views	
285	
10.6 Views en performance	
286	
10.7 Materialized views	
287	
10.8 Opgaven	
290	
11. SQL*Plus en SQL Developer	
291	
11.1 SQL*Plus versus SQL Developer	
291	
11.2 SQL*Plus-variabelen	
292	
11.2.1 Substitutievariabelen	
293	

11.2.2 Gebruikersvariabelen

295

11.2.3 Systeemvariabelen

298

11.3 SQL\*Plus-scripts

302

11.4 Rapportage met SQL\*Plus

308

11.5 BREAK en COMPUTE

312

12. Object-relationele features

316

12.1 Nog meer datatypes

316

12.2 Arrays

318

12.3 Geneste tabellen

323

12.4 Zelfgedefinieerde types

326

12.5 Multiset-operatoren

328

Apendix A De casus

335

Index

347

Achterflap

364

368

# Document Outline

- [Voorblad](#)
- [Titelpagina](#)
- [Copyright](#)
- [Voorwoord](#)
- [Inhoud](#)
- [1. Inleiding relationele databasesystemen en Oracle](#)
  - [1.1 Informatiebehoefte en informatiesystemen](#)
  - [1.2 Databaseontwerp](#)
  - [1.3 Database-managementsysteem](#)
  - [1.4 Relationele databases](#)
  - [1.5 Relationele gegevensstructuur](#)
  - [1.6 Relationele operatoren](#)
  - [1.7 Hoe relationeel is mijn DBMS?](#)
  - [1.8 De Oracle-software](#)
  - [1.9 De casus](#)
- [2. Kennismaking met SQL, SQL\\*Plus en SQL Developer](#)
  - [2.1 Overzicht SQL](#)
    - [2.1.1 Databeeld](#)
    - [2.1.2 Datomanipulatie](#)
    - [2.1.3 Raadpleging](#)
    - [2.1.4 Beveiliging](#)
  - [2.2 Enkele basisbegrippen](#)
  - [2.3 Kennismaking met SQL\\*Plus](#)
    - [2.3.1 De SQL-buffer](#)
    - [2.3.2 Het gebruik van een externe editor](#)
    - [2.3.3 De SQL\\*Plus-editor](#)
    - [2.3.4 Commando's bewaren](#)
    - [2.3.5 SQL\\*Plus-instellingen](#)
    - [2.3.6 Nog een paar nuttige SQL\\*Plus-commando's](#)
  - [2.4 Kennismaking met SQL Developer](#)
- [3. Databeeld – deel I](#)
  - [3.1 Schema's en gebruikers](#)
  - [3.2 Tabellen maken](#)

- [3.3 Datatypes](#)
- [3.4 De casustabellen](#)
- [3.5 De datadictionary](#)
- [4. Raadpleging – de basis](#)
  - [4.1 Overzicht van de SELECT-componenten](#)
  - [4.2 De SELECT-component](#)
  - [4.3 De WHERE-component](#)
  - [4.4 De ORDER BY-component](#)
  - [4.5 AND, OR, NOT](#)
  - [4.6 BETWEEN, IN, LIKE](#)
  - [4.7 CASE-expressies](#)
  - [4.8 NULL-waarden](#)
  - [4.9 Subqueries](#)
  - [4.10 Opgaven](#)
- [5. Raadpleging – functies](#)
  - [5.1 Inleiding](#)
  - [5.2 Rekenfuncties](#)
  - [5.3 Tekstfuncties](#)
  - [5.4 Reguliere expressies](#)
  - [5.5 Datumfuncties](#)
  - [5.6 Algemene functies](#)
  - [5.7 Conversiefuncties](#)
  - [5.8 Opgeslagen functies](#)
  - [5.9 Opgaven](#)
- [6. Datamanipulatie](#)
  - [6.1 Het INSERT-commando](#)
  - [6.2 Het UPDATE-commando](#)
  - [6.3 Het DELETE-commando](#)
  - [6.4 Transactieverwerking](#)
  - [6.5 Read consistency en locking](#)
- [7. Datadefinitie – deel II](#)
  - [7.1 CREATE TABLE](#)
  - [7.2 Datatypes](#)
  - [7.3 ALTER TABLE](#)
  - [7.4 Constraints](#)
  - [7.5 Indexen](#)
  - [7.6 Performance](#)

- [7.7 Sequences](#)
- [7.8 Synoniemen](#)
- [7.9 DROP TABLE](#)
- [7.10 Overige commando's](#)
- [7.11 Opgaven](#)
- [8. Raadpleging – meerdere tabellen en aggregatie](#)
  - [8.1 Tuple-variabelen](#)
  - [8.2 Joins](#)
  - [8.3 De ANSI/ISO standaard join syntax](#)
  - [8.4 De outerjoin](#)
  - [8.5 De GROUP BY-component](#)
  - [8.6 Groepsfuncties](#)
  - [8.7 De HAVING-component](#)
  - [8.8 Extra mogelijkheden van de GROUP BY-component](#)
  - [8.9 Verzamelingsoperatoren](#)
  - [8.10 Opgaven](#)
- [9. Raadpleging – enkele geavanceerde mogelijkheden](#)
  - [9.1 Subqueries: vervolg](#)
  - [9.2 Subqueries in de SELECT-component](#)
  - [9.3 Subqueries in de from-component](#)
  - [9.4 De WITH-component](#)
  - [9.5 Hiërarchische queries](#)
  - [9.6 Vensters en analytische functies](#)
  - [9.7 Flashback queries](#)
  - [9.8 Opgaven](#)
- [10. Views](#)
  - [10.1 Wat zijn views?](#)
  - [10.2 Toepassingsmogelijkheden](#)
  - [10.3 Datamanipulatie via views](#)
  - [10.4 De CHECK OPTION](#)
  - [10.5 Datamanipulatie via inline views](#)
  - [10.6 Views en performance](#)
  - [10.7 Materialized views](#)
  - [10.8 Opgaven](#)
- [11. SQL\\*Plus en SQL Developer](#)
  - [11.1 SQL\\*Plus versus SQL Developer](#)
  - [11.2 SQL\\*Plus-variabelen](#)

- [11.2.1 Substitutievariabelen](#)
- [11.2.2 Gebruikersvariabelen](#)
- [11.2.3 Systeemvariabelen](#)
- [11.3 SQL\\*Plus-scripts](#)
- [11.4 Rapportage met SQL\\*Plus](#)
- [11.5 BREAK en COMPUTE](#)
- [12. Object-relationele features](#)
  - [12.1 Nog meer datatypes](#)
  - [12.2 Arrays](#)
  - [12.3 Geneste tabellen](#)
  - [12.4 Zelfgedefinieerde types](#)
  - [12.5 Multiset-operatoren](#)
- [Appendix A De casus](#)
- [Index](#)
- [Achterflap](#)