CURSISTENMAP

PL/SQL, de procedurele extensie

Versie 2.0, november-2012

Inhoudsopgave

Inhoudsopgave

Inhoudsopgave

Inhoudsopgave

1 Introductie tot de procedurele extensie

1.1 Inleiding

In de cursus PL/SQL hebben we de basis gelegd om te kunnen programmeren met PL/SQL. Bij het bouwen van applicaties komen de onderdelen van PL/SQL, zoals cursoren, lussen en exceptions, veelvuldig terug. Naast deze technische kennis is het van groot belang dat het PL/SQL-programma zo is opgebouwd dat het goed te onderhouden is en door anderen makkelijk te begrijpen is. Hiervoor is het noodzakelijk dat het PL/SQL-programma een duidelijke structuur heeft. In dit hoofdstuk zien we een aantal manieren waarmee we de structuur van een PL/SQL-programma kunnen verbeteren.

1.2 Structuur van een PL/SQL-blok

Een eenvoudige manier om PL/SQL-code leesbaarder te maken is door de code uit te lijnen. Ook een eenduidige en zelfverklarende naamgeving van variabelen kan de code een stuk leesbaarder maken.

Voorbeeld (u hoeft dit voorbeeld niet uit te voeren):

```
declare
  v_nummer number(3);
begin
  v_nummer:=&getal;
  if v_nummer > 100 then
    insert into hulptabel
      values (v_nummer, '> 100', null);
  elsif v_nummer > 50  then
    insert into hulptabel
      values (v_nummer, '> 50', null);
  elsif v_nummer > 0 then
    insert into hulptabel
      values (v_nummer, '> 0', null);
   else
    insert into hulptabel
      values (v_nummer, 'Negatief', null);
  end if;
end;

declare
var1 number(3);
begin
variabele1:=&getal;
if var2 > 100
then insert into hulptabel values (var1, '> 100', null);
else if var1 > 50 then
insert into hulptabel values (var1, '> 50',null);
else if var1 > 0 then insert into hulptabel values (var1, '> 0', null);
else
insert into hulptabel values (var2, 'Negatief', null); end if;
end if;
end if;
end;
```

Deze twee programma's hebben dezelfde werking. Bij het eerste voorbeeld is de structuur echter duidelijk te herkennen en daardoor makkelijker te onderhouden en aan te passen.

1.3 Modulair werken

Bij het schrijven van grote programma's kan het handig zijn om de code op te delen in verschillende stukken. We noemen dit modulair werken. Modulair werken betekent dat het complexe programma wordt opgedeeld in kleinere stukken, modules (ook wel routines genoemd), die weer andere modules kunnen aanroepen.

Door een modulaire opbouw te gebruiken voor de programma's zal het geheel overzichtelijker en beter te onderhouden worden. Verder zijn deze kortere programma's makkelijker te schrijven en de kans op fouten wordt hierdoor kleiner. Een ander voordeel van het gebruik van modules is dat de verschillende onderdelen binnen verschillende applicaties gebruikt kunnen worden.

Binnen PL/SQL kennen we procedures, functies en packages om de code op te slaan in kleinere onderdelen.

Procedure:	Dit is een programma dat opgeslagen is in de database. De procedure moet een naam hebben. In het programma van de procedure kunnen verschillende acties worden uitgevoerd. Door parameters kan er eventueel informatie worden meegegeven of teruggegeven.
Functie:	Dit is een programma dat één bepaalde waarde, bijvoorbeeld een nummer of een datum, teruggeeft. Een functie heeft altijd een naam en wordt onder deze naam opgeslagen in database. Net als bij de procedure kan er informatie aan de functie worden doorgegeven door parameters. Het teruggeven van een waarde is de specifieke eigenschap van een functie.
Package:	Een package is een verzameling van procedures, functies, cursoren, etc. die samen worden opgeslagen in de database. We kunnen dit eigenlijk geen echte module meer noemen, aangezien de package zelf bestaat uit modules. De structuur van een programma kan door het gebruik van packages sterk verbeteren.

In een volgend hoofdstuk gaan we verder in op het gebruik van packages.
Procedures, functies en packages worden gecompileerd in de database opgeslagen. Dit betekent dat, als we een procedure aanroepen, de code van deze procedure niet meer gecontroleerd hoeft te worden. Alle rechten, zoals de aanwezigheid van tabellen en dergelijke, zijn al gecontroleerd. Dit heeft een gunstige invloed op de performance. In deze cursus werken we met de taal PL/SQL om programmacode te schrijven, maar het is ook mogelijk om de taal C of Java te gebruiken en in de database op te slaan. Uiteindelijk moet echter wel SQL gebruikt worden in deze talen om de database te kunnen benaderen. Alle drie de soorten code kunnen overigens vanuit SQL aangeroepen en uitgevoerd worden met het CALL- statement (het CALL-statement wordt besproken in paragraaf 2.3.2).

Andere objecten, die behoren tot de procedurele extensie, zijn in de database opgeslagen triggers.

Trigger: Hierdoor kunnen bepaalde acties ondernomen worden wanneer bijvoorbeeld rijen aan een tabel worden toegevoegd of uit de tabel worden verwijderd. Een trigger kan worden gebruikt als beveiliging, bijvoorbeeld om te zorgen dat bepaalde gegevens niet gewijzigd kunnen worden. Verder kunnen triggers worden gebruikt als een soort controle om te kijken wie welke acties heeft ondernomen. Dit laatste noemen we auditing.

Een trigger voert een bepaald PL/SQL-programma uit, waarbinnen procedures, functies en DML-statements kunnen worden aangeroepen.

Daarnaast wordt aandacht besteed aan een aantal door Oracle geleverde standaard packages. Met deze standaard packages kan onder andere tekst op het scherm worden gezet en kunnen DDL-statements worden uitgevoerd binnen een PL/SQL-blok.

1 Introductie tot de procedurele extensie

2 Procedures en functies

2.1 Inleiding

Procedures en functies zijn beiden objecten die bestaan uit een PL/SQL-programma. Procedures en functies kunnen eenvoudig opnieuw uitgevoerd worden. Ze kunnen binnen een PL/SQL-programma worden gedeclareerd (lokaal) of in de database worden opgeslagen (stored). Wanneer procedures en functies zijn opgeslagen in de database behoren ze altijd bij een bepaalde gebruiker. Voor het benaderen van deze objecten kunnen eventueel synoniemen gebruikt worden. De objecten kunnen aangeroepen worden vanuit interactieve hulpmiddelen (zoals SQL Developer), vanuit applicaties of vanuit andere procedures of functies.

2.2 Voordelen van opgeslagen procedures en functies

Het belangrijkste voordeel van het gebruik van in de database opgeslagen procedures en functies is dat applicaties beter onderhouden kunnen worden. Een wijziging in een applicatie kan vaak doorgevoerd worden door opgeslagen objecten te wijzigen in plaats van een groot aantal modules binnen de applicatie. Daarnaast levert het opslaan van procedures en functies die veel SQL statements bevatten een belangrijk performance voordeel op. Wanneer een opgeslagen procedure of functie wordt uitgevoerd vanaf de clientmachine, wordt de hoeveelheid netwerkverkeer drastisch verminderd.

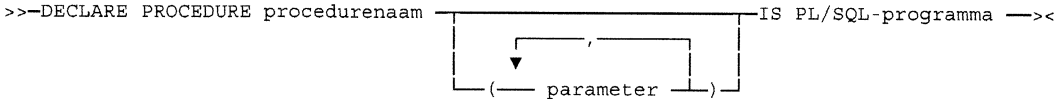
Grote stukken PL/SQL-code maken het lezen en onderhouden van een programma nodeloos ingewikkeld. Beter is het om PL/SQL code gemoduleerd op te bouwen. Dit bevordert de herbruikbaarheid en de leesbaarheid. Binnen PL/SQL kan gemoduleerd gewerkt worden door gebruik te maken van zogenaamde stored procedure (i.e. procedures, functies en packages).

2.3 Procedures

2.3.1 Lokale procedures

We kunnen procedures binnen een PL/SQL-programma op lokaal niveau gebruiken. De procedure of functie dient in de declaratiesectie bekend gemaakt te worden en kan dan binnen de programmasectie worden aangeroepen. Een lokale procedure wordt altijd pas gedeclareerd na alle variabelen, types en cursoren. Hetzelfde geldt voor lokale functies.

Syntax



Hieronder volgt een voorbeeld van een lokale procedure:

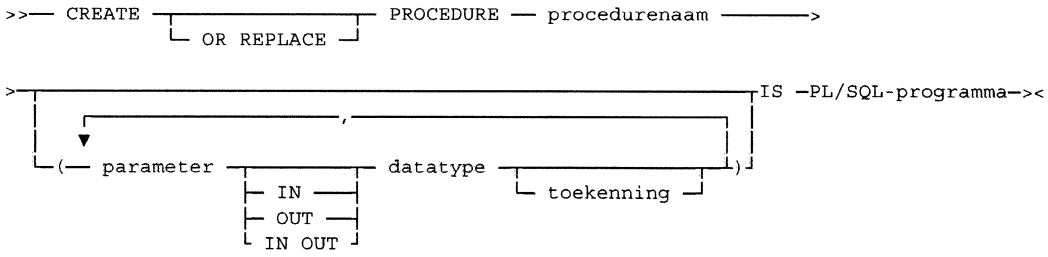
```
declare
  procedure gemiddelde_salaris
    (p_functie varchar2) is
    cursor c_wer (b_functie varchar2)is
      select sal
      from p_werknemers wer
      where upper(functie)=upper(b_functie);
    v_teller number:=0;
    v_totaal number:=0;
  begin
    for r_wer in c_wer(p_functie) loop
       v_teller:=v_teller+1;
       v_totaal:=v_totaal+r_wer.sal;
    end loop;
    insert into hulptabel values (null, p_functie, v_totaal/v_teller);
  end gemiddelde_salaris;
begin
  gemiddelde_salaris('VERKOPER');
  gemiddelde_salaris('KLERK');
end;
```

Lokale procedures moeten altijd als laatste in de declaratiesectie worden opgenomen. Hetzelfde geldt voor lokale functies die verderop in dit hoofdstuk nog behandeld worden.

2.3.2 Opgeslagen procedures

Procedures die in de database worden opgeslagen (stored procedures) kunnen gedefinieerd worden met behulp van het CREATE PROCEDURE statement.

Syntax



Direct na IS beginnen de declaraties; DECLARE is niet meer nodig. Indien een procedure gedefinieerd wordt die reeds bestaat, wordt in principe een foutmelding gegeven. Met de optie CREATE OR REPLACE wordt de procedure vervangen door de nieuwe procedure. Net als bij de parameters binnen een cursor mag ook bij de parameter van een procedure of functie alleen het datatype worden opgegeven en geen lengte:

```
(parameternaam VARCHAR2(3))
```

levert een foutmelding op.

Voorbeeld

```
create procedure werknemers_kantoor(p_kantnr in number)
```
Header

```
is
```

```
cursor c_werknemers_ophalen (b_kantnr number)is
       select naam
       ,      functie
       from p_werknemers
       where kantnr=b_kantnr;
v_kantoornr         number;
v_geldig            varchar2(6):='FALSE';
```
Declaraties

```
Begin
```

```
for r_werknemers_ophalen in c_werknemers_ophalen(p_kantnr)
loop
    v_geldig:='TRUE';
    insert into hulptabel
    values (p_kantoornummer, r_werknemers_ophalen.naam,
r_werknemers_ophalen.functie);
  end loop;
```
Uitvoering

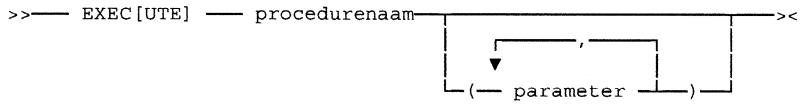
```
exception
```

```
when e_ongeldig_kantoor then
  insert into hulptabel
  values (p kantoornummer, 'Ongeldig kantoornummer', null);
```
Foutafhandeling

De opbouw van een opgeslagen procedure is bijna gelijk aan de opbouw van een PL/SQL-programma, alleen de header is erbij gekomen en het keyword DECLARE is niet meer nodig. Ook voor procedures geldt dat de uitvoering sectie SQL statements kan bevatten, maar dat dit niet noodzakelijk is.

Het uitvoeren van een procedure kan met behulp van het commando EXECUTE.

Syntax

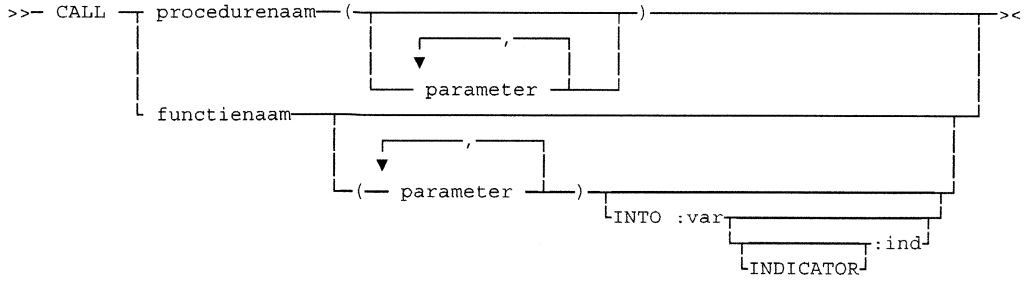


Bijvoorbeeld:

```
execute werknemers_kantoor(30)
```

Het is ook mogelijk om in plaats van EXECUTE het CALL- statement te gebruiken

Syntax



Met de optie `INTO :var` kan een returnwaarde worden opgevangen van een functie (meer over het gebruik van functies in paragraaf:2.4). Het declareren van de benodigde variabelen werkt hetzelfde als bij EXECUTE en wordt ook in paragraaf: 2.4 uitgelegd. Een indicator is niet van belang bij PL/SQL-routines. Bij het gebruik van andere talen kan een indicator gebruikt worden om aan te gegeven of de functie een NULL waarde retourneert.
Voorbeeld

```
CALL werknemers_kantoor(20);
```

Het verschil tussen CALL en EXECUTE is dat CALL een SQL statement is, waarmee een procedure of functie wordt aangeroepen. CALL werkt dus ook in andere tools waar SQL gebruikt wordt. EXECUTE is een SQL commando en werkt alleen in SQL Developer en SQL*Plus. EXECUTE is bedoeld om PL/SQL uit te voeren. U kunt er desnoods een compleet anoniem blok achter typen. EXECUTE zal een BEGIN ervoor zetten en een END aan het einde toevoegen en de code wordt vervolgens uitgevoerd zoals elke PL/SQL-code. Zoals u kunt zien in het syntax diagram van het CALL-statement, zijn de haken verplicht. Wanneer een procedure of functie zonder parameters aangeroepen wordt met behulp van CALL dan ziet dit er dus als volgt uit:

```
call procedure_zonder_parameter();
```

Zonder haken zal dit de volgende foutmelding opleveren:

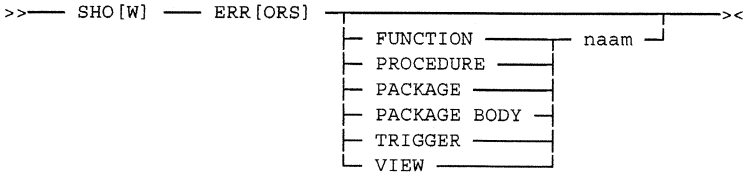
```
Error starting at line 1 in command:
call procedure_zonder_parameter;
Error report:
SQL Error: ORA-06576: not a valid function or procedure name
06576. 00000 -  "not a valid function or procedure name"
*Cause:    Could not find a function (if an INTO clause was present) or
           a procedure (if the statement did not have an INTO clause) to
           call.
*Action:   Change the statement to invoke a function or procedure
```

2.3.3 Fouten in de procedure

Wanneer Oracle fouten constateert bij het compileren dan zal de volgende melding op het beeldscherm verschijnen:

```
Warning: execution completed with warning
```

De procedure is dus wel degelijk opgeslagen in de database. Hetzelfde geldt voor de fouten. De fouten verschijnen niet meteen op het beeldscherm maar kunnen met SHOW ERRORS worden bekeken.
Syntax



Indien er geen argumenten worden meegegeven aan het commando, dan toont SHO[W] ERR[ORS] het meest recent gecompileerde object. De fouten kunnen ook bekeken worden via de datadictionary view USER_ERRORS. In deze cursus gebruiken we SHOW ERRORS omdat daarmee de fouten in een makkelijk te lezen formaat staan. Het formaat is het enige

verschil tussen SHO ERR en SELECT * FROM user_errors WHERE....
We gaan proberen onderstaande procedure aan te maken in de database:

```
create or replace procedure werknemers_fout
  (p_kantoornummer in number
  ) is
  cursor c_werkn (b_kantnr number) is
    select naam,      functie
    from p_werknemers werkn
    where kantnr=b_kantnr;
  v_kantoornr number;
  v_geldig varchar2(6):='FALSE';
  e_ongeldig_kantoor exception;
begin
  for r_werkn in c_werkn(p_kantoornummer) loop
    v_geldig:='TRUE'
    insert into hulptabel
      values (r_werkn.naam, r_werkn.functie, p_kantoornummer);
  end loop;
  if v_geldig='FALSE' then
    raise e_ongeldig_kantoor;
exception
  e_ongeldig_kantoor then
  insert into hulptabel
    values ('Ongeldig kantoornummer', p_kantoornummer, null);
end;
/
```

Na uitvoer met behulp van Run Script verschijnt de volgende tekst in het Run Script Output window:

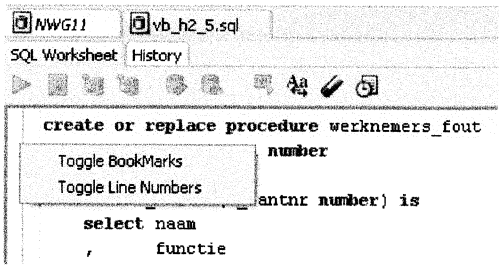
```
Warning: execution completed with warning
procedure werknemers_fout Compiled.
```

Met SHOW ERRORS kunnen we de fouten binnen deze procedure bekijken.

```
15/5            PLS-00103: Encountered the symbol "INSERT" when expecting one of the
following:

   * & = - + ; < / > at in is mod remainder not rem
   <an exponent (**)> <> or != or ~= >= <= <> and or like like2
   like4 likec between || multiset member submultiset
The symbol ";" was substituted for "INSERT" to continue.

20/1            PLS-00103: Encountered the symbol "EXCEPTION" when expecting one of
the following:

   ( begin case declare else elsif end exit for goto if loop mod
   null pragma raise return select update while with
   <an identifier> <a double-quoted delimited-identifier>
   <a bind variable> << continue close current delete fetch lock
   insert open rollback savepoint set sql execute commit forall
   merge pipe purge
The symbol "elsif was inserted before "EXCEPTION" to continue.

24/4            PLS-00103: Encountered the symbol ";" when expecting one of the
following:

   if
```

We zien een aantal malen bij de foutmelding: `when expecting one of the following:...`
Met een aantal suggesties wat er ingevuld zou kunnen worden, zoals een ; bij regel 15, **end** in regel 20 of **if** in regel 24. Een foutmelding met deze tekst heeft meestal betrekking op een stuk statement dat we vergeten hebben, zoals een ; of een END IF bij een IF statement.

Wanneer we in SQL Developer met de linker muisbutton in de kantlijn voor de code klikken dan kunnen we de Line Numbers activeren:



Dit heeft het volgende resultaat:

```
 1  create or replace procedure werknemers_fout
 2    (p_kantoornummer in number
 3    ) is
 4    cursor c_werkn (b_kantnr number)is
 5      select naam
 6      ,      functie
 7      from p_werknemers werkn
 8      where kantnr=b_kantnr;
 9    v_kantoornr number;
10    v_geldig varchar2(6):='FALSE';
11    e_ongeldig_kantoor exception;
12  begin
13    for r_werkn in c_werkn (p_kantoornummmer) loop
14      v_geldig:='TRUE'
15      insert into hulptabel
16          values (r_werkn.naam, r_werkn.functie, p_kantoornummer);
17    end loop;
18    if v_geldig='FALSE' then
19      raise e_ongeldig_kantoor;
20  exception
21    e_ongeldig_kantoor then
22    insert into hulptabel
23      values ('Ongeldig kantoornummer', p_kantoornummer, null);
24* end;
```

Regel 15: In regel 15 zit geen fout, maar in regel 14 mist een puntkomma (;).
Regel 20: Voor regel 20 ontbreekt een END IF.
Regel 24: In regel 24 geeft Oracle nogmaals aan de er een END IF ontbreekt. Door het toevoegen van END IF na regel 19 verdwijnt ook deze foutmelding.

Als we de genoemde fouten oplossen krijgen we na het starten de volgende foutmelding:

```
22/3          PLS-00103: Encountered the symbol "E_ONGELDIG_KANTOOR" when
expecting one of the following:

   pragma when
The symbol "when" was substituted for "E_ONGELDIG_KANTOOR" to continue.
```

In regel 22 hebben we de WHEN vergeten bij de afhandeling van de exception E_ONGELDIG_KANTOOR. Wanneer dit wordt aangepast kan de procedure worden aangemaakt zonder fouten. De procedure is nu syntactisch in orde, maar dat betekent niet altijd dat alle fouten eruit zijn.
We proberen de procedure WERKNEMERS_FOUT uit te voeren.

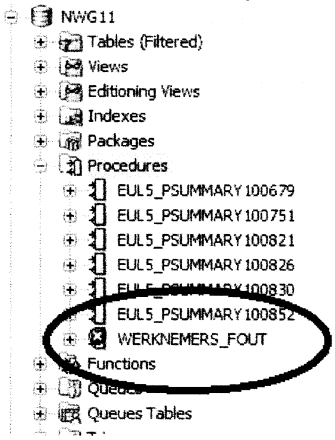
```
execute werknemers_fout(30)
```

Na afloop verschijnt de volgende foutmelding in het Run Script Output window:

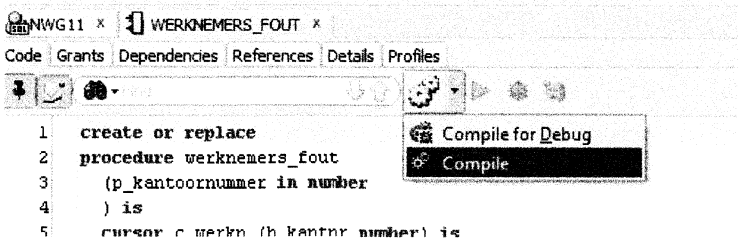
```
Error report:
ORA-01722: invalid number
ORA-06512: at "<USER>.WERKNEMERS_FOUT", line 15
ORA-06512: at line 1
01722. 00000 -  "invalid number"
*Cause:
*Action:
```

Er blijkt toch nog een fout in ons programma te zitten. Het betreft een semantische fout. Dit betekent dat hoewel de syntax goed is, er een fout optreedt doordat we een statement niet goed uitvoeren. Het INSERT statement, in regel 15, probeert de eerst kolom van HULPTABEL met tekst te vullen. Dit is echter een numerieke kolom. P_KANTOORNUMMER moet in de eerste kolom van HULPTABEL worden gezet. Om dit soort foutsituaties te voorkomen, is het van groot belang dat procedures en functies grondig getest worden.

Er is een tweede manier om in SQLDeveloper fouten op te sporen. Hiervoor open je de program unit (is een procedure/functie/package specificatie/package body) door erop te dubbelklikken. Indien een unit fouten heeft geeft SQLDeveloper dat hier al aan met een wit kruis in een rood rondje.



In het scherm wat dan opent kan gecompileerd worden met de radertjes bovenin:



De fouten verschijnen dan in de log console, daarnaast is in de code op de plaats(en) waar het fout gaat volgens Oracle een rood kriebeltje verschenen. De log kan eventueel van te voren geleegd worden met het menu van de rechtermuisknop. Nadeel van deze manier is dat wanneer de code uit een script kwam, dit script en de versie in de database uit elkaar gaan lopen. Wanneer de code ook in het script bewaard moet worden, zal deze aan het einde daarin teruggekopieerd moeten worden.

2.3.4 Parameters

Bij het creëren van een procedure of functie kunnen parameters gebruikt worden. Mogelijke typen zijn:

- IN — Dit type is de default waarde voor parameters. Een parameter van dit type is een invoerparameter en moet altijd gespecificeerd worden bij het uitvoeren van de betreffende procedure.
- OUT — Dit type is voor uitvoerparameters. De parameter krijgt een waarde bij uitvoering van de betreffende procedure. Bij het uitvoeren van de procedure moet er een variabele zijn waarin de waarde kan worden weggeschreven.
- IN OUT — Dit type is een combinatie van de andere twee typen.

Met behulp van parameters is het mogelijk om waarden door te geven aan en terug te krijgen van de aanroepende omgeving. Parameters worden na de procedurenaam en voor het IS keyword gedeclareerd. Wanneer parameters gedeclareerd worden, moet het datatype gespecificeerd worden. Daarbij is het niet toegestaan een maximale lengte op te geven, maar mogen %TYPE en %ROWTYPE wel gebruikt worden. Aan een parameter kan ook een DEFAULT waarde meegegeven worden, meer hierover in paragraaf 2.3.4.1.

We kunnen de procedure binnen SQL Developer uitvoeren met behulp van het EXECUTE statement. De OUT parameter moet in een variabele worden gezet. Hiervoor definiëren we eerst een variabele binnen SQL Developer.

```
variable hulpvar number
```

Daarna voeren we de procedure uit in een lege SQL Worksheet:

```
execute tel_werknemers(30,:hulpvar)   --de waarde van de OUT parameter komt in hulpvar
```

In het Run Script Output window zien we dat de procedure succesvol is uitgevoerd:

```
anonymous block completed
```

Vervolgens kunnen we met PRINT de waarde van een variabele in SQL Developer opvragen.

```
print hulpvar

hulpvar
-
6
```

2.3.4.1 DEFAULT waarden

We kunnen bij een IN-parameter een default waarde opgeven. Een default waarde kennen we toe door achter het datatype een waarde toe te kennen door:

```
parameternaam datatype:=waarde;
```

Of

```
parameternaam datatype DEFAULT waarde;
```

Let op dat OUT- en IN OUT-parameters geen default waarde kunnen krijgen.

Voorbeeld

```
create or replace procedure tel_werknemers
  ( p_kantoornr in number := 10
  , p_aantal out number
  ) is
  cursor c_aantal_wer (b_kantnr number)is
    select count(*)
    from p_werknemers wer
    where kantnr = b_kantnr;
begin
  open c_aantal_wer(p_kantoornr);
  fetch c_aantal_wer into p_aantal;
  close c_aantal_wer;
end;
/
```

Als we niet alle parameters een waarde geven, kunnen we aangeven welke parameter we gaan vullen door de parameternaam => waarde op te geven:

```
procedurenaam(parameternaam=>waarde, parameternaam=>waarde,...)
```

Parameters die niet in de aanroep worden meegegeven **moeten** een **default waarde** hebben.

Voorbeeld

```
declare
  v_aantal number;
begin
  tel_werknemers(30,v_aantal);
  insert into hulptabel
    values (null, v_aantal, 'Kantoor 30');
tel_werknemers(p_aantal => v_aantal);
 --we gebruiken de default waarde van p_kantoornr
  insert into hulptabel
    values (null, v_aantal, 'Default waarde => kantoor 10');
end;
/
```

2.3.5 Verwijderen van procedures

In de database opgeslagen procedures kunnen verwijderd worden met behulp van het DROP statement:

Syntax

```
>>—— DROP PROCEDURE — naam ——><
```

Voorbeeld

```
DROP PROCEDURE tel_werknemers;
```

2.3.6 Datadictionary views

Vanaf Oracle 9i kennen we de Datadictionary view USER_PROCEDURES. Hiermee kunnen we een overzicht verkrijgen van de eigen procedures. Tevens zijn de procedures, maar ook de functions en packages, die een gebruiker aanmaakt in de datadictionaryview USER_OBJECTS terug te vinden. .

```
desc user_objects

Name                           Null     Type
------------------------------ -------- ----------------------------------------
OBJECT_NAME                             VARCHAR2(128)
SUBOBJECT_NAME                          VARCHAR2(30)
OBJECT_ID                               NUMBER
DATA_OBJECT_ID                          NUMBER
OBJECT_TYPE                             VARCHAR2(19)
CREATED                                 DATE
LAST_DDL_TIME                           DATE
TIMESTAMP                               VARCHAR2(19)
STATUS                                  VARCHAR2(7)
TEMPORARY                               VARCHAR2(1)
GENERATED                               VARCHAR2(1)
SECONDARY                               VARCHAR2(1)
NAMESPACE                               NUMBER
EDITION_NAME                            VARCHAR2(30)

14 rows selected
```

Wanneer we nu de kolommen OBJECT_NAME en CREATED selecteren van alle objecten met als object type het type procedure, dan zien we alle procedures terug die zich op dit moment in uw schema bevinden.

```
SELECT substr(object_name,1,30) object_name
,      created
FROM user_objects
WHERE object_type='PROCEDURE';
```

Wanneer we bovenstaande query uitvoeren zou het resultaat als volgt kunnen zijn:

```
OBJECT_NAME                    CREATED
------------------------------ -------------------------
WERKNEMERS_KANTOOR             13-FEB-08
WERKNEMERS_FOUT                13-FEB-08
BEPAAL_PERCENTAGE              13-FEB-08
```

De inhoud / code van een procedure, functie of package kunnen we terug vinden in de data dictionary view USER_SOURCE. Deze view ziet er als volgt uit:

```
desc user_source

Name                           Null?    Type
------------------------------ -------- ----
NAME                           NOT NULL VARCHAR2(30)
TYPE                                    VARCHAR2(12)
LINE                           NOT NULL NUMBER
TEXT                                    VARCHAR2(4000)
```

De view USER_SOURCE bevat voor iedere regel code uit een procedure, functie, package, trigger of type een los record. Hierin wordt per regel code de naam en het type steeds herhaald. Daarom zijn vooral de kolommen LINE en TEXT interessant om te selecteren omdat ze unieke informatie bevatten:

2 Procedures en functies

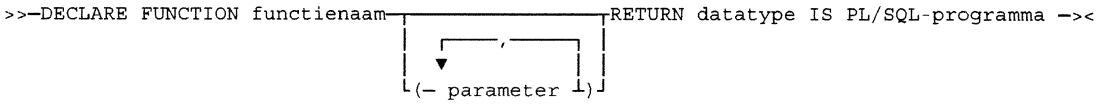
2.4 Functies

Een functie verschilt van een procedure omdat een functie altijd een resultaat teruggeeft. Een functie bevat dus altijd een RETURN-statement. Het is (helaas) zeer wel mogelijk om een functie aan te maken zonder RETURN-statement want daar wordt door de compiler niet op gecontroleerd. Runtime echter zal wel degelijk een foutmelding optreden. Zo gauw als een return is opgetreden stopt de executie. Eventuele code na het uitgevoerde RETURN-statement wordt niet meer uitgevoerd. Daarnaast is het mogelijk om in een SQL-statement een functie aan te roepen, terwijl dat bij een procedure niet mogelijk is.

2.4.1 Lokale functies

Net als procedures kunnen functies op lokaal niveau binnen een PL/SQL-programma worden gebruikt. De functie dient dan in de declaratiesectie bekend gemaakt te worden en kan dan binnen de programmasectie worden aangeroepen.

Syntax

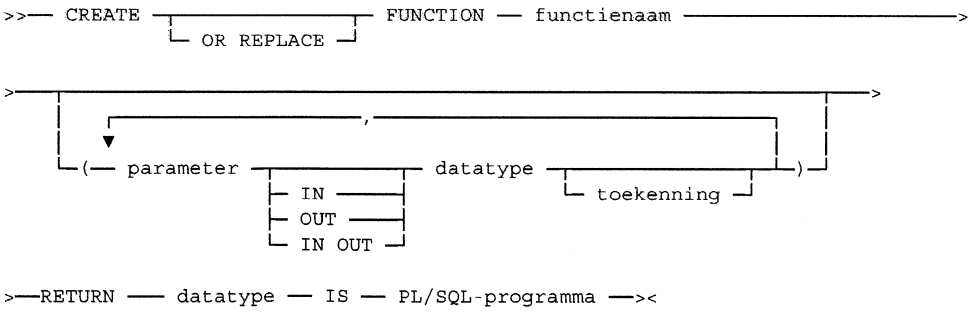


Direct na IS beginnen de declaraties. De lokale functie zelf wordt gedeclareerd nadat alle lokale variabelen, cursoren en type gedeclareerd zijn. Indien u dit niet doet volgt er een foutmelding.

2.4.2 Opgeslagen functies

Functies kunnen ook in de database worden opgeslagen. Hiervoor wordt gebruik gemaakt van het SQL statement CREATE FUNCTION.

Syntax



Direct na IS beginnen de declaraties.

Met CREATE OR REPLACE kan een procedure niet in een functie worden gewijzigd. De procedure moet eerst worden verwijderd voordat een functie met dezelfde naam kan worden aangemaakt.

De waarde die de functie oplevert moet in een variabele worden gezet. Het aanroepen van een functie is anders dan het aanroepen van een procedure met een OUT-parameter. De waarde van de functie wordt rechtstreeks toegekend aan de variabele.

Voorbeeld:

```
create or replace function tel_werknemers
  ( p_kantoornr in number:=10
  ) return number is
  cursor c_aantal_wer (b_kantnr number) is
    select count(*)
    from p_werknemers wer
    where kantnr = b_kantrnr;
  v_aantal number;
begin
  open c_aantal_wer (p_kantoornr);
  fetch c_aantal_wer into v_aantal;
  close c_aantal_wer;
  return v_aantal;
end;
/
```

Deze functie kunnen we als volgt aanroepen:

```
declare
  v_aantal number;
begin
  v_aantal:=tel_werknemers(30);
end;
```

2.4.3 Functies binnen SQL

Het is mogelijk om in een SQL-statement een GLOBALE functie aan te roepen. Dit is een functie die los in de database is opgeslagen. De functie kan in een SQL-statement overal geplaatst worden waar een Oracle functie (zoals UPPER of MOD) kan worden geplaatst. Zoals:

- in de SELECT, WHERE, GROUP BY, HAVING en ORDER BY clausules van een SELECT statement.
- in het SELECT statement van een view.
- in de CONNECT BY en START WITH clausules bij hiërarchische queries
- in de VALUES clausule van een INSERT statement
- in de SET clausule van een UPDATE statement.

Het is echter niet mogelijk om een functie te gebruiken in een checkconstraint of default-clausule van een CREATE TABLE-statement..

Voorbeeld

```
SELECT naam
,      percentage_berekenen(functie, salaris) percentage
FROM p_stafleden;
```

In het SELECT statement staan tussen haakjes de waarden van de parameters die mee moeten worden gegeven aan de functie PERCENTAGE. In het voorbeeld zijn dit kolommen uit de tabel P_STAFLEDEN.

De volgende voorwaarden zijn van toepassing op functies die op een dergelijke manier worden aangeroepen:

- Ze mogen geen OUT- of IN OUT-parameters bevatten. Alleen IN-parameters zijn toegestaan.
- Ze mogen de gegevens in de database niet aanpassen. Er mogen dus geen UPDATE-, INSERT- of DELETE-statements instaan.
- Ze mogen de huidige transactie niet afsluiten. COMMIT, ROLLBACK of SAVEPOINT mogen dus niet worden uitgevoerd.
- Ze mogen niet het systeem of de sessie wijzigen. Dus bijvoorbeeld geen SET ROLE en ALTER SYSTEM uitvoeren.

2.4.4 Verwijderen van functies

In de database opgeslagen functies kunnen verwijderd worden met behulp van het DROP-statement:

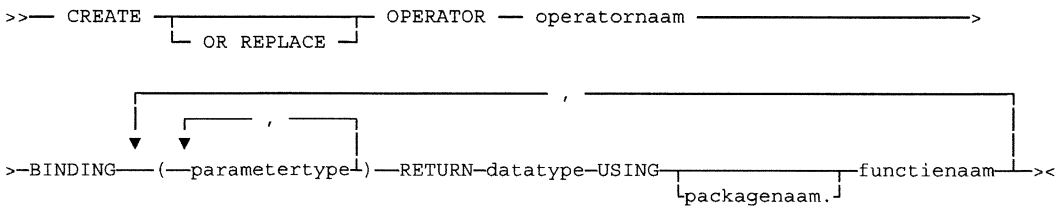
Syntax

```
>>—— DROP FUNCTION — naam ——><
```

2.5 Operatoren

Naast de standaard operatoren van Oracle, zoals + of >, kunnen binnen Oracle8*i* ook zelfgedefinieerde operatoren worden gebruikt. Een operator voert altijd een bewerking uit op één of meerdere parameters en geeft een resultaat terug. De bewerking die moet worden uitgevoerd is vastgelegd in een opgeslagen functie.
Operatoren kunnen in de SELECT-, WHERE-, GROUP BY-, HAVING- of ORDER BY-clausule van een SELECT-statement worden gebruikt.
Bij het definiëren van een operator moet worden opgegeven welke parameters de operator kan afhandelen. Afhankelijk van het aantal en het datatype van de parameters kan een andere functie worden aangeroepen. Het is niet mogelijk om twee dezelfde sets parameters aan een operator te koppelen. Het aantal of de datatypes moeten verschillen.

Syntax



parametertype	Het datatype van de parameters die worden afgehandeld door de operator.

Naast opgeslagen functies kunnen ook functies uit een package worden gebruikt. Packages worden in het volgende hoofdstuk behandeld.

Voorbeeld

We willen een operator PLUS maken die als de parameters twee getallen zijn deze getallen bij elkaar optelt. Als de parameters twee stukken tekst bevatten moet de operator deze aan elkaar koppelen. Dus plus(4,5) => 9 en plus('con', 'cat') => concat.

Voor deze bewerkingen moeten twee functies worden gedefinieerd. De functie PLUSNUMBER heeft als parameters twee getallen en de functie PLUSVARCHAR2 twee alfanumerieke parameters.

```
create or replace function plusnumber (f_nr1 number, f_nr2 number)
  return number is
begin
  return (f_nr1+f_nr2);
end;
/

create or replace function plusvarchar2 (f_tekst1 varchar2, f_tekst2 varchar2)
  return varchar2 is
begin
  return (to_char(to_number(f_tekst1) + to_number(f_tekst2)));
exception
  --Als het tekst is aan elkaar koppelen
  when value_error then return f_tekst1||f_tekst2;
end;
/
create operator plus
binding (number, number) return number using plusnumber,
        (varchar2, varchar2) return varchar2 using plusvarchar2;
```

Wanneer de bovenstaande functies en operator in de database worden aangemaakt dan kan de operator PLUS op de volgende manieren aangeroepen worden:

Allereerst met twee numerieke parameters:

```
SELECT plus(1,2) FROM dual;

PLUS(1,2)
----------------------
3
```

Vervolgens met twee alfanumerieke parameters :

```
SELECT plus('aan','een') FROM dual;

PLUS('AAN','EEN')
--------------------------------------------------------------------
aaneen
```

En tenslotte creëren we een fout situatie door een niet gedefinieerde combinatie van parameters op te geven:

```
SELECT plus('maal',2) FROM dual;
```

De volgende foutmelding zal in het Run Script Output window verschijnen:

```
Error starting at line 1 in command:
SELECT plus('maal',2) FROM dual
Error at Command Line:1 Column:7
Error report:
SQL Error: ORA-29900: operator binding does not exist
ORA-06553: PLS-307: too many declarations of 'PLUS' match this call
29900. 00000 -  "operator binding does not exist"
*Cause:    There is no binding for the current usage of the operator.
*Action:   Change the operator arguments to match any of the existing
           bindings or add a new binding to the operator.
ORA-06553: PLS-307: too many declarations of 'PLUS' match this call
```

Met BINDING wordt aangegeven welke datatypes bij deze operator kunnen worden gebruikt. In dit geval dus of twee numerieke waarden of twee alfanumerieke waarden.
De operator PLUS voert voor verschillende parameters verschillende functies uit. Dit principe noemen we overloading en komt in hoofdstuk 3 verder aan de orde.

2.6 Afhankelijkheden

In deze paragraaf zullen we de afhankelijkheden tussen de verschillende database procedures en functies bespreken. Hierbij kunnen we onderscheid maken tussen:

- directe afhankelijkheden
- indirecte afhankelijkheden

Het verschil tussen beide afhankelijkheden zullen we in de volgende subparagrafen toelichten.

2.6.1 Directe afhankelijkheden

Binnen een procedure of een functie kunnen we een andere functie of procedure aanroepen. De directe afhankelijkheden die een procedure of functie heeft worden bijgehouden in de datadictionary view USER_DEPENDENCIES.

```
desc user_dependencies
```

```
Name                           Null?    Type
------------------------------ -------- ----
NAME                           NOT NULL VARCHAR2(30)
TYPE                                    VARCHAR2(18)
REFERENCED_OWNER                        VARCHAR2(30)
REFERENCED_NAME                         VARCHAR2(64)
REFERENCED_TYPE                         VARCHAR2(18)
REFERENCED_LINK_NAME                    VARCHAR2(128)
SCHEMAID                                NUMBER
DEPENDENCY_TYPE                         VARCHAR2(4)
```

In USER_DEPENDENCIES staan alleen de directe afhankelijkheden.

2.6.2 Indirecte afhankelijkheden

Met de procedure DEPTREE_FILL kunnen we een overzicht krijgen van directe en indirecte afhankelijkheden. We geven bij de procedure op van welk object we de afhankelijke objecten willen zien, dus we beginnen bij het laagste niveau.

Voorbeeld

```
execute deptree_fill('PROCEDURE', user, 'BEPAAL_PERCENTAGE');
```

De procedure DEPTREE_FILL vult nu de tabel DEPTREE_TEMPTAB met de objecten die afhankelijk zijn van BEPAAL_PERCENTAGE. We moeten aangeven wat voor type object BEPAAL_PERCENTAGE is en in welk schema dit object staat.
In de view DEPTREE kunnen we zien welke objecten afhankelijk zijn van procedure BEPAAL_PERCENTAGE.

```
SELECT *
FROM deptree;
```

```
NESTED_LEVEL     TYPE         SCHEMA     NAME                SEQ#
---------------- ------------ ---------- ------------------- --------
0                PROCEDURE    <USER>     BEPAAL_PERCENTAGE   0
1                FUNCTION     <USER>     BEPAAL_VERHOGING    1
2                PROCEDURE    <USER>     LOONSVERHOGING      2
```

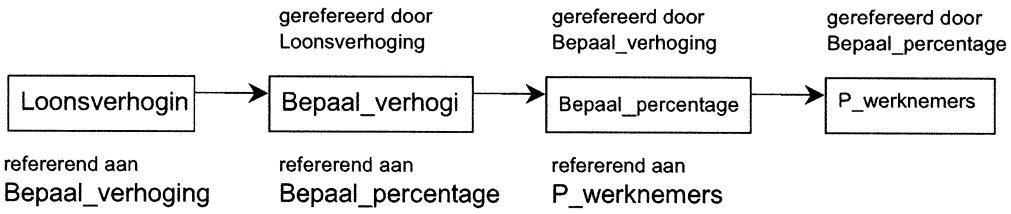
In de view IDEPTREE wordt een hiërarchisch overzicht gegeven van de onderlinge relaties. De onderste procedure LOONSVERHOGING roept de functie BEPAAL_VERHOGING aan en deze functie roept vervolgens weer de procedure BEPAAL_PERCENTAGE aan.

```
SELECT *
FROM ideptree;

DEPENDENCIES
----------------------------------------------------------------
   FUNCTION <USER>.BEPAAL_VERHOGING
      PROCEDURE <USER>.LOONSVERHOGING
PROCEDURE <USER>.BEPAAL_PERCENTAGE

3 rows selected
```

Om in de praktijk met DEPTREE_FILL te kunnen werken moet het script UTLDTREE.SQL worden gedraaid, dit script bevind zich normaal gesproken in de directory ORACLE_HOME\RDBMS\ADMIN.

2.6.3 Hercompilatie

Wat gebeurt er als een gerefereerd object wordt gewijzigd? Kunnen de afhankelijke objecten dan nog worden uitgevoerd?



LOONSVERHOGING is indirect afhankelijk van BEPAAL_PERCENTAGE. We zien ook dat alle objecten afhankelijk zijn van P_WERKNEMERS. Stel dat we de definitie van deze tabel aanpassen. We verbreden bijvoorbeeld de kolom PERSNR naar 6 posities:

```
ALTER TABLE p_werknemers MODIFY (persnr number(6));
```

Als we kijken in de view USER_OBJECTS, dan zien we dat alle procedures en functies die gebruik maken van de tabel P_WERKNEMERS de status INVALID hebben gekregen.

```
SELECT substr(object_name,1,30) objectname,status
FROM user_objects
WHERE object_type in ('PROCEDURE', 'FUNCTION')
AND status = 'INVALID'
/

OBJECT_NAME                    STATUS
------------------------------ -------
BEPAAL_PERCENTAGE              INVALID
BEPAAL_VERHOGING               INVALID
LOONSVERHOGING                 INVALID
```

Als er objecten op een remote database afhankelijk zijn, dan zullen die niet invalid worden.

Wanneer we nu een procedure uitvoeren, zal deze eerst automatisch gecompileerd worden. We starten bijvoorbeeld de procedure LOONSVERHOGING:

```
execute loonsverhoging
```

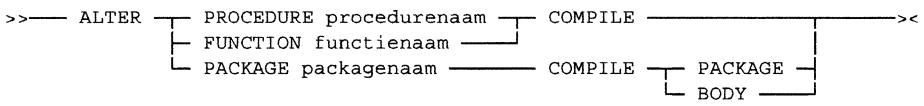
We bekijken vervolgens nogmaals de status van de verschillende procedures en functie met behulp van de view USER_OBJECTS:

```
SELECT object_name
,      status
FROM user_objects
WHERE object_type in ('PROCEDURE', 'FUNCTION');

OBJECT_NAME                    STATUS
------------------------------ -------
TEL_WERKNEMERS                 VALID
BEPAAL_PERCENTAGE              VALID
BEPAAL_VERHOGING               VALID
LOONSVERHOGING                 VALID
PLUSNUMBER                     VALID
PLUSVARCHAR2                   VALID
DEPTREE_FILL                   VALID
WERKNEMERS_KANTOOR             VALID
WERKNEMERS_FOUT                VALID
```

We zien dat niet alleen LOONSVERHOGING is gecompileerd maar ook de objecten die deze procedure aanroept, zoals BEPAAL_VERHOGING en BEPAAL_PERCENTAGE (zie de afhankelijkheid op pagina 2-15). Dit principe wordt automatische impliciete hercompilatie genoemd. Indien een procedure of functie in een package zit (zie het volgende hoofdstuk) dan zal de complete package gecompileerd worden. Het wijzigen van een functie of een procedure in een package maakt dan ook de gehele body invalid.

In de praktijk wordt meestal niet gewacht totdat INVALID objecten automatisch worden gecompileerd. Het compileren van de procedure of functie gebeurt dan handmatig met ALTER PROCEDURE ...COMPILE om te zien of er echte problemen zijn.

Syntax



Voorbeeld:

```
ALTER FUNCTION tel_werknemers COMPILE;
```

Als er veel objecten INVALID zijn, is het natuurlijk veel werk om deze allemaal met de hand te compileren. Oracle levert een procedure COMPILE_SCHEMA uit de standaard package DBMS_UTILITY waarmee alle objecten binnen een bepaald schema, dus alle objecten van een bepaalde gebruiker, kunnen worden gecompileerd. Het gebruik van deze package en andere standaard packages wordt in hoofdstuk 4 nader uitgelegd.

Voorbeeld

```
execute dbms_utility.compile_schema(user)
```

De code van het afhankelijke object moet worden herschreven als:

- Het gerefereerde object is verwijderd uit de database.
- Het gerefereerde object een andere naam heeft gekregen.
- De benodigde rechten zijn ingetrokken.
- De parameterlijst van de het gerefereerd object is het gewijzigd.

U kunt al bij voorbaat programmeren om hercompilatie zoveel mogelijk succesvol te laten voorlopen:

- Gebruik %TYPE en %ROWTYPE variabelen.
- Gebruik select * in queries.
- Gebruik een column list bij insert statements.

2.6.4 Remote afhankelijkheden

Afhankelijkheden tussen PL/SQL subprogramma's via een remote database kunnen op twee manieren afgehandeld worden. De ene manier is via timestamps (de default) en de andere manier via een signature (letterlijk handtekening).

Wanneer je gebruik maakt van timestamps en er een programmaeenheid of afhankelijk schemaobject gewijzigd wordt, dan worden alle afhankelijke eenheden invalid. De afhankelijke eenheden kunnen pas weer uitgevoerd worden na hercompilatie. Iedere programma eenheid heeft namelijk een timestamp die door de server bepaald wordt en overeenkomt met de tijd waarop de eenheid aangemaakt of gehercompileerd is.

Dit systeem heeft een aantal nadelen. Hercompilatie over het netwerk wordt vaak uitgevoerd wanneer het niet strikt noodzakelijk is, wat leidt tot performance verlies. Verder kan het timestampmodel aan de client-kant leiden tot situaties waarbij de applicatie helemaal niet meer kan draaien. Een applicatie zoals Forms Runtime beschikt bijvoorbeeld niet over een PL/SQL compiler.

Om sommige van deze problemen op te lossen is er in Oracle de mogelijkheid van het signaturemodel gebruik te maken voor remote afhankelijkheden. Dit signaturemodel is niet van toepassing op lokale afhankelijkheden (op dezelfde server), aangezien hercompilatie altijd mogelijk is in deze omgeving. Een signature wordt meegegeven aan ieder gecompileerd opgeslagen subprogramma. Het identificeert de eenheid met behulp van de volgende criteria:

- de naam van de eenheid (package, procedure of functie naam)
- de typen van iedere parameter van het subprogramma
- de modes van de parameters (IN, OUT, IN OUT)
- het aantal parameters
- het type van de return waarde van een functie

De gebruiker kan zelf bepalen of het signature- of timestampmodel gebruikt moet worden via de dynamische initialisatieparameter `REMOTE_DEPENDENCIES_MODE`. Voor het wijzigen van de mode in de huidige sessie kan het volgende DDL-statement gebruikt worden.

```
ALTER SESSION SET REMOTE_DEPENDENCIES_MODE = { TIMESTAMP | SIGNATURE }
```

Voorbeeld

Er is een databaselink 'LINK' aangemaakt tussen de lokale database LOKAAL en de remote database REMOTE.

Op de remote database wordt de functie F_REMOTE aangemaakt.

```
create or replace function f_remote return varchar2 is
  begin
   return 'De functie f_remote is aangeroepen';
end;
```

Op de lokale database wordt de procedure P_LOKAAL aangemaakt, die de functie F_REMOTE aanroept.

```
create or replace procedure p_lokaal is
   v_out varchar2(100);
 begin
   v_out:=f_remote@link;
   dbms_output.put_line(v_out);  -- put_line zet de waarde van v_out op het scherm
 end;
 /
```

Wanneer op de lokale database nu de procedure P_LOKAAL wordt uitgevoerd dan krijgen we het volgende resultaat te zien:

```
execute p_lokaal

anonymous block completed
De functie f_remote is aangeroepen
```

Wanneer de functie F_REMOTE opnieuw gecompileerd wordt (zonder de code te wijzigen) verandert de timestamp, maar niet de signature. Het aanroepen van de procedure P_LOKAAL levert in het timestampmodel (de default) een foutmelding op.

```
execute p_lokaal

Error report:
ORA-04062: timestamp of function "NWG801.F_REMOTE" has been changed
ORA-06512: at "NWG806.P_LOKAAL", line 4
ORA-06512: at line 1
04062. 00000 -  "%s of %s has been changed"
*Cause:    Attempt to execute a stored procedure to serve
           an RPC stub which specifies a timestamp or signature that is
           different from the current timestamp/signature of the procedure.
*Action:   Recompile the caller in order to pick up the new timestamp.
```

Pas na hercompilatie van de procedure P_LOKAAL is de procedure weer valid. Wanneer we echter het signature model gebruiken, vindt er geen foutmelding plaats. De signature van F_REMOTE is immers ongewijzigd gebleven. Wanneer de signature van de functie F_REMOTE wel gewijzigd wordt, door bijvoorbeeld het return datatype te wijzigen, vindt er wel een foutmelding plaats (`ORA-04062: signature of function "NWG801.F_REMOTE" has been changed`).

```
alter session set REMOTE_DEPENDENCIES_MODE = signature;

alter session succeeded.

execute p_lokaal

Error report:
ORA-04062: signature of function "NWG801.F_REMOTE" has been changed
ORA-06512: at "NWG806.P_LOKAAL", line 4
ORA-06512: at line 1
04062. 00000 -  "%s of %s has been changed"
*Cause:    Attempt to execute a stored procedure to serve
           an RPC stub which specifies a timestamp or signature that is
           different from the current timestamp/signature of the procedure.
*Action:   Recompile the caller in order to pick up the new timestamp.
```

3 Packages

3.1 Inleiding

Procedure en functies kunnen logisch gezien bij elkaar horen. In een applicatiesysteem zullen dit soort situaties natuurlijk vaak voorkomen. Om deze objecten zo goed mogelijk te kunnen beheren worden deze objecten vaak in een package geplaatst.

Een package is een verzameling die de volgende objecten kan bevatten:

- procedures;
- functies;
- types;
- constanten en variabelen;
- cursoren;
- exceptions.

3.2 Voordelen van een package

Het onderbrengen van logisch gezien bij elkaar horende procedures en functies in een package heeft de volgende voordelen:

- De betreffende objecten kunnen makkelijker beheerd worden. In plaats van het onderhouden van losse objecten kunnen er nu een kleiner aantal objecten, namelijk packages, worden onderhouden. Het uitdelen van rechten is ook eenvoudiger. Het is mogelijk om een gebruiker de bevoegdheid te geven om een bepaalde package te gebruiken in plaats van op de afzonderlijke procedures en functies.
- De beveiliging van een package is zodanig, dat alleen de eigenaar van de package de package body kan zien. Op die manier kan een applicatie worden beveiligd.
- Bij het gebruik van packages wordt bij het aanroepen van één onderdeel uit package de gehele package in het geheugen geladen. Aangezien de objecten in de package een logische samenhang vertonen, zullen deze vaak een ander object uit de package aanroepen. Deze hoeft dan niet meer van schijf gelezen te worden wat gunstig is voor de performance.
- Behalve uit een verzameling van objecten bestaat een package ook uit een specificatie. Door de scheiding tussen een package specificatie en een package body is het bijvoorbeeld mogelijk om aanvankelijk alleen de specificatie te definiëren en op een later tijdstip de procedures en de functies die tot de package behoren. Voordeel hiervan is dat in de tussentijd al aanroepen van de betreffende procedures en functies opgenomen kunnen worden in applicaties.

  Zolang de specificatie VALID is, zullen de aanroepende procedures ook VALID zijn. Ook als in de package body iets wordt gewijzigd. De aanroepende procedures hoeven niet opnieuw gecompileerd te worden.
- Public of Global variabelen en cursors bestaan tijdens de hele sessie. Als de waarde van een variabele verandert tijdens een sessie wordt verder gegaan met die nieuwe waarde. De waarden van de variabelen blijven tijdens verschillende transacties bestaan zonder dat deze in de database moeten worden vastgelegd.

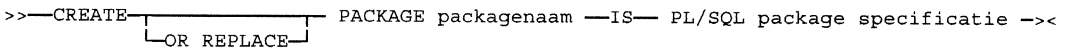
Wanneer een sessie wordt afgesloten zal de waarde van een variabele weer gereset worden, dit gebeurt ook wanneer de package specificatie wordt overschreven

- Binnen packages en binnen de declaratiesectie van een PL/SQL-blok kunnen verschillende procedures of functies met dezelfde naam worden gemaakt. Dit heet overloading. De parameterlijsten van de procedures of functies moeten dan wel verschillen. Een voorbeeld hiervan is bijvoorbeeld de functie TO_CHAR van Oracle We kunnen bij deze functie een nummer opgeven maar ook een datum.

3.3 Package specificatie

Om een package op te slaan in de database moet eerst de specificatie gedefinieerd worden met behulp van het CREATE PACKAGE statement.

Syntax



Bij een pacakge specificatie hoort meestal ook een package body, maar het is ook mogelijk om een package te maken die alleen uit een specificatie bestaat.

Voorbeeld

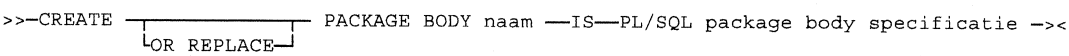
```
create or replace package specificatie is
  max_aantal_cursisten constant number:=50;
  max_aantal_dagen constant p_cursussen.aant_dag%type:=99;
  eerste_boeking constant date:=add_months(sysdate,12);
  laatste_boeking constant date:=sysdate+1;
  e_verkeerd_datatype exception;
  pragma exception_init(e_verkeerd_datatype, -1722);
end;
/
```

In deze specificatie zijn een aantal variabelen met een vaste waarde en een exception die aan de foutmelding ORA-01722, invalid number, een duidelijke naam geeft.
Aangezien in de specificatie geen procedures of functies staan is het niet nodig een package body te maken. Het voordeel van het gebruik van deze package is dat er een groot aantal constanten kan worden gedefinieerd. Deze constanten kunnen in allerlei programma's gebruikt worden. Mocht een waarde van die constante veranderen, bijvoorbeeld het aantal cursisten wordt verhoogd van 50 naar 100, dan hoeft de waarde alleen aangepast worden in deze package en niet alle afzonderlijke programma's die van deze constante gebruik maken.

3.4 Package body

Nadat een package specificatie gedefinieerd is kunnen we met behulp van het CREATE PACKAGE BODY statement de procedures en/of functies in de package definiëren.

Syntax



De package body brengt de package specificatie tot uitvoering. Elke procedure, functie of cursor die in de specificatie is genoemd, wordt in de package body gedefinieerd. De package specificatie en de package body moeten daarom gesynchroniseerd zijn. PL/SQL vergelijkt de headers van de programma's uit de specificatie met die in de body. Als die twee niet exact gelijk zijn volgt er een foutmelding.

Zet daarom de specificatie en de body in één commandobestand, zodat eenvoudig gecontroleerd kan worden of de headers gelijk zijn. Kopieer de header meteen als deze is aangepast. Op die manier zijn de headers in de specificatie en de body altijd gelijk.

Stel dat eerst de procedure BEPAAL_PERCENTAGE is opgenomen in de body, vervolgens de functie BEPAAL_VERHOGING en als laatste de procedure LOONSVERHOGING. Dit is ook de volgorde waarin ze van elkaar afhankelijk zijn. Wanneer deze routines alledrie in de specificatie zijn opgenomen is dit niet relevant; elke willekeurige volgorde zou goed zijn geweest.

Stel nu dat eindgebruikers alleen de procedure LOONSVERHOGING mogen uitvoeren. Dit is eenvoudig te realiseren door alleen LOONSVERHOGING op te nemen in de specificatie en BEPAAL_VERHOGING en BEPAAL_PERCENTAGE alleen in de body op te nemen. Echter omdat de routines van elkaar afhankelijk zijn, zou dan deze volgorde van declareren wel verplicht zijn.

Anders gezegd kun je in een procedure of functie binnen een package niet verwijzen naar een procedure of functie die nog niet gedefinieerd is. Dit definiëren mag zowel in de specificatie als in de body gedaan zijn. Voor routines die niet van elkaar afhankelijk zijn, maakt de volgorde waarin ze worden opgenomen niet uit.
Hieruit volgt automatisch dat procedures en functies die naar elkaar verwijzen (mutually referential subprograms) dit alleen kunnen doen als minimaal één van beiden in de specificatie van de package is opgenomen.

Zoals uit bovenstaande blijkt kan in een package body dus code staan die niet gedeclareerd is in de specificatie. De code is in dit geval private en er mag alleen aan gerefereerd worden door andere constructs die zich in de package bevinden. Wanneer een package construct (o.a. procedure, functie, variabele, constante) zowel in de package specificatie als package body is gedeclareerd, is het een publieke construct en kan deze ook buiten de package gebruikt worden.

Voorbeeld

We kunnen zo dus ook private constanten of variabelen aanmaken, die dus alleen binnen de package body bekend zijn.:

```
create or replace package voorbeeld_teller2 is
  procedure uitvoeren;
end;
/

package voorbeeld_teller2 Compiled.

create or replace package body voorbeeld_teller2 is
  v_teller number:=0;
 procedure uitvoeren is
 begin
   v_teller:=v_teller+1;
   insert into hulptabel(kolom1) values (v_teller);
 end uitvoeren;
end;
/
package body voorbeeld_teller2 Compiled.
```

Dit statement laat de specificatie en de body zien van de package VOORBEELD_TELLER2. U ziet dat ook in de body gewoon variabelen kunnen worden gedeclareerd. Dit soort code wordt uitgevoerd wanneer binnen een sessie een package voor het eerst wordt uitgevoerd.

3.5 Het aanroepen van een functie binnen SQL

We hebben in hoofdstuk 2 gezien dat we een opgeslagen functie ook kunnen gebruiken binnen SQL. Functies kunnen ook vanuit een package worden uitgevoerd in SQL. Er zijn een aantal voorwaarden waaraan een functie binnen een package moet voldoen voordat hij gebruikt kan worden binnen SQL.

- De functie mag geen database tabellen wijzigen; er mag geen UPDATE-, INSERT- of DELETE-statement voorkomen in de functie.
- Als de functie waarden van variabelen uit de package leest of aanpast dan kan deze niet via een databaselink (remote) worden gebruikt.
- Als de functie waarden van variabelen uit de package aanpast dan kan deze alleen voorkomen in de SELECT-, VALUES- of SET-clausules en niet in de WHERE of GROUP BY.
- De functie mag geen subprogramma's aanroepen die niet aan de bovenstaande regels voldoen.

3.6 RAISE_APPLICATION_ERROR

Tot nu toe hebben we foutsituaties binnen een programma steeds weggeschreven naar HULPTABEL. Ideaal is dit natuurlijk niet. Het zou handig zijn om meteen een melding op het scherm te krijgen bij een foutsituatie. Dit kan met behulp van de procedure RAISE_APPLICATION_ERROR.

Syntax

```
RAISE_APPLICATION_ERROR(fout_nummer, 'tekst')
```

FOUT_NUMMER moet tussen de -20000 en -20999 liggen.

Voorbeeld:

```
create or replace package aanvraag is
 procedure boeking_curdag( p_persnr in number, p_cursusnr in number, p_dag in date);
end;
/
show errors
-- ---------------------------------------------------------------------------
create or replace package body aanvraag is
 procedure boeking_curdag( p_persnr in number, p_cursusnr in number, p_dag in date)
  is
       cursor c_aantal_cur (b_dag date)is
         select count(distinct per_nr) aantal
         from p_boekingen
         where datum=b_dag;
       r_aantal_cur c_aantal_cur%rowtype;
     begin
       open c_aantal_cur(p_dag);
       fetch c_aantal_cur into r_aantal_cur;
       if r_aantal_cur.aantal >= specificatie.max_aantal_cursisten then
         close c_aantal_cur;
         insert into hulptabel
           values (p_persnr, 'Cursist kan niet meer geplaatst worden',
                   to_char(p_dag));
         raise_application_error(-20010,'Er zijn meer dan '
                    ||specificatie.max_aantal_cursisten||
                    ' cursisten, deze dag is vol');
```

```
      else
        insert into p_boekingen
          values (p_persnr, p_cursusnr, p_dag);
      end if;
      close c_aantal_cur;
    end;
  end;
  /
```

Wanneer we de procedure BOEKING_CURDAG uit de package AANVRAAG vervolgens aanroepen, dan zien we het effect van de RAISE_APPLICATION_ERROR:

```
execute aanvraag.boeking_curdag(1,4, TO_DATE('03-MAY-99','DD-MON-YYYY'))

ORA-20010:  Er zijn meer dan2 cursisten, deze dag is vol
```

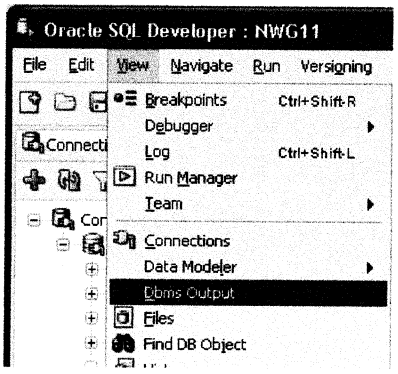
Bij een RAISE_APPLICATION_ERROR stopt het programma, en waarden die zijn toegekend aan OUT of IN OUT parameters worden ongedaan gemaakt. DML-statements, zoals INSERT en UPDATE, worden ook teruggedraaid.

Er komt een melding op het scherm net als bij een Oracle foutmelding. Het is mogelijk om deze melding af te vangen met behulp van PRAGMA EXCEPTION_INIT op dezelfde manier als we de standaard Oracle foutmeldingen kunnen afvangen. Deze PRAGMA zorgt ervoor dat bij de foutmelding er naar een EXCEPTION wordt gesprongen in plaats van dat er een fout optreedt.

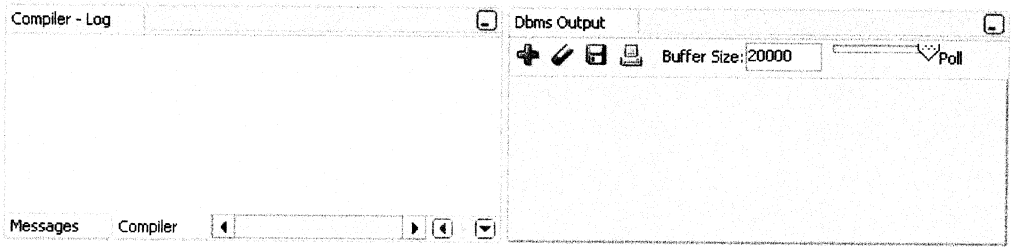
Als de RAISE_APPLICATION_ERROR door een exception wordt afgehandeld, dan worden de DML-statements niet teruggedraaid.

```
create or replace package body aanvraag is
  procedure boeking_curdag( p_persnr in number, p_cursusnr in number, p_dag in date)
    is
      cursor c_aantal_cur (b_dag date)is
        select count(distinct per_nr) aantal
        from p_boekingen
        where datum=b_dag;
      r_aantal_cur c_aantal_cur%rowtype;
      e_te_veel_cursisten exception;
      pragma exception_init(e_te_veel_cursisten, -20010);

    begin
      open c_aantal_cur(p_dag);
      fetch c_aantal_cur into r_aantal_cur;
      if r_aantal_cur.aantal >= specificatie.max_aantal_cursisten then
        close c_aantal_cur;
       insert into hulptabel
         values (p_persnr, 'Cursist kan niet meer geplaatst worden',
  to_char(p_dag));
        raise_application_error(-20010,' Er zijn meer dan '
                ||specificatie.max_aantal_cursisten||' cursisten, deze dag is vol');
      else
        insert into p_boekingen
          values (p_persnr, p_cursusnr, p_dag);
      end if;
      close c_aantal_cur;
    exception
      when e_te_veel_cursisten then
      dbms_output.put_line(sqlerrm); --SQLERRM geeft de melding van RAISE_APPLICATION_ERROR
    end;
  end;
  /
```

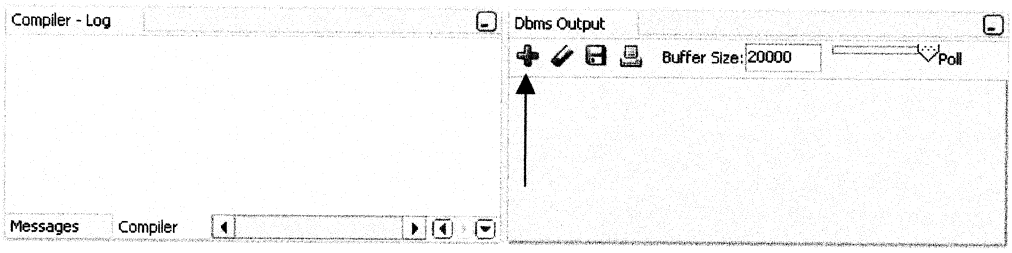
Binnen bovenstaande package wordt gebruik gemaakt van de standaard package DBMS_OUTPUT. In hoofdstuk 6 wordt het gebruik van standaard packages nog verder behandeld. Met behulp van de procedure PUT_LINE kan tekst op het scherm worden gezet. Deze boodschappen worden alleen getoond als de serverouput op het tabblad DBMS Output aangezet wordt. Dit tabblad kunt u oproepen middels de menu optie: View → Dbms Output.



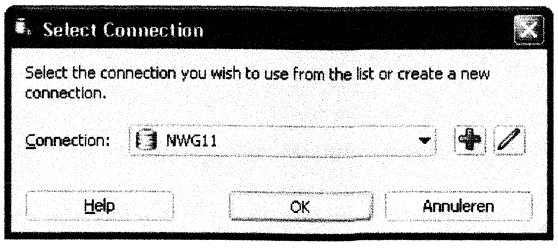
Het Dbms Output tabblad verschijnt vervolgens naast de Compiler Log:



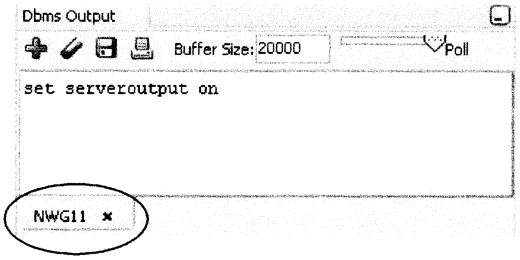
Op het DBMS Output tabblad kan per database connectie een sub-tabblad worden opgenomen waarin de serveroutput uit die connectie wordt getoond. Alle boodschappen die met behulp van de package DBMS_OUTPUT worden gegenereerd worden hier vervolgens getoond. Om een sub-tabblad voor onze reeds openstaande SQL Worksheet connectie toe te voegen moeten we op de Add new DBMS Output Tab ( ) button klikken.



Hierna verschijnt een window waarin we de connectie kunnen kiezen die we aan het sub-tabblad willen toevoegen:



Kies hier voor de database connectie waarvoor u de DBMS Output wilt activeren en klik vervolgens op de button OK. In het DBMS Output scherm verschijnt nu het statement `set serveroutput on` tevens ziet u onderin het sub-tabblad met de naam van de database connectie.



Als een aanroepende routine de exception kent die in een aangeroepen routine plaatsvindt, dan kan het afhandelen van de routine ook overgelaten worden aan de aanroepende routine. Dit gebeurt door in de afhandeling het statement RAISE te gebruiken zonder verdere toevoegingen. Hierdoor zal naar een zelfde exception in de aanroepende code worden gezocht. U dient er zelf voor te zorgen dat de aangeroepen routine deze exception niet kan aanroepen als de routine los wordt uitgevoerd of wanneer de routine wordt uitgevoerd vanuit een andere routine die de exception niet kent.

Voorbeeld:

```
create or replace package propagatie is
 procedure proc1;
 procedure proc2;
end;
/

create or replace package body propagatie is
 procedure proc1 is
   afhandeling1 exception;
  begin
   proc2;
  exception
   when afhandeling1 then
     dbms_output.put_line ('Dit is afhandeling1 in procedure proc1');
  end proc1;

 procedure proc2 is
  begin
   raise afhandeling1;
  EXCEPTION
     WHEN afhandeling1 THEN RAISE;
  end proc2;
end;
/
```

3.7 DESCRIBE

Als u snel wilt zien hoe u een procedure, functie of package moet gebruiken, kunt u dat (net zoals voor een tabel of view) doen met de functie DESC[RIBE]. Dit geld ook voor de standaard packages die in hoofdstuk 5 worden behandeld. Geeft u een procedure of functie op dan wordt voor die procedure of functie aangegeven wat de parameters zijn. Geeft u een package op, dan wordt voor elke procedure en functie die in de specificatie van die package staat een beschrijving gegeven. U kunt echter niet één procedure of functie uit een package zien met DESCRIBE.

3.8 Rechten op procedures, functies en packages.

In deze paragraaf zullen we bespreken hoe procedures, functies en packages beveiligd kunnen worden in de database.

3.8.1 Rechten om procedures, functies en packages uit te voeren

In het begin van dit hoofdstuk staat al bij de voordelen van packages genoemd dat je veel handiger rechten kunt verlenen op één package dan op een hele rits procedures en functies. Deze rechten worden uitgedeeld met het statement GRANT EXECUTE ON <object> TO <gebruiker >waarin het object een procedure, functie of package is. Om deze rechten weer te kunnen ontnemen is er ook een statement en wel REVOKE EXECUTE ON.<object> FROM <gebruiker>.

Voorbeeld:

```
revoke execute on medewerkers from scott;
```

Als u een procedure, al dan niet in een package, gebruikt om ervoor te zorgen dat bedrijfsregels worden afgedwongen, moet u er wel voor zorgen dat eindgebruikers niet rechtstreeks kunnen werken. Als u bijvoorbeeld een procedure wilt laten gebruiken om een update uit te voeren, moet u ervoor zorgen dat eindgebruikers dus geen update rechten op de tabel hebben. Eventuele al bestaande rechten moeten weer ontnomen (REVOKE) worden. De update rechten zijn ook niet nodig.

Opgeslagen procedures en functies kunnen dus worden uitgevoerd door een gebruiker (de Invoker) die alleen execute rechten heeft gekregen op de betreffende procedure of functie. Hierbij hoeft de gebruiker verder niet noodzakelijkerwijs rechten te hebben op de tabellen die in de procedure of functie worden benaderd. Bij het uitvoeren van een procedure of functie wordt namelijk gebruik gemaakt van de privileges van degene die ze heeft aangemaakt (de Definer). De AUTHID-optie maakt het vanaf Oracle8*i* mogelijk dat deze rechten van de Invoker alsnog worden gecontroleerd.

Als gebruiker Scott een procedure maakt met daarin een INSERT op de tabel EMP, dan zal de tabel van Scott worden gevuld, ongeacht wie deze procedure uitvoert. Deze werkwijze heeft twee nadelen. Ten eerste wordt, zodra een bepaalde gebruiker rechten heeft op de procedure van Scott, niet meer gecontroleerd of deze gebruiker rechten heeft op de objecten die vanuit de procedure worden benaderd. Ten tweede benadert één bepaalde procedure altijd tabellen in een vooraf te bepalen schema, ongeacht vanuit welk schema (onder welke gebruiker) de procedure wordt uitgevoerd.

Om de tabellen te benaderen van de gebruiker die de procedure uitvoert, kunnen we de procedure kopiëren naar het schema van de ander gebruiker, maar dit is in verband met de onderhoudbaarheid niet wenselijk.

Dit kan ook op een andere manier worden opgelost. Bij een procedure, functie of package wordt opgegeven of er gebruik moet worden gemaakt van de rechten van degene die de procedure of functie heeft aangemaakt (DEFINER, dit is de standaard instelling), of van de rechten van de uitvoerder (INVOKER). Willen we de rechten van de uitvoerder gebruiken dan geven we bij de procedure, functie of package AUTHID CURRENT_USER op. AUTHID DEFINER duidt aan dat we de rechten van degene die de procedure, functie of package heeft aangemaakt willen gebruiken.

Voorbeeld

We maken twee procedures: UITVOERDERS_RECHTEN en MAKERS_RECHTEN die de naam en functie van een bepaalde werknemer in de tabel NAAMFUNCTIE zetten.

```
create or replace procedure uitvoerders_rechten (p_persnr number)
authid current_user is
 cursor c_ophalen (b_persnr number) is
   select naam, functie
   from p_werknemers
   where persnr=b_persnr;
 v_naam varchar2(20);
 v_functie varchar2(20);
begin
 open c_ophalen(p_persnr);
 fetch c_ophalen into v_naam, v_functie;
 close c_ophalen;
 insert into naamfunctie values (v_naam, v_functie);
end;
/

create or replace procedure makers_rechten(p_persnr number)
authid definer is
 cursor ophalen (b_persnr number) is
    select naam, functie
    from p_werknemers
    where persnr=b_persnr;
 v_naam varchar2(20);
 v_functie varchar2(20);
begin
 open ophalen(p_persnr);
 fetch ophalen into v_naam, v_functie;
 close ophalen;
 insert into naamfunctie values (v_naam, v_functie);
end;
/
```

We geven nu Scott rechten om de beide procedures uit te voeren en om uit de tabel NAAMFUNCTIE te mogen selecteren, zodat Scott het resultaat van de procedures kan bekijken. Dit kan met de onderstaande statements:

```
grant execute on uitvoerders_rechten to scott;
grant execute on makers_rechten to scott;
grant select on naamfunctie to scott;
```

Bij de procedure UITVOERDERS_RECHTEN wordt in het schema van Scott gekeken naar de tabel P_WERKNEMERS. Die bestaat niet, hetgeen resulteert in een foutmelding.

```
execute <gebruikersnaam>.uitvoerders_rechten(3381)
```

```
Error starting at line 1 in command:
execute nwg806.uitvoerders_rechten(3381)
Error report:
ORA-00942: table or view does not exist
ORA-06512: at "NWG806.UITVOERDERS_RECHTEN", line 4
ORA-06512: at "NWG806.UITVOERDERS_RECHTEN", line 10
ORA-06512: at line 1
00942. 00000 -  "table or view does not exist"
*Cause:
*Action:
```

Wanneer de gebruiker SCOTT een rij probeert toe te voegen aan de tabel NAAMFUNCTIE in uw schema, resulteert dit ook in een foutmelding. SCOTT heeft immers geen INSERT rechten op uw tabel.

```
insert into <gebruikersnaam>.naamfunctie values('naam','functie')
                    *
insert into nwg806.naamfunctie values('naam','functie')
Error at Command Line:1 Column:19
Error report:
SQL Error: ORA-01031: insufficient privileges
01031. 00000 -  "insufficient privileges"
*Cause:    An attempt was made to change the current username or password
           without the appropriate privilege. This error also occurs if
           attempting to install a database without the necessary operating
           system privileges.
           When Trusted Oracle is configure in DBMS MAC, this error may occur
           if the user was granted the necessary privilege at a higher label
           than the current login.
*Action:   Ask the database administrator to perform the operation or grant
           the required privileges.
           For Trusted Oracle users getting this error although granted the
           the appropriate privilege at a higher label, ask the database
           administrator to regrant the privilege at the appropriate label.
```

De Cause en Action die bij deze foutmelding verschijnen zijn heel algemeen. Met betrekking tot privileges kan er van alles fout gaat in een statement. Onze foutsituatie wordt niet echt verklaard in de toelichting van Oracle.

SCOTT kan wel de procedure MAKERS_RECHTEN uitvoeren, omdat deze procedure gebruik maakt van de rechten en het schema van de maker (definer).

```
execute <gebruikersnaam>.makers_rechten(3381)
```

De tabel NAAMFUNCTIE zal hierna de volgende inhoud hebben:

```
select * from <gebruikersnaam>.naamfunctie;

NAAM                 FUNCTIE
-------------------- ----------
SMITS                KLERK
```

Onder uw eigen gebruikersnaam kunt u wel beide procedures uitvoeren. In het algemeen geldt: wanneer een procedure uit het eigen schema wordt uitgevoerd, maakt het niet uit of er AUTHID DEFINER of AUTHID CURRENT_USER is opgegeven. Deze zijn immers gelijk.

3.8.2 Rechten om procedures, functies en packages aan te maken

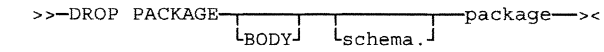
Het feit dat u procedures, functies en packages aan mag maken komt omdat u als cursist in het bezit bent van het CREATE PROCEDURE recht. Dit recht zorgt ervoor dat u procedures, functies en packages mag aanmaken, uitvoeren, veranderen en verwijderen in uw eigen schema. Er bestaat dus bijvoorbeeld geen DROP PROCEDURE recht.

Hiernaast bestaan nog de rechten CREATE ANY PROCEDURE, EXECUTE ANY PROCEDURE, ALTER ANY PROCEDURE en DROP ANY PROCEDURE. Deze rechten staan respectievelijk voor het aanmaken, uitvoeren, veranderen en verwijderen van procedures, functies en packages in elk willekeurig schema. Het hangt van database-instellingen af, die buiten het bestek van deze cursus vallen, maar op de meeste databases betekent "elk willekeurig schema" alle schema's uitgezonderd het schema van SYS.

3.9 Package verwijderen

Een package kan verwijderd worden met het DROP PACKAGE statement.

Syntax



Met de optie BODY kan alleen de body van een PACKAGE verwijderd worden. Zonder deze optie worden zowel de specificatie als de body allebei verwijderd.

4 NDS en standaard packages voor SQL

4.1 Inleiding

Binnen PL/SQL-programma's kunnen DDL-statements, zoals het statement CREATE TABLE, niet worden gebruikt. Daarom zijn door Oracle een aantal zogenaamde standaard packages gedefinieerd. Deze standaard packages bieden dezelfde functionaliteit als een aantal van deze DDL-statements.
Ook zijn er standaard nog een aantal andere packages gedefinieerd waarmee functionaliteit toegevoegd wordt aan PL/SQL, zoals het geven van uitvoer op het scherm.

4.2 Overzicht van standaard packages

Oracle stelt zeer veel packages ter beschikking. Hier volgt een overzicht van enkele interessante packages. Een volledige overzicht van de standaard packages staat in de PL/SQL Packages and Types Reference.

DBMS_ALERT	Deze package is bedoeld om applicaties te kunnen bouwen die zichzelf of elkaar een seintje kunnen geven.
DBMS_DDL	In deze package zitten procedures die onder meer dezelfde functionaliteit bieden als de SQL-statements ALTER, COMPILE en ANALYZE.
DBMS_JOB	In deze package zitten procedures waarmee programma's in een job gezet kunnen worden. Een job kan op een bepaald tijdstip worden uitgevoerd, of herhaaldelijk worden uitgevoerd. Bijvoorbeeld elke dag een export maken van de database, of elke uur gegevens van een andere database ophalen (replicatie). Deze package is verouderd en wordt vanaf Oracle10*g* vervangen door de DBMS_SCHEDULER.
DBMS_LOCK	Met deze package kunnen onder andere locks geplaatst, omgezet, verwijderd en benoemd worden. De package bevat de procedure SLEEP, waarmee een sessie een bepaalde tijd "wachtend" kan worden gezet.
DBMS_OUTPUT	Met behulp van de functies en procedures uit deze package kan informatie naar een buffer geschreven worden. De inhoud van de buffer kan ook weer uitgelezen worden.
DBMS_PIPE	Met deze package kunnen sessies die tot dezelfde database behoren met elkaar communiceren.
DBMS_RANDOM	Met deze package kunnen willekeurige getallen of karakters worden gegenereerd.
DBMS_SCHEDULER	Deze package bestaat vanaf Oracle10*g* en vervangt de DBMS_JOB package.
DBMS_SQL	Met behulp van deze package is het mogelijk om dynamische SQL te gebruiken in procedures, functies en PL/SQL-programma's. Hierdoor kunnen flexibeler procedures gemaakt worden omdat bijvoorbeeld de kolomnaam in een SELECT-statement nog niet bekend

	hoeft te zijn op het moment dat de procedure gecreëerd wordt.
DBMS_UTILITY	De procedures uit deze package zijn gelijk aan de procedures uit de package DBMS_DDL, met het verschil dat de DBMS_UTILITY procedures de betreffende statements uitvoeren voor **alle** objecten in plaats van voor een enkel object.
DBMS_WARNING	Beschikbaar vanaf Oracle10*g*. Deze package wordt behandeld in de cursus PL/SQL geavanceerd. Deze package maakt het mogelijk om tijdens compilatie ook bij ongebruikelijke en/of niet logische code een foutboodschap te krijgen.
UTL_COMPRESS	Beschikbaar vanaf Oracle10*g*. Deze package maakt het mogelijk om blob en raws in- (zip) en uit te pakken (unzip) binnen PL/SQL.
UTL_FILE	Met behulp van deze package kunnen gegevens vanuit PL/SQL worden weggeschreven in, of gelezen worden uit, een file op de server.
UTL_HTTP	Met behulp van deze package is het mogelijk om hypertext transfer protocol (http) te gebruiken in PL/SQL (en SQL). Hiermee kunnen gegevens vanuit internet benaderd worden via http.
UTL_MAIL	Beschikbaar vanaf Oracle10*g* als gedeeltelijke vervanger van de UTL_SMTP package. UTL_MAIL maakt het mogelijk om e-mails te verzenden vanuit PL/SQL. UTL_MAIL is veel eenvoudiger dan het gebruik van de UTL_SMTP package. De package ondersteunt echter alleen de meest algemene functionaliteiten van het bredere pakket dat UTL_SMTP biedt.
UTL_SMTP	Deze package maakt het mogelijk om e-mails te verzenden. Hiervoor is, i.t.t. de eenvoudigere package UTL_MAIL, kennis nodig van het SMTP protocol.
UTL_TCP	Met deze package kunnen PL/SQL applicaties via TCP/IP communiceren met externe op TCP/IP gebaseerde servers, zoals internet servers. De package is daardoor handig voor PL/SQL applicaties die gebruik maken van internet protocollen en e-mail.

In de volgende paragrafen zullen we meer aandacht besteden aan de packages DBMS_SQL en DBMS_UTILITY. Daarnaast wordt aandacht geschonken aan Native Dynamic SQL. (NDS). Deze packages en NDS hebben met elkaar gemeen dat ze het mogelijk maken om binnen PL/SQL allerhande SQL statements uit te voeren die niet rechtstreeks mogelijk zijn. Daarnaast kan vooral bij NDS een grote performancewinst worden geboekt t.o.v. gewoon SQL. Van de overige packages die vooral andersoortige functionaliteit toevoegen, worden de belangrijkste in de volgende twee hoofdstukken behandeld.

4 NDS en standaard packages voor SQL

4.3 De package DBMS_SQL

Met de package DBMS_SQL kan gebruik worden gemaakt van dynamische SQL statements binnen PL/SQL. Daarnaast worden DDL-statements zoals CREATE TABLE, dat anders niet gebruikt kan worden binnen PL/SQL, mogelijk met deze package.

Dynamische SQL is een SQL-statement dat pas bij het uitvoeren gecompileerd wordt. Hierdoor is het mogelijk om procedures te creëren die heel algemeen zijn, omdat in het EXECUTE statement verschillende variabelen aan de procedure kunnen worden meegegeven, zoals kolom- en tabelnamen. Het is dan mogelijk om met één procedure verschillende tabellen te bewerken of om programma's te maken voor objecten die nog niet bestaan. In het algemeen is het nadeel van het verplaatsen van het compileren naar het moment van uitvoeren dat het programma iets trager wordt. Compileren tijdens het runnen wordt naast dynamisch soms ook late binding genoemd. Het tegenovergestelde, gecompileerd opslaan zodat code snel uitvoerbaar is, heet static of early binding.
In het volgende voorbeeld wordt een gebruiksmogelijkheid van dynamische SQL verduidelijkt.

In een standaard PL/SQL-blok is het niet mogelijk een tabel aan te maken met behulp van het CREATE TABLE-statement. De reden hiervoor is dat dit een DDL-statement is. Probeer het volgende PL/SQL-blok uit te voeren:

```
begin
  create table h4_voorbeeld (kolom varchar2(1));
end;
/
```

Uit de foutmelding die nu verschijnt blijkt dat het woord CREATE gereserveerd is en niet gebruikt mag worden op deze manier.

```
Error starting at line 1 in command:
begin
  create table h4_voorbeeld (kolom varchar2(1));
end;
Error report:
ORA-06550: line 2, column 3:
PLS-00103: Encountered the symbol "CREATE" when expecting one of the following:

   ( begin case declare exit for goto if loop mod null pragma
   raise return select update while with <an identifier>
   <a double-quoted delimited-identifier> <a bind variable> <<
   continue close current delete fetch lock insert open rollback
   savepoint set sql execute commit forall merge pipe purge
06550. 00000 -  "line %s, column %s:\n%s"
*Cause:    Usually a PL/SQL compilation error.
*Action:
```

Om nu toch een tabel te kunnen maken in een PL/SQL-programma kan gebruik worden gemaakt van dynamische SQL. Wanneer het volgende PL/SQL-programma wordt uitgevoerd, zal de tabel wel worden aangemaakt.

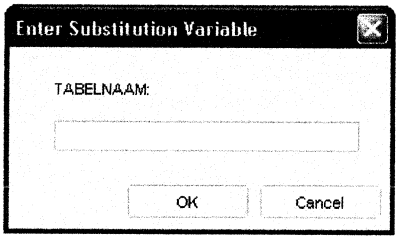
```
declare
  v_dyna_cur  integer;
  v_uitvoer integer;
begin
  v_dyna_cur := dbms_sql.open_cursor;
  dbms_sql.parse (v_dyna_cur,
                  'create table h4_voorbeeld (kolom varchar2(1))', 1);
  v_uitvoer:= dbms_sql.execute (v_dyna_cur);
  dbms_sql.close_cursor (v_dyna_cur);
end;
```

```
desc h4_voorbeeld
Name                 Null?    Type
-------------------- -------- ------------------------------
KOLOM                         VARCHAR2(1)

1 rows selected
```

Nu blijkt dat het wel mogelijk is een tabel te maken vanuit een PL/SQL-programma. Over de verschillende statements die in dit PL/SQL-programma worden gebruikt, volgt in een volgende paragraaf een nadere uitleg.

In een standaard PL/SQL-blok is het niet mogelijk om een cursor te schrijven die gebruik maakt van een variabele tabelnaam. Tijdens het parsen (compileren) van het SQL statement wordt namelijk onder andere gecontroleerd of de tabelnaam in de FROM-clausule bekend is in de database. De onderstaande code voeren we uit voeren in SQL Developer:

```
declare
  v_tabel varchar2(30):= '&tabelnaam';
  cursor ophalen_naam is
       select naam from v_tabel where rownum = 1;
begin
for rij in ophalen_naam loop
 dbms_output.put_line (rij.naam);
end loop;
end;
/
```



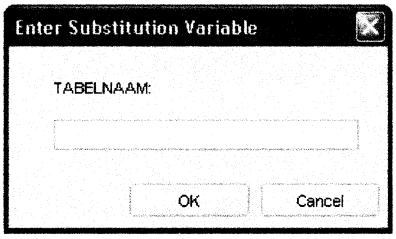
← We vullen hier als tabelnaam P_KANTOREN in.

Uit de foutmelding die verschijnt blijkt dat tijdens de compilatie van de cursor in de FROM-clausule nog steeds de variabele naam V_TABEL staat, vandaar de melding: `table or view does not exist`:

```
Error starting at line 1 in command:
declare
  v_tabel varchar2(30):= '&tabelnaam';
  cursor ophalen_naam is
       select naam from v_tabel where rownum = 1;
begin
for rij in ophalen_naam loop
 dbms_output.put_line (rij.naam);
end loop;
end;
Error report:
ORA-06550: line 4, column 25:
PL/SQL: ORA-00942: table or view does not exist
ORA-06550: line 4, column 8:
PL/SQL: SQL Statement ignored
ORA-06550: line 7, column 24:
PLS-00364: loop index variable 'RIJ' use is invalid
ORA-06550: line 7, column 2:
PL/SQL: Statement ignored
06550. 00000 -  "line %s, column %s:\n%s"
*Cause:    Usually a PL/SQL compilation error.
*Action:
```

Om nu toch dynamisch een tabelnaam op te kunnen geven, kan gebruik worden gemaakt van dynamische SQL. Wanneer het volgende PL/SQL-programma wordt uitgevoerd, zullen wel alle namen uit de tabel P_KANTOREN op het scherm afgedrukt worden:

```
declare
  p_tabel varchar2(30):= '&tabelnaam';
  v_naam  varchar2(15);
  v_dyna_cur   integer;
  v_uitkomst integer;
begin
  v_dyna_cur := dbms_sql.open_cursor;
  dbms_sql.parse (v_dyna_cur, 'select initcap(naam) from '||p_tabel, 1);
  dbms_sql.define_column (v_dyna_cur, 1, v_naam, 15);
  v_uitkomst := dbms_sql.execute (v_dyna_cur);
  while dbms_sql.fetch_rows (v_dyna_cur) = 1 loop
    dbms_sql.column_value (v_dyna_cur, 1, v_naam);
    dbms_output.put_line (v_naam);
  end loop;
  dbms_sql.close_cursor (v_dyna_cur);
exception
  when others then
    dbms_sql.close_cursor (v_dyna_cur);
    raise_application_error (-20001,sqlerrm);
end;
/
```



← We vullen hier als tabelnaam P_KANTOREN in.

```
anonymous block completed
Boekhouding
Onderzoek
Verkoop
Productie
```

Nu blijkt dat het wel mogelijk is om dynamisch een tabelnaam aan een PL/SQL-programma mee te geven, we kunnen dit programma voor meerdere tabellen starten zolang er maar een kolom NAAM in die tabel aanwezig is.

Over de verschillende statements die in dit PL/SQL-programma worden gebruikt, volgt in de volgende paragrafen een nadere uitleg.

4.3.1 De procedures en functies

In deze paragraaf zullen eerst een aantal procedures en functies besproken worden die in de package DBMS_SQL zijn opgenomen.

4.3.1.1 OPEN_CURSOR

Om een SQL statement uit te kunnen voeren moet een cursor geopend zijn. De functie OPEN_CURSOR opent een cursor en geeft als waarde een ID van het type INTEGER terug.

Syntax

```
cursor_id := dbms_sql.open_cursor;
```

4.3.1.2 PARSE

Elk statement moet worden geparst. Hierbij wordt de syntax gecontroleerd en wordt gekeken of de gebruiker de bevoegdheid heeft die nodig is om het betreffende statement uit te voeren, voordat het wordt uitgevoerd. Een dynamisch SQL statement wordt geparst door het aanroepen van de procedure PARSE.

Syntax

```
dbms_sql.parse(<cursor_id> , <statement>, <language_flag>)
```

cursor_id:	Het bij het openen van de cursor verkregen ID.
statement:	Het statement dat moet worden geparst.
language_flag:	Om aan te geven met welke database versie wordt gewerkt. 0 Om aan te geven dat het een versie 6 database is (V6). 1. Oracle neemt de database versie over waarmee gewerkt wordt (NATIVE). 2. Om aan te geven dat het een versie 7 database is (V7).

4.3.1.3 DEFINE_COLUMN

Deze procedure wordt alleen gebruikt, wanneer gebruik wordt gemaakt van een dynamische query. Alle kolommen die in de SELECT-clausule worden gebruikt, moeten worden gedefinieerd waarbij het volgnummer in de SELECT clausule wordt opgegeven.

Syntax

```
dbms_sql.define_column(<cursor_id>, <positie>, <kolom>[, <lengte>])
```

cursor_id:	Het bij het openen van de cursor verkregen ID.
positie:	De positie van de kolom in het SELECT statement. De positie wordt altijd aangegeven door een nummer.
kolom:	De naam van de te definiëren kolom. Deze naam moet als variabele worden gedeclareerd, het datatype moet hetzelfde zijn als het datatype van de kolom in de tabel.
lengte:	Bij kolommen van de datatypen CHAR, VARCHAR2 en RAW moet de lengte van de kolom worden opgegeven, bij kolommen van de datatypen NUMBER, DATE of ROWID moet de lengte worden weggelaten.

4.3.1.4 BIND_VARIABLE

Deze procedure koppelt een invoerwaarde aan een variabele uit het te parsen statement. De variabele in het statement moet worden aangegeven met een dubbele punt (:variabele). Door het gebruik van een bindvariabele hoeft het statement voor verschillende waarden van de bindvariabele slechts één keer te worden geparst. Bind-variabelen zijn bedoeld voor gebruik in de WHERE en HAVING clausule van het SELECT statement.

Syntax

```
dbms_sql.bind_variable(<cursor_id>, <name>, <value>)
```

cursor_id:	De ID van de cursor waarin het statement staat waaraan de variabele moet worden toegekend.
name:	de naam van de variabele die in het statement is opgenomen.
value:	de waarde van de variabele.

4.3.1.5 EXECUTE

De EXECUTE functie voert het SQL-statement uit en geeft als returnwaarde het aantal rijen dat is uitgevoerd. Dit aantal rijen is alleen van belang bij het uitvoeren van INSERT-, UPDATE- en DELETE-statements. Bij SELECT-statements is de returnwaarde ongedefinieerd.

Syntax

```
dbms_sql.execute(<cursor_id>)
```

cursor_id:	Het bij het openen van de cursor verkregen ID.

4.3.1.6 FETCH_ROWS

Door het aanroepen van de functie FETCH_ROWS wordt steeds één rij opgehaald die voldoet aan de query. De functie geeft als return waarde het getal 1 wanneer een rij kan worden opgehaald. Wanneer geen rij meer kan worden opgehaald wordt de returnwaarde gelijk aan 0.

Syntax

```
dbms_sql.fetch_rows(<cursor_id>)
```

cursor_id: Het bij het openen van de cursor verkregen ID.

4.3.1.7 EXECUTE_AND_FETCH

Deze functie combineert de EXECUTE en FETCH_ROWS functies. Wanneer deze functie wordt gebruikt in plaats van de EXECUTE functie, dan wordt direct de eerste rij opgehaald die voldoet aan de query. De overige rijen kunnen daarna met de functie FETCH_ROWS worden opgehaald.

Syntax

```
dbms_sql.execute_and_fetch(<cursor_id>[, <exact>])
```

cursor_id:	Het bij het openen van de cursor verkregen ID.
exact:	Dit is een BOOLEAN die standaard de waarde FALSE heeft. Dit betekent dat de fetch één rij ophaalt, ongeacht het aantal rijen dat aan de query voldoet. Geven we deze parameter de waarde TRUE, dan zal een exception aangeheven worden wanneer meer dan één rij of geen rij voldoet aan de query, vergelijkbaar met SELECT ... INTO.

4.3.1.8 COLUMN_VALUE

Deze procedure bepaalt de waarde van een kolom die is opgehaald na het aanroepen van de functie FETCH_ROWS. Deze waarde wordt in een variabele gezet. Voorwaarde hiervoor is dat er een kolom gedefinieerd moet zijn met DEFINE_COLUMN.

Syntax

```
dbms_sql.column_value(<cursor_id>, <positie>, <kolom>)
```

cursor_id:	Het bij het openen van de cursor verkregen ID.
positie:	De positie van de kolom in het SELECT statement. De positie wordt altijd aangegeven door een nummer.
kolom:	De naam van de kolom, deze naam moet gedeclareerd worden. Om het overzicht beter te kunnen bewaren gebruiken we dezelfde naam als bij DEFINE_COLUMN.

4.3.1.9 IS_OPEN

Met deze functie kan gekeken worden of een bepaalde cursor geopend is.

Syntax

```
dbms_sql.is_open(<cursor_id>)
```

cursor_id:	Het bij het openen van de cursor verkregen ID. De functie retourneert TRUE als de cursor open is, FALSE als de cursor gesloten is.

4.3.1.10 CLOSE_CURSOR

Wanneer deze procedure wordt aangeroepen, dan wordt de betreffende cursor gesloten. Hierna kan hier niet meer aan worden gerefereerd.

Syntax

```
dbms_sql.close_cursor(<cursor_id>)
```

cursor_id:	Het bij het openen van de cursor verkregen ID.

Voorbeeld

In het voorbeeld gaan we achtereenvolgens de volgende stappen uitvoeren:

1. De cursor dyna_cur wordt geopend en er wordt een stukje geheugen gereserveerd voor het uitvoeren van de statements.
2. Het SELECT statement wordt gecontroleerd (*geparst*), bijvoorbeeld of de syntax klopt en of de gebruiker de juiste rechten heeft om het betreffende statement uit te voeren. Binnen dit statement maken we gebruik van een zogenaamde bindvariabele :T_NAAM. Een bindvariabele wordt altijd voorafgegaan door een dubbele punt (:). De waarde van deze variabele wordt pas bij het uitvoeren van de procedure toegekend. Het SELECT statement wordt dan in 2 stappen geparst, eerst wordt gekeken of het statement klopt en daarna wordt de waarde aan de bindvariabele toegekend.
3. Met DEFINE_COLUMN wordt aangeduid dat we een kolom uit het SELECT-statement willen gaan gebruiken in de procedure.
4. Met EXECUTE wordt het SELECT-statement uitgevoerd en de rijen in het geheugen gezet.

5. Daarna worden deze rijen door FETCH_ROWS opgehaald uit het geheugen, totdat er geen rijen meer zijn. Als er geen rijen meer kunnen worden opgehaald geeft de functie FETCH_ROWS de waarde 0 terug.
6. De opgehaalde waarden worden met COLUMN_VALUE in een variabele gezet, die gebruikt wordt om de waarden met DBMS_OUTPUT op het scherm te zetten.
7. Als alle rijen zijn opgehaald wordt de cursor gesloten en het geheugen vrijgegeven.

```
create or replace procedure dynamisch_sql(p_naam varchar2) is
  v_dyna_cur integer;
  v_regelnr number;
  v_tekst varchar2(1000);
  v_uitkomst integer;
begin
  v_dyna_cur:=dbms_sql.open_cursor;                                    (stap 1)
  dbms_sql.parse (v_dyna_cur,                                          (stap 2)
             'select line, text
               from user_source
               where name=upper(:t_naam)', 1);
   dbms_sql.define_column(v_dyna_cur, 1, v_regelnr);                   (stap 3)
   dbms_sql.define_column(v_dyna_cur, 2, v_tekst, 1000);               (stap 3)
   dbms_sql.bind_variable(v_dyna_cur, 't_naam', p_naam);
   v_uitkomst:=dbms_sql.execute(v_dyna_cur);                           (stap 4)
   while dbms_sql.fetch_rows(v_dyna_cur)=1 loop                        (stap 5)
     dbms_sql.column_value(v_dyna_cur, 1, v_regelnr);                  (stap 6)
     dbms_sql.column_value(v_dyna_cur, 2, v_tekst);                    (stap 6)
     dbms_output.put_line(v_regelnr||' '||v_tekst);
   end loop;
   dbms_sql.close_cursor(v_dyna_cur);                                  (stap 7)
 end dynamisch_sql;
 /
```

Voorbeeld van de output die de functie DYNAMISCH_SQL genereerd:

```
execute dynamisch_sql('SPECIFICATIE')

1 package specificatie is
2   max_aantal_cursisten constant number:=2;
3   max_aantal_dagen constant p_cursussen.aant_dag%type:=99;
4   eerste_boeking constant date:=add_months(sysdate,12);
5   laatste_boeking constant date:=sysdate+1;
6   e_verkeerd_datatype exception;
7   pragma exception_init(e_verkeerd_datatype, -1722);
8 end;
```

4.3.2 DBMS_SQL gebruiken

In procedures en functie met dynamische SQL staat vaak een when others exception met ongeveer de volgende code:

Voorbeeld

```
create or replace procedure x is
...
begin
...
exception
  when others then
   dbms_sql.close_cursor (v_dyna_cur);
   raise_application_error (-20001,sqlerrm);
end;
/
```

Hierdoor wordt in geval van fouten altijd netjes de cursor gesloten en er wordt geen ROLLBACK uitgevoerd. Als er een foutmelding ontstaat en de serveroutput staat niet aan, dan komt er geen foutmelding op het scherm. Ook wanneer deze procedure vanuit een

andere tool aangeroepen, bijv. Forms, wordt aangeroepen komt er geen foutmelding op het scherm.

Dynamische SQL wordt veelgebruikt om DDL-statements binnen een PL/SQL-programma uit te voeren. Met de volgende functie worden NOT NULL constraints aan gemaakt.

```
create or replace function notnull_constraints
  ( f_tabelnaam in varchar2
  , f_kolomnaam in varchar2)
  return varchar2
is
  v_dyna_cur integer;
  v_uitkomst integer;
begin
  v_dyna_cur:=dbms_sql.open_cursor;
  dbms_sql.parse(v_dyna_cur, 'alter table '||f_tabelnaam||' modify '||f_kolomnaam||'
                 not null', 1);
  v_uitkomst:=dbms_sql.execute(v_dyna_cur);
  dbms_sql.close_cursor(v_dyna_cur);
  return 'Gelukt';
exception
  when others then
    dbms_sql.close_cursor(v_dyna_cur);
    raise_application_error (-20001,sqlerrm);
end;
/
```

Bij het aanroepen van de functie moeten we de tabelnaam en kolomnaam opgeven.

```
variable hulpvar varchar2(20)
exec :hulpvar:=notnull_constraints ('p_werknemers', 'naam')

print hulpvar

HULPVAR
------------------
Gelukt
```

De kolom NAAM van de tabel P_WERKNEMERS bevat nu een NOT NULL constraint.

```
desc p_werknemers
```

Na afloop ziet de tabel P_WERKNEMERS er als volgt uit:

```
desc p_werknemers
Name                            Null?    Type
------------------------------- -------- ------------
PERSNR                          NOT NULL NUMBER(6)
NAAM                            NOT NULL VARCHAR2(10)
FUNCTIE                                  VARCHAR2(9)
MGR                                      NUMBER(4)
SAL                                      NUMBER(5)
TOESLAG                                  NUMBER(5)
KANTNR                                   NUMBER(2)
```

Als de functie wordt uitgevoerd wordt het volgende DDL-statement uitgevoerd.

```
alter table p_werknemers modify naam not null
```

Let op dat variabele kolomnamen en tabelnamen in de string geconcateneerd (via ||) moeten worden, terwijl voor variabele velden/waarden altijd bindvariabelen (:variabele) gebruikt moeten worden.

4.3.3 Het gebruik van bindvariabelen

We kunnen met een bindvariabele verschillende waarden doorgeven aan een statement.

Voorbeeld

Met de procedure MEERDERE_WAARDEN gaan we de functie en het salaris van 3 werknemers ophalen uit de tabel P_WERKNEMERS. De namen van de werknemers geven we met 3 parameters door.

```
create or replace procedure meerdere_waarden ( p_naam1 varchar2, p_naam2 varchar2
                                              , p_naam3 varchar2)
is
  v_dyn_cur integer;
  v_uitkomst integer;
  v_functie p_werknemers.functie%TYPE;
  v_sal p_werknemers.sal%TYPE;
begin
 v_dyn_cur:= dbms_sql.open_cursor;
 dbms_sql.parse(v_dyn_cur, 'select functie, sal from p_werknemers
                           where upper(naam)=upper(:t_naam)',1);
 dbms_sql.define_column(v_dyn_cur, 1, v_functie, 20);
 dbms_sql.define_column(v_dyn_cur, 2, v_sal);
 dbms_sql.bind_variable(v_dyn_cur, 't_naam', p_naam1);          --p_naam1 doorgeven
 v_uitkomst:=dbms_sql.execute_and_fetch(v_dyn_cur);
  if v_uitkomst = 1 then
     dbms_sql.column_value(v_dyn_cur, 1, v_functie);
     dbms_sql.column_value(v_dyn_cur, 2, v_sal);
     dbms_output.put_line (p_naam1||' ' ||v_functie||' '||v_sal);
  end if;
 dbms_sql.bind_variable(v_dyn_cur, 't_naam', p_naam2);          --p_naam2 doorgeven
 v_uitkomst:=dbms_sql.execute_and_fetch(v_dyn_cur);
  if v_uitkomst = 1 then
     dbms_sql.column_value(v_dyn_cur, 1, v_functie);
     dbms_sql.column_value(v_dyn_cur, 2, v_sal);
     dbms_output.put_line (p_naam2||' '||v_functie||' '||v_sal);
  end if;
 dbms_sql.bind_variable(v_dyn_cur, 't_naam', p_naam3);          --p_naam3 doorgeven
 v_uitkomst:=dbms_sql.execute_and_fetch(v_dyn_cur);
  if v_uitkomst = 1 then
     dbms_sql.column_value(v_dyn_cur, 1, v_functie);
     dbms_sql.column_value(v_dyn_cur, 2, v_sal);
     dbms_output.put_line (p_naam3||' '||v_functie||' '||v_sal);
  end if;
 dbms_sql.close_cursor(v_dyn_cur);
end;
```

We testen de procedure vervolgens met de werknemers Smits, Alkema en Walstra

```
execute meerdere_waarden('smits', 'alkema', 'walstra')
```

```
smits KLERK 2400
alkema VERKOPER 2600
walstra VERKOPER 2250
```

In deze code is het statement dat in de parse staat maar een keer geparsed maar wel drie keer uitgevoerd. Door dit hergebruik van cursoren, kan dynamische SQL ondanks het runtime compileren toch sneller zijn dan gewoon SQL of PL/SQL.

Vanaf Oracle11*g* zijn de procedures en functies uit de DBMS_SQL package uitgebreid met parameters voor alle mogelijke data types binnen Oracle, zoals bijvoorbeeld collecties en bulk items. Daarnaast heeft men nu ook de beschikking over de functies TO_REFCURSOR TO_CURSOR_NUMBER, waarmee REF CURSOR naar DBMS_SQL cursoren kunnen worden getransformeerd en vice versa. Op collecties en REF cursoren gaan wij dieper in, in de cursus Oracle Database 11g: PL/SQL geavanceerd.

4.4 De package DBMS_UTILITY

De package DBMS_UTILITY bevat uiteenlopende procedures en functies. Zo bevat de package procedures en functies die te maken hebben met het genereren van statistische informatie over tabellen en indexen.
Verder bevat de package procedures die van PL/SQL- tabellen een lijst met waarden kunnen maken en omgekeerd. Een PL/SQL-tabel is een collectietype en wordt besproken in de cursus PL/SQL geavanceerd.
Daarnaast bevat de package DBMS_UTILITY, zoals we in hoofdstuk 2 hebben gezien, de procedure COMPILE_SCHEMA om op een eenvoudige wijze alle procedures, functies en packages van een gebruiker te kunnen compileren.

4.4.1 De procedures

De package is erg uitgebreid en veel van de procedures vallen buiten het gebied dat we binnen de cursus behandelen. We zullen er hier slechts twee bespreken: COMPILE_SCHEMA en EXEC_DDL_STATEMENT. Een beschrijving van de package vindt u in de PL/SQL Packages and Types Reference.

4.4.1.1 COMPILE_SCHEMA

Deze procedure compileert alle procedures, functies en packages van het opgegeven schema.

Syntax

```
dbms_utility.compile_schema(<schema>)
```

schema	De naam van het schema (de gebruikersnaam) waarvan de procedures en dergelijke moeten worden gecompileerd.

4.4.1.2 EXEC_DDL_STATEMENT

Deze procedure voert het opgegeven DDL-statement uit.

Syntax

```
dbms_utility.exec_ddl_statement(<statement>)
```

statement	Het statement dat moet worden uitgevoerd.

4.4.2 Het gebruik van EXEC_DDL_STATEMENT

De procedure EXEC_DDL_STATEMENT, maakt het programmeren met dynamische SQL een stuk eenvoudiger.
Het uitvoeren van DML statements met behulp van deze procedure is echter niet mogelijk. Voor het uitvoeren van dynamische SELECT, INSERT en UPDATE statements moet de package DBMS_SQL worden gebruikt.

Voorbeeld

Met de procedure TABEL_KOPIEREN kunnen we een kopie maken van de kolom NAAM uit een opgegeven tabel. De tabelnaam wordt bij het uitvoeren van de procedure opgegeven.

```
create or replace procedure tabel_kopieren(p_tabelnaam varchar2) is
  v_dyna_cur integer;
  v_uitkomst integer;
begin
  dbms_utility.exec_ddl_statement('create table kopie (naam varchar2(20))');
  v_dyna_cur:=dbms_sql.open_cursor;
  dbms_sql.parse(v_dyna_cur, 'insert into kopie select naam from '||p_tabelnaam, 1);
  v_uitkomst:=dbms_sql.execute(v_dyna_cur);
  dbms_sql.close_cursor(v_dyna_cur);
end;
/
```

We testen de procedure met als parameter de tabel P_KANTOREN:

```
exec tabel_kopieren('p_kantoren')
```

De tabel KOPIE ziet er na afloop als volgt uit:

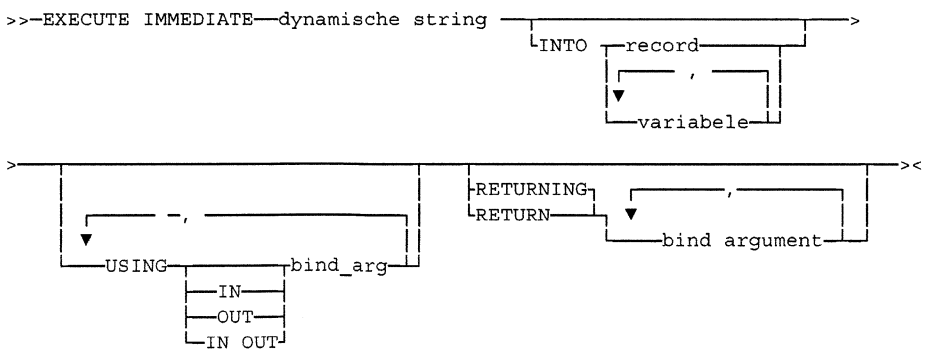
```
select * from kopie;

NAAM
--------------------
BOEKHOUDING
ONDERZOEK
VERKOOP
PRODUCTIE

desc kopie
Name                           Null     Type
------------------------------ -------- ----------------------------------------
NAAM                                    VARCHAR2(20)
```

Met Native Dynamic SQL kan het nog korter. Native Dynamic SQL wordt in de volgende paragraaf besproken.

4.5 Native Dynamic SQL

In Oracle bestaat naast de package DBMS_SQL nog een andere methode om dynamische SQL te gebruiken, namelijk Native Dynamic SQL. Met Native Dynamic SQL kunnen we DML-, DDL-statements en PL/SQL-blokken uitvoeren.

Syntax



Met EXECUTE IMMEDIATE wordt het statement doorgeven en uitgevoerd Het lijkt erg op EXEC_DDL_STATEMENT uit de package DBMS_UTILITY, alleen met Native Dynamic SQL kunnen ook DML-statements worden uitgevoerd.

Voorbeeld

```
create or replace procedure nds_dml ( p_nr number, p_naam varchar2
                                    , p_plaats varchar2)
is
  v_dml_string varchar2(200);
  r_kantoren p_kantoren%rowtype;
begin
  v_dml_string:='insert into p_kantoren values (:knr, :knaam, :kplaats)';
  execute immediate v_dml_string using p_nr, p_naam, p_plaats;
  v_dml_string:='select * from p_kantoren where kantnr=:knr';
  execute immediate v_dml_string into r_kantoren using p_nr;
  dbms_output.put_line('Kantoor: '||r_kantoren.naam||' te '||r_kantoren.plaats);
end;
```

Met de procedure NDS_DML wordt een rij toegevoegd aan P_KANTOREN. In de variabele V_DML_STRING staat het statement dat moet worden uitgevoerd.
Bij EXECUTE IMMEDIATE wordt het statement uit DML_STRING uitgevoerd met de waarden die opgegeven zijn bij USING.

Bij het SELECT statement worden waarden opgehaald die bij de EXECUTE IMMEDIATE met INTO in een variabele moeten worden gezet. Deze variabele kan een record zijn, maar ook uit verschillende losse variabelen bestaan. INTO kan alleen gebruikt worden bij een SELECT statement en het statement mag slechts één rij opleveren.

Als we naar het syntaxdiagram kijken van EXECUTE IMMEDIATE, zien we dat USING IN hetzelfde doet als USING, omdat IN de default is. Zoals we in bovenstaand voorbeeld hebben kunnen zien, wordt USING [IN] gebruikt om de bindvariabelen in de string te vullen met waarden.

Wanneer de string geen bindvariabelen bevat, maar bijvoorbeeld geconcateneerde variabele kolom- of tabelnamen, hoeft geen USING [IN] te worden gebruikt.
USING OUT kan worden gebruikt om waarden die terugkomen op te vangen.
Specifiek voor waarden die terugkomen van een INSERT-, UPDATE- of DELETE-statement, kan de RETURN[ING] clausule van EXECUTE IMMEDIATE gebruikt worden. In Oracle maakt het niet uit of u hiervoor USING OUT of RETURN[ING] gebruikt. Oracle raadt ten sterkste aan om alleen RETURN[ING] te gebruiken.

Voorbeelden:

```
declare
 v_num number;
begin
 execute immediate 'begin :1 := plusnumber(2,3); end;' using out v_num;
 dbms_output.put_line( 'De uitkomst is '||to_char(v_num));
end;
/
anonymous block completed
De uitkomst is 5

declare
 v_salterug number;
 v_stmt varchar2(500);
begin
  v_stmt:= 'update pl_werknemers set sal = sal+2000 where naam = ''SMITS'''||
        ' returning sal into :1';
 execute immediate v_stmt returning into v_salterug;
 dbms_output.put_line(v_salterug);
end;
/
anonymous block completed
4400
```

In bovenstaand voorbeeld ziet u dat het opnemen van 'SMITS' in de string niet zo eenvoudig is. Om een apostrof (') te gebruiken in een string, moeten we twee quotes achter elkaar gebruiken. (Let op dat dit niet hetzelfde is als dubbele aanhalingstekens!) Het schrijven van twee quotes kan echter lastig zijn.

In Oracle10*g* is het mogelijk om zelf een karakter te kiezen dat het begin en einde van een string aangeeft. De notatie q'{.....}' maakt het mogelijk om enkele quotes in de string te gebruiken.

Voorbeeld

```
declare
 v_salterug number;
 v_stmt varchar2(500);
begin
v_stmt := q'{update pl_werknemers set sal = sal +2000 where naam = 'SMITS'}'
          ||' returning sal into :1';
 execute immediate v_stmt returning into v_salterug;
 dbms_output.put_line(v_salterug);
end;
/
```

Hier is gekozen voor de accolade, maar er zou bijvoorbeeld ook voor het uitroepteken gekozen kunnen worden (q'!.....!'). De enige voorwaarde is dat het gekozen karakter niet voorkomt in de string zelf.

4.6 Verschillen tussen Native Dynamic SQL en DBMS_SQL

Om een juiste keuze te kunnen maken tussen het gebruik van Native Dynamic SQL (NDS) of DBMS_SQL, moeten de voordelen van beide methoden tegen elkaar worden afgewogen.

De voordelen bij het gebruik van Native Dynamic SQL zijn:

- De code van Native Dynamic SQL is korter dan de code met DBMS_SQL. Hierdoor is de code makkelijker te schrijven en te onderhouden.
- Betere performance doordat de statementvoorbereiding (parse), de toekenning van waarde aan de variabelen (binding) en de uitvoering (execute) in één keer worden uitgevoerd.

Er is echter ook een voordeel bij het gebruik van DBMS_SQL ten opzichte van Native Dynamic SQL (NDS):

- De procedure DESCRIBE_COLUMNS kan gebruikt worden om de kolomdefinitie te omschrijven van kolommen die geopend en geparst zijn.

5 Scheduling

5.1 Inleiding

Wanneer we PL/SQL code niet direct willen uitvoeren, maar op een opgegeven tijdstip en eventueel herhaaldelijk, kunnen we ze plannen. Dit plannen wordt in Oracle termen "*job scheduling*" genoemd.

Oorspronkelijk is scheduling in Oracle ontwikkeld voor replicatie. Met replicatie kunnen door snapshots (momentopname) gegevens van de ene database naar de ander database worden overgehaald. Op die manier kunnen gebruikers met gegevens van een andere database werken zonder dat deze database zelf benaderd wordt. Dit levert vaak een grote performancewinst op. Scheduling kan echter ook worden gebruikt om omvangrijke procedures, zoals het maken van een export, uit te voeren op momenten dat er weinig andere gebruikers zijn.

De package die gebruikt moet worden voor scheduling is DBMS_SCHEDULER. In voorgaande Oracle versies (Oracle9*i* en ouder) werd voor scheduling gebruik gemaakt van de package is DBMS_JOB. DBMS_JOB is nog wel te gebruiken, maar de nieuwe package DBMS_SCHEDULER biedt uitgebreidere mogelijkheden.

5.2 DBMS_JOB

Voor de volledigheid en voor personen die te maken krijgen met Oracle9*i* database behandelen we hier kort de mogelijkheden van de package DBMS_JOB.

5.2.1 Procedures

De belangrijkste onderdelen van de package DBMS_JOB worden in de volgende subparagrafen besproken.

5.2.1.1 SUBMIT

Met deze procedure wordt een nieuwe job in een wachtrij (=job queue) gezet.

Syntax

```
dbms_job.submit( <job>, <what> [, <next_date>][, <interval>][, <no_parse>])
```

job	Het nummer van de job, deze wordt bepaald door de sequence SYS.JOBSEQ en dit nummer wordt teruggegeven als OUT parameter, van het type NUMBER.
what	De PL/SQL code die moet worden uitgevoerd. De tekst moet worden omgeven door enkele quotes (').
next_date	De datum waarop de job moet worden uitgevoerd. DEFAULT staat deze op SYSDATE.
interval	Een datumexpressie van het type VARCHAR2. Met deze expressie wordt aangeven wanneer de job opnieuw moet worden uitgevoerd. DEFAULT staat deze op 'null', dit betekent dat de job slechts één keer wordt uitgevoerd.

no_parse	Een Boolean die aangeeft of de PL/SQL code geparst moet worden bij het plaatsen in de job queue (FALSE) of bij het uitvoeren van de job (TRUE). FALSE is de default waarde.

Bij het aanmaken van een job met SUBMIT wordt automatisch de sequence SYS.JOBSEQ aangeroepen en een nieuw jobnummer gegenereerd. Het jobnummer wordt op het scherm getoond, dit kunt u bij de volgende voorbeelden gebruiken.

5.2.1.2 CHANGE

Door het gebruik van deze procedure kunnen de job parameters worden gewijzigd, zoals NEXT_DATE en WHAT.

Syntax

```
dbms_job.change( <job>, <what>, <next_date>, <interval>)
```

job	Het nummer van de job, die moet worden aangepast.
what	De PL/SQL code die moet worden uitgevoerd. De tekst moet worden omgeven door enkele quotes (').
next_date	De datum waarop de job moet worden uitgevoerd.
interval	Een datumexpressie van het type VARCHAR2. Met deze expressie wordt aangegeven wanneer de job opnieuw moet worden uitgevoerd.

5.2.1.3 WHAT, NEXT_DATE en INTERVAL

Voor het wijzigen van slechts één parameter kan beter één van de procedures WHAT, NEXT_DATE of INTERVAL worden gebruikt.

Syntax

```
dbms_job.what( <job>, <what>)

dbms_job.next_date( <job>, <next_date>)

dbms_job.interval( <job>, <interval>)
```

5.2.1.4 RUN

Deze procedure voert de opgegeven job meteen uit.

Syntax

```
dbms_job.run( <job>)
```

job:	Het nummer van de job, die moet worden uitgevoerd.

5.2.1.5 REMOVE

Deze procedure verwijdert de aangegeven job uit de wachtrij.

Syntax

```
dbms_job.remove(<job>)
```

job:	Het nummer van de job, die moet worden aangepast.

5.2.2 Het gebruik van DBMS_JOB

De package DBMS_JOB wordt binnen een Oracle database veelgebruikt voor replicatie. Oracle maakt dan jobs aan om op vaste tijden de gegevens uit een andere database te verversen. Ook binnen een applicatie kan het handig zijn om een bepaalde actie regelmatig uit te voeren. Door het opgeven van het interval kunnen we precies bepalen wanneer een bepaalde job wordt uitgevoerd.

Voorbeeld

Bij een cursusinstituut moet elke maandagochtend, om 9 uur, een overzicht worden gemaakt van het aantal cursisten van die week. Hiervoor is de procedure AANTAL_CURSISTEN aangemaakt in de database.

```
create or replace procedure aantal_cursisten(f_datum in date) is
  cursor c_aantal_cur is
    select count(*)
    from p_boekingen
    where to_char(datum, 'ww yyyy')=to_char(f_datum, 'ww yyyy');
  v_aantal number;
begin
  open c_aantal_cur;
  fetch c_aantal_cur into v_aantal;
  close c_aantal_cur;
  insert into hulptabel
    values (null, to_char(f_datum, 'ww yyyy'), v_aantal);
end;
/
```

Met het onderstaande script maken we een job aan in de database die direct wordt uitgevoerd en daarna elke maandag om 9 uur opnieuw wordt gestart.

```
declare
  v_jobnr number;
  v_code varchar2(200);
begin
  v_code:='begin
         aantal_cursisten(sysdate);
         end;';
  dbms_job.submit(v_jobnr,v_code,sysdate, 'next_day(trunc(sysdate), ''MONDAY'')
+9/24');
  dbms_output.put_line('Jobnr: '||to_char(v_jobnr));
end;
/
```

Let op bij MONDAY staan 2 **enkele** quotes, om er voor te zorgen dat MONDAY bij het uitvoeren van het statement tussen quotes staat.

Pas als er een commit wordt gegeven, wordt de job daadwerkelijk uitgevoerd. Een alternatief voor commit is om zelf dbms_job.run aan te roepen.

Voorbeeld:

```
exec dbms_job.run(<jobnr>);
```

5.2.3 Datadictionary views

In de datadictionary view USER_JOBS kan worden bekeken welke jobs er zijn aangemaakt en wanneer deze worden uitgevoerd.

```
SELECT job
,      what
,      last_date
,      last_sec
,      next_date
,      next_sec
FROM user_jobs;
```

```
      JOB WHAT                                 LAST_DATE LAST_SEC NEXT_DATE NEXT_SEC
--------- ------------------------------------ --------- -------- --------- --------
      124  begin                               18-FEB-08 16:36:02 18-FEB-08 16:37:02
           aantal_cursisten(sysdate);
           end;
```

5.3 DBMS_SCHEDULER (Oracle10g)

De package DBMS_SCHEDULER is nieuw in Oracle10g en vervangt de DBMS_JOB package. Ook deze package is ontwikkeld om bepaalde programma's op een bepaald tijdstip of vaker uit te voeren. De te plannen programma's kunnen PL/SQL-blokken of opgeslagen procedures zijn. Tevens is het met de DBMS_SCHEDULER package mogelijk om shell scripts of executables uit te voeren die buiten de database zijn opgeslagen, mits deze vanaf de command-line van het O.S. uitvoerbaar zijn.

De DBMS_SCHEDULER package heeft meer mogelijkheden dan de DBMS_JOB package. Zo is het mogelijk om de te plannen actie apart op te slaan als "program". Zaken als de begindatum, het tijdsinterval en de eindtijd kunnen apart worden opgeslagen in een "schedule". Het voordeel hiervan is dat deze programs en schedules voor verschillende jobs gebruikt kunnen worden.

Verder verschaft de DBMS_SCHEDULER package een aantal geavanceerde mogelijkheden. Zo is het bijvoorbeeld mogelijk om jobklassen aan te maken, waarin soortgelijke jobs gegroepeerd kunnen worden. Per jobklasse kan vervolgens de prioriteit ingesteld worden, zodat jobs altijd in een bepaalde volgorde worden uitgevoerd. Windows maken het juist mogelijk om gedurende een tijdsperiode (bijvoorbeeld 's ochtends, 's nachts, in het weekend) prioriteiten voor verschillende jobs in te stellen.

Aangezien de mogelijkheden van de DBMS_SCHEDULER package erg uitgebreid zijn, zullen we in deze cursus alleen op de basisprocedures en functies ingaan.

5.3.1 Procedures

De belangrijkste onderdelen van de package DBMS_SCHEDULER worden in de volgende subparagrafen besproken.

5.3.1.1 CREATE_SCHEDULE

Met deze procedure wordt een *schedule* aangemaakt. Een schedule is een plan waarin aangegeven wordt hoe vaak een job uitgevoerd moet worden en of dit meerdere keren moet gebeuren. Met de procedure CREATE_JOB kunnen er later gemakkelijk verschillende programma's aan deze schedule gekoppeld worden.

Syntax

```
dbms_scheduler.create_schedule (schedule_name, start_date, repeat_interval,
                                end_date, comments)
```

schedule_name	Unieke naam voor de schedule.
start_date	De datum waarop de job moet worden gepland. Default wordt de job direct gepland.
repeat_interval	Met deze expressie wordt aangeven wanneer de job opnieuw moet worden uitgevoerd. Default wordt de job maar één keer uitgevoerd. Bijvoorbeeld kan de job jaarlijks ('FREQ=YEARLY') of ieder uur ('FREQ=HOURLY') worden uitgevoerd. Een volledig overzicht van de mogelijkheden vindt u in de appendix Appendix C.
end_date	Met deze expressie wordt aangeven wanneer de job verlopen is en niet meer mag worden uitgevoerd. Default heeft de job geen einddatum.
comments	Commentaar.

5.3.1.2 CREATE_PROGRAM

Hiermee kan een programma gedefinieerd worden. Het programma kan een PL/SQL-blok, een opgeslagen procedure of een executable zijn. Het programma kan later met de procedure CREATE_JOB eenvoudig aan een schedule gekoppeld worden.

Syntax

```
dbms_scheduler.create_program(program_name, program_type, program_action,
                              number_of_arguments, enabled, comments)
```

program_name	Unieke naam voor het programma.
program_type	Het programmatype kan PLSQL_BLOCK, STORED_PROCEDURE of EXECUTABLE zijn.
program_action	Dit is afhankelijk van het program_type. Het kan een PL/SQL blok bevatten, de naam van een opgeslagen procedure of de naam van de executable inclusief padnaam.
arguments	Het aantal argumenten dat het programma heeft. Een programma kan maximaal 255 argumenten hebben. Wanneer het programmatype PL/SQL-blok is, kunnen er geen argumenten bestaan en wordt dit genegeerd.
enabled	Een boolean die default de waarde FALSE heeft. Wanneer de waarde op TRUE wordt gezet, wordt het programma gevalideerd en wordt het programma ENABLED.
comments	Commentaar.

5.3.1.3 CREATE_JOB

Met deze procedure wordt een nieuwe job aangemaakt. Bij deze procedure kan overloading plaatsvinden. Het is namelijk mogelijk om een job aan te maken op basis van alleen een schedule, alleen een programma, beide of geen van beide. We zullen hier alleen de procedure toelichten die een job aanmaakt op basis van zowel een schedule (CREATE SCHEDULE) als een programma (CREATE_PROGRAM). Voor de andere mogelijkheden verwijzen we naar de PL/SQL Packages and Types Reference.

Syntax

```
dbms_scheduler.create_job(job_name, program_name, schedule_name, job_class,
                          enabled, auto_drop, comments)
```

job_name	Unieke naam voor de job.
program_name	Naam van het externe programma dat met de procedure CREATE_PROGRAM is aangemaakt.
schedule_name	Naam van de externe schedule die met de procedure CREATE_SCHEDULE is aangemaakt.
job_class	De naam van de jobklasse. Het is mogelijk om naast de default jobklasse zelf met DBMS_SCHEDULER.CREATE_JOB_CLASS jobklassen aan te maken in het schema van SYS. Het definiëren van jobklassen helpt mee het overzicht te behouden over jobs.
enabled	Om een aangemaakte job direct in de wachtrij (=job queue) te zetten moet deze ENABLED worden. Default wordt een job echter DISABLED (FALSE) aangemaakt. De status van een eenmaal aangemaakte job kan gewijzigd worden met de procedures ENABLE en DISABLE uit de DBMS_SCHEDULER package.
auto_drop	Een boolean die een uitgevoerde job, waarvoor geen interval is ingesteld, automatisch verwijdert (TRUE) of niet verwijdert (FALSE). De default is TRUE.
comments	Commentaar.

5.3.1.4 SET_ATTRIBUTE

Deze procedure maakt het mogelijk om wijzigingen aan te brengen in een gedefinieerde schedule. De wijzigingen hebben alleen betrekking op toekomstige runs van de job.

Syntax

```
dbms_scheduler.set_attribute(name, attribute, value)
```

name	Naam van de te wijzigen schedule.
attribute	Attribuutnaam.
value	Nieuwe waarde voor het attribuut.

5.3.1.5 ENABLE

Met deze procedure kan de status van onder meer een job of een programma gewijzigd worden van DISABLED in ENABLED. Om ervoor te zorgen dat een job daadwerkelijk wordt uitgevoerd, moet de job coordinator de job uit de wachtrij halen. Hiervoor is het noodzakelijk dat een job de status ENABLED heeft.

Syntax

```
dbms_scheduler.enable(name)
```

name	Naam van de job of het programma.

Via de datadictionary view USER_SCHEDULER_JOBS is te achterhalen wanneer een job wordt uitgevoerd.

```
SELECT next_run_date
FROM user_scheduler_jobs;

NEXT_RUN_DATE
-------------
18-FEB-09 04.52.56.000000 PM EUROPE/BERLIN

1 rows selected
```

Let erop dat een job niet altijd direct uitgevoerd wordt. Dit heeft te maken het jaarlijkse herhalingsinterval van de schedule.Wanneer er geen herhalingsinterval voor de job zou gelden, dan zal een job met de status ENABLED wel direct (eenmalig) uitgevoerd worden. Met de procedure DISABLE, die een soortgelijke eenvoudige syntax heeft, kan de status juist gewijzigd worden in DISABLED.

5.3.1.6 RUN_JOB

Deze procedure voert de opgegeven job meteen uit. Voorwaarde is wel dat de job ENABLED is.

Syntax

```
dbms_scheduler.run_job(job_name)
```

job_name	Naam van de te wijzigen schedule.

5.3.1.7 Het verwijderen van jobs, schedules en programma's

Om jobs, schedules en programma's te verwijderen kunnen we gebruik maken van de DROP_JOB, DROP_PROGRAM en DROP_SCHEDULE procedures. De procedures hebben een soortgelijke syntax.

Syntax

```
dbms_scheduler.drop_job(job_name [, force])
dbms_scheduler.drop_program(program_name [, force])
dbms_scheduler.drop_schedule(schedule_name [, force])
```

name	Afhankelijk van de gebruikte procedure is dit een jobnaam, programmanaam of schedulenaam.
force	Heeft default de waarde FALSE. Wanneer force de waarde TRUE krijgt krijgen alle afhankelijke objecten de status DISABLED.

5.3.2 Het gebruik van DBMS_SCHEDULER

Het gebruik van de DBMS_SCHEDULER package is vergelijkbaar met dat van de DBMS_JOB package. In de vorige paragraaf hebben we een job aangemaakt om de procedure aantal_cursisten iedere maandag om 9 uur uit te kunnen voeren. We gaan een soortgelijke job nu maken met behulp van de DBMS_SCHEDULER package.

Voorbeeld

```
begin
  dbms_scheduler.create_program
   (program_name        => 'program_aantal_cursisten',
    program_type        => 'plsql_block',
    program_action      => 'begin aantal_cursisten(sysdate); end;',
    enabled             =>  true);
  dbms_scheduler.create_schedule
    (schedule_name      => 'schedule_aantal_cursisten',
     start_date         =>  sysdate,
     repeat_interval    => 'freq=weekly;byday=mon;byhour=9');
  dbms_scheduler.create_job
   (job_name            => 'job_aantal_cursisten',
    program_name        => 'program_aantal_cursisten',
    schedule_name       => 'schedule_aantal_cursisten',
    enabled             =>  true);
end;
/
```

Alhoewel de start_date als default waarde SYSDATE heeft, wordt de job niet direct uitgevoerd. Dat komt omdat er een repeat_interval is opgegeven, die ervoor zorgt dat de job iedere maandagmorgen om 9 uur uitgevoerd wordt. In de datadictionary view USER_SCHEDULER_JOBS is te achterhalen op welke datum de job de volgende keer uitgevoerd wordt.

```
SELECT next_run_date
FROM user_scheduler_jobs;

NEXT_RUN_DATE
-------------
25-FEB-08 09.55.15.900000 AM +01:00
```

We zien dat de job pas op de eerstvolgende maandag om ongeveer 9 uur uitgevoerd gaat worden. Als we de job echter meteen willen uitvoeren, kunnen we dat doen via de procedure RUN_JOB.

```
exec dbms_scheduler.run_job('job_aantal_cursisten')
```

Wanneer we het interval van een job willen wijzigen dan kan dit met de procedure SET_ATTRIBUTE, de interval in het onderstaande voorbeeld wordt gewijzigd naar 1 minuut:

```
begin
 dbms_scheduler.set_attribute
 (name=>'schedule_aantal_cursisten',
  attribute=>'repeat_interval',
  value=>'freq=minutely');
end;
/
```

5.3.3 Datadictionary views

In de datadictionary view USER_SCHEDULER_JOBS kan worden bekeken welke jobs er zijn aangemaakt en wanneer deze worden uitgevoerd.

```
select job_name
,      program_name
,      schedule_name
,      enabled
,      auto_drop
from user_scheduler_jobs;

JOB_NAME             PROGRAM_NAME             SCHEDULE_NAME             ENABL AUTO_
-------------------- ------------------------ ------------------------- ----- -----
JOB_AANTAL_CURSISTEN PROGRAM_AANTAL_CURSISTEN SCHEDULE_AANTAL_CURSISTEN TRUE  TRUE
```

Andere belangrijke datadictionary views zijn USER_SCHEDULER_SCHEDULES en USER_SCHEDULER_PROGRAMS. Hiermee kunnen we informatie opvragen over de schedules en programma's die we aangemaakt hebben.

Het verwijderen van de job gaat als volgt:

```
execute dbms_scheduler.drop_job('job_aantal_cursisten')
```

6 Veelzijdige standaard packages

6.1 Inleiding

In dit hoofdstuk worden enkele standaard packages behandeld die een uiteenlopende reeks mogelijkheden toevoegen aan de standaard functionaliteit van PL/SQL. De besproken packages zijn DBMS_OUTPUT, UTL_FILE, UTL_MAIL, DBMS_RANDOM, DBMS_ALERT en DBMS_PIPE.

6.2 De package DBMS_OUTPUT

Met behulp van deze package kan informatie in een buffer worden gezet die getoond kan worden wanneer een trigger, procedure of package wordt uitgevoerd, zoals we al in sommige voorbeelden hebben gezien. De package is vooral nuttig om debug informatie te tonen. Door het gebruik van SET SERVEROUTPUT ON in SQL Developer kan deze informatie daadwerkelijk op het scherm verschijnen.

6.2.1 De procedures

De meest gebruikte procedures uit de package DBMS_OUTPUT zullen hier worden besproken, een volledig overzicht vindt u in de PL/SQL Packages and Types Reference.

6.2.1.1 PUT_LINE

Deze procedure specificeert het stuk informatie dat de gebruiker als een regel in de buffer wil plaatsen. De informatie wordt afgesloten door een eindregelteken. De volgende informatie die hierna in de buffer geplaatst wordt begint op de volgende regel. Wanneer de buffer vol is wordt een foutmelding gegeven.

Syntax

```
dbms_output.put_line(<informatie>)
```

informatie:	de informatie die in de buffer moet worden geplaatst. Het datatype van deze informatie is VARCHAR2.

6.2.1.2 NEW_LINE

Deze procedure plaatst een eindregelmarkering in de buffer. Hierdoor verschijnt de inhoud van de buffer op het scherm.

Syntax

```
dbms_output.new_line
```

6.2.1.3 PUT

Deze procedure specificeert het stuk informatie dat de gebruiker in de buffer wil plaatsen. Deze informatie wordt niet afgesloten met een einde-regel teken waardoor de volgende informatie die in de buffer wordt geplaatst direct achter de voorgaande informatie wordt geplaatst op dezelfde regel. Wanneer de buffer vol zit wordt een foutmelding gegeven.

Syntax

```
dbms_output.put(<informatie>)
```

informatie: de informatie die in de buffer moet worden geplaatst. Het datatype van deze informatie mag VARCHAR2, NUMBER of DATE zijn.

6.2.2 Het gebruik van DBMS_OUTPUT

De package DBMS_OUTPUT wordt vaak gebruikt bij het debuggen van in de database opgeslagen procedures en triggers. In het PL/SQL-programma kunnen op bepaalde plaatsen stukken informatie in de buffer gezet worden (of op het scherm getoond) die van belang zijn voor de uitvoering van het blok, zodat makkelijk te achterhalen is waar het programma fout gaat.

Naast het gebruik voor debugging kan de package DBMS_OUTPUT gebruikt worden om bepaalde informatie op te slaan of te laten zien. De package kan gebruikt worden om een bepaalde boodschap voor de gebruiker op het scherm te plaatsen tijdens de uitvoering van een procedure.

Wanneer met behulp van PUT informatie in de buffer wordt geplaatst, dan is deze pas te gebruiken na het afsluiten van deze informatie met behulp van NEW_LINE. Deze procedure voegt een einde-regel teken toe aan het einde van de huidige regel in de buffer waarna de regel op het scherm worden getoond.

Voorbeeld:

```
declare
  cursor c_ophalen_kantnr is
    select kantnr
    from p_kantoren;
begin
  for r_ophalen_kantnr in c_ophalen_kantnr loop
    dbms_output.put(r_ophalen_kantnr.kantnr);
    dbms_output.put(',');
  end loop;
  dbms_output.new_line;
end;
/
10,20,30,40,
```

Wanneer DBMS_OUTPUT wordt gebruikt in functies binnen een package, dan kan deze functie niet remote, dus via een databaselink, worden gebruikt . De procedures uit de package DBMS_OUTPUT lezen en schrijven namelijk in een package.

6.3 De package UTL_FILE

Met deze package kunnen vanuit PL/SQL programmagegevens in een operating-system-file worden weggeschreven. Verder is het mogelijk om gegevens uit een file op te halen en bijvoorbeeld weg te schrijven in een tabel.
Om deze package te kunnen gebruiken moet er in de database worden aangegeven welke directories geschreven of gelezen kunnen worden. Dit gebeurt met de initialisatie parameter UTL_FILE_DIR. Vanaf Oracle10g wordt aangeraden om het CREATE DIRECTORY statement te gebruiken. Alleen files die door de server benaderd kunnen worden, kunnen door deze package worden gebruikt.

6.3.1 De procedures en functies

Deze package is in Oracle10*g* uitgebreid. Daardoor is het nu mogelijk geworden om binaire data te lezen en weg te schrijven naar een file. De belangrijkste procedures en functies uit de package UTL_FILE worden in de volgende subparagrafen besproken.

6.3.1.1 FOPEN

Met deze functie wordt een bepaalde file geopend. De functie geeft een UTL_FILE.FILE_TYPE terug.

Syntax

```
utl_file.fopen(<locatie>, <filenaam>, <openmode>)
```

locatie	De directory waarin de file staat.
filenaam	De naam van de file die geopend moet worden.
openmode	In welke mode de file geopend moet worden.
	r of rb voor lezen (Read) in normale of byte mode.
	w *of wb* voor schrijven in normale of byte mode, waarbij de bestaande tekst wordt geschreven (Write).
	a of ab voor schrijven in normale of byte mode, waarbij achter de bestaande tekst wordt geschreven (Append).

6.3.1.2 GET_LINE

Met deze procedure wordt een regel uit de file gelezen. De openmode moet dan R zijn.

Syntax

```
utl_file.get_line(<file>, <buffer>)
```

file	De file, van het type UTL_FILE.FILE_TYPE, waaruit de rijen moet worden opgehaald.
buffer	Dit is een OUT parameter van het type VARCHAR2. Hierin wordt de tekst geplaatst die wordt opgehaald.

6.3.1.3 PUT_LINE

Met deze procedure wordt een regel tekst in de file geplaatst. De openmode moet dan W of A zijn.

Syntax

```
utl_file.put_line(<file>, <tekst>)
```

file	De file, van het type UTL_FILE.FILE_TYPE, waaruit de rijen moet worden opgeslagen.
tekst	De tekst die in de file moet worden geplaatst.

6.3.1.4 PUTF

Met deze procedure kan de tekst in de file worden geplaatst. De tekst wordt nu niet als één stuk tekst gelezen, maar geïnterpreteerd als verschillende onderdelen. Het aantal verschillende onderdelen is maximaal 5. Door het opgeven van een bepaald formaatmasker kan voor een bepaalde lay-out worden gezorgd.

Syntax

```
utl_file.putf(<file>, <formaat>,<arg1> [,<arg2>][,<arg3>][,<arg4>][,<arg5>])
```

file	Het bestand, van het type UTL_FILE.FILE_TYPE, waarin de rijen worden opgeslagen.
formaat	Het masker waarin de tekst moet worden gezet. Dit masker kan bestaan uit letterlijke tekst en de volgende patronen:
%s	Om aan te geven dat PUTF het corresponderende stuk tekst in de file moet zetten.
\n	Geeft aan dat er op een nieuwe regel moet worden begonnen.
argN	De tekst die in het bepaalde masker moet worden ingelezen. De DEFAULT waarde is NULL.

6.3.1.5 FCLOSE

Met deze procedure wordt het geopende bestand gesloten.

Syntax

```
utl_file.fclose(file)
```

file	De file, van het type UTL_FILE.FILE_TYPE, die gesloten moet worden. Het is een IN OUT parameter, als het bestand gesloten is wordt de waarde NULL geretourneerd.

6.3.1.6 FREMOVE

Met deze procedure kan een file uit het O.S. verwijderd worden. Deze procedure kan pas vanaf Oracle10g gebruikt worden.

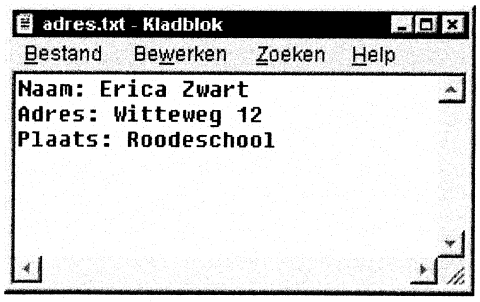
Syntax

```
utl_file.fremove(<lokatie>, <filenaam>)
```

lokatie	De directory waarin de file staat.
filenaam	De naam van de te verwijderen file.

Voorbeeld

```
declare
  v_file_id utl_file.file_type;
begin
  v_file_id:=utl_FILE.FOPEN('DIR_GEGEVENS', 'adres.txt', 'a');
  utl_file.putf(v_file_id,'Naam: %s\nAdres: %s\nPlaats: %s',
                'Erica Zwart','Witteweg 12','Roodeschool');
  utl_file.fclose(v_file_id);
  utl_file.fremove('c:', 'adres.txt');
end;
```



6.4 De package UTL_MAIL

Deze package is beschikbaar vanaf Oracle10g en maakt het mogelijk om vrij gemakkelijk een e-mail te verzenden. De package bevat drie procedures: SEND, SEND_ATTACH_RAW en SEND_ATTACH_VARCHAR2. Via de procedure SEND kan een eenvoudige e-mail verstuurd worden. De andere twee procedures overloaden deze procedure, waardoor attachments aan de e-mail gekoppeld kunnen worden. Een volledig overzicht van de packages kunt u terugvinden in de PL/SQL Packages and Types Reference.

In eerdere versies dan Oracle10g konden e-mails verstuurd worden door gebruik te maken van de packages SMTP_MAIL en UTL_TCP. Het schrijven van de PL/SQL code was echter een stuk bewerkelijker.

Voorbeeld

We zullen hier achtereenvolgens laten zien hoe u een e-mail kunt versturen met behulp van de UTL_SMTP en UTL_MAIL packages. Beide PL/SQL-programma's versturen een e-mail van verzender@domein.nl naar ontvanger@domein.nl. Het onderwerp van de e-mail is "E-mail verzenden via PL/SQL". U kunt deze voorbeelden NIET zelf uitvoeren, maar krijgt op deze manier wel een idee van het gebruik van deze packages.

Allereerst de code met behulp van de package UTL_SMTP:

```
declare
 v_mailhost varchar2(30):='<hostname>.<domeinnaam>.com';
 v_mail_connectie utl_smtp.connection;
begin
 v_mail_ connectie:=utl_smtp.open_connection(v_mailhost,25);
 utl_smtp.helo(v_mail_connectie, v_mailhost);
 utl_smtp.mail(v_mail_connectie, 'verzender@domein.nl');
 utl_smtp.rcpt(v_mail_connectie, 'ontvanger@domein.nl');
 utl_smtp.open_data(v_mail_connectie);
 utl_smtp.write_data(v_mail_connectie,
                     'Subject: E-mail verzenden via PL/SQL'||utl_tcp.crlf
                     ||'Content-type:text/html;'||utl_tcp.crlf||utl_tcp.crlf);
 utl_smtp.write_data(v_mail_connectie,'Hier kunt u een bericht schrijven.');
 utl_smtp.close_data(v_mail_connectie);
 utl_smtp.quit(v_mail_connectie);
end;
```

We gaan nu kijken hoe we dit veel eenvoudiger kunnen schrijven met behulp van de package UTL_MAIL:

```
begin
 utl_mail.send (sender => 'verzender@domein.nl',
                recipients => 'ontvanger@domein.nl',
                subject => 'E-mail verzenden via PL/SQL',
                message => 'Hier kunt u een bericht schrijven.');
end;
```

We zien dat in de code de lokatie van de mailhost niet meer terug is te vinden. Default wordt het databasedomein als mailhost lokatie overgenomen. Wanneer we een ander domein willen instellen, kan dit in de initialisatieparameter SMTP_OUT_SERVER.

6.5 DBMS_RANDOM

Met de package DBMS_RANDOM kunnen willekeurige waarden worden gegenereerd.

6.5.1 De procedures en functies

6.5.1.1 VALUE

Met deze procedure kan opgegeven worden tussen welke waarden het randomgetal moet liggen. De onderste waarde is inclusief, de bovenste waarde is exclusief, anders gezegd, de uitkomst kan gelijk zijn aan de onderste waarde maar is altijd kleiner dan de bovenste waarde.

```
begin
  dbms_output.put_line (trunc(dbms_random.value(10000000,99999999)));
end;
/
76879202
```

Wanneer geen waarden worden opgegeven gebruikt Oracle 0 en 1 als default waarden. Er bestaat naast VALUE ook een functie RANDOM die eveneens een willekeurig getal oplevert. Echter vanaf Oracle9*i* wordt aanbevolen om in plaats van RANDOM de functie VALUE te gebruiken. RANDOM is als obsolete aangemerkt.

6.5.1.2 STRING

Met deze procedure kan een string gegenereerd worden, waarbij de lengte opgegeven kan worden.

```
begin
  dbms_output.put_line(dbms_random.string('U',8));  -- U staat voor Upper
end;
/
ZPABQRSC
```

6.6 De package DBMS_ALERT

Met behulp van de procedures en functies uit deze package kan een applicatie bijvoorbeeld signaleren dat gegevens in bepaalde tabellen gewijzigd zijn door een andere gebruiker.

Wanneer door een bepaalde applicatie een alert is aangemaakt, dan kan deze alert zonder tussenkomst van een gebruiker door een andere applicatie worden ontvangen waardoor een bepaalde actie ondernomen kan worden. Een voorbeeld hiervan is het bijhouden op een scherm van de actuele stand van zaken (bijvoorbeeld van de voorraad in het magazijn). Dit overzicht wordt steeds bijgewerkt wanneer in de database gegevens wijzigen met betrekking tot de voorraadgegevens.

6.6.1 De procedures

De procedures die in de package DBMS_ALERT zitten worden in de volgende subparagrafen besproken.

6.6.1.1 REGISTER

Met behulp van deze procedure kan geregistreerd worden dat de huidige sessie moet reageren wanneer een opgegeven alert wordt aangemaakt.

Syntax

```
REGISTER(<naam>);
```

naam: de naam van de alert waarop de sessie moet reageren.

Voorbeeld

```
DBMS_ALERT.REGISTER('TELEFOON');
```

Hiermee wordt geregistreerd dat de gebruiker geïnteresseerd is in alle boodschappen met de naam 'TELEFOON'.

6.6.1.2 REMOVE

Met deze procedure kunnen de namen van de alerts die met REGISTER zijn opgeslagen weer verwijderd worden. Hierdoor zal de sessie niet meer reageren op het aanmaken van een alert. Wanneer een sessie wordt afgesloten (of afgebroken) zonder dat met behulp van REMOVE wordt aangegeven dat de sessie geen belangstelling meer heeft voor een bepaalde alert, dan zal steeds worden geprobeerd de niet meer aanwezige sessie te waarschuwen dat er een alert is opgetreden.
Dit komt omdat de registratie van de belangstelling van een sessie voor een alert los staat van het wel of niet bestaan van een sessie op het moment dat een alert optreedt.

Syntax

```
REMOVE(<naam>)
```

Naam: de naam van de alert waarop de sessie niet meer moet reageren.

Voorbeeld

```
DBMS_ALERT.REMOVE('TELEFOON');
```

6.6.1.3 REMOVEALL

Met deze procedure worden alle alerts die met REGISTER zijn opgeslagen weer verwijderd. Hierdoor zal de sessie niet meer reageren op welke alert dan ook.

Syntax

```
REMOVEALL
```

6.6.1.4 SET_DEFAULTS

Soms wordt er binnen een sessie geregeld gecontroleerd of er nog signalen uitgezonden zijn waarin de gebruiker geïnteresseerd is, met behulp van deze procedure kan het tijdsinterval tussen twee peilingen bepaald worden (in seconden). Standaard is 5 seconden.

Syntax

```
SET_DEFAULTS(<interval>);
```

Interval:	de interval tussen 2 peilingen in seconden.

6.6.1.5 SIGNAL

Zendt een bepaald signaal uit naar alle geïnteresseerden.

Syntax

```
SIGNAL(<naam>, <boodschap>);
```

naam:	de naam van de alert die het signaal moet geven. Alle sessies die geregistreerd staan voor deze alert zullen de boodschap ontvangen.
boodschap:	de boodschap die moet worden verzonden.

Voorbeeld

```
DBMS_ALERT.SIGNAL('TELEFOON', 'Wil dhr. Foppele dhr. Graaff terugbellen.');
```

Er wordt een boodschap verzonden met de naam 'TELEFOON'.

6.6.1.6 WAITANY

Wacht op een willekeurig signaal waarin de gebruiker geïnteresseerd is.

Syntax

```
WAITANY(<naam>, <boodschap>, <status>, <timeout>);
/* de parameters naam, boodschap en status zijn OUT parameters */
```

naam:	de naam van de alert die een boodschap heeft verstuurd.
boodschap:	de boodschap die door de betreffende alert is verzonden.
status:	de status van de WAITANY procedure. Wanneer als status de waarde 0 wordt gegeven, dan is er een alert opgetreden binnen de bij <timeout> opgegeven periode. Wanneer dit niet het geval is wordt de waarde 1 geretourneerd wat betekent dat de time-out periode is verstreken zonder een alert.
timeout:	de periode in seconden waarin maximaal moet worden gewacht op het optreden van een alert.

6.6.1.7 WAITONE

Wacht op een specifiek signaal.

Syntax

```
WAITONE(<naam>, <boodschap>, <status>, <timeout>)
/* de parameters boodschap en status zijn OUT parameters */
```

naam:	de naam van de alert waarop moet worden gewacht.
boodschap:	de boodschap die door de opgegeven alert is verzonden wanneer deze is opgetreden binnen de bij <timeout> opgegeven periode.
status:	de status van de WAITONE procedure. Wanneer als status de waarde 0 wordt gegeven, dan is de opgegeven alert opgetreden binnen de bij <timeout> opgegeven periode. Wanneer dit niet het geval is wordt de waarde 1 geretourneerd wat betekent dat de time-out periode is verstreken zonder een alert.
timeout:	de periode in seconden waarin maximaal moet worden gewacht op het optreden van de alert.

Voorbeeld

```
WAITONE('TELEFOON',boodschap,status,30)
```

Kijkt alleen of er nog alerts zijn met de naam 'TELEFOON' (zonder hiervoor de database te benaderen). WAITANY daarentegen kijkt naar elke andere willekeurige melding.

6.6.2 Het gebruik van DBMS_ALERT

Een alert wordt pas verstuurd wanneer de transactie die de alert veroorzaakt in de database wordt vastgelegd. Dit betekent dat wanneer een rollback wordt uitgevoerd na het aanroepen van de procedure SIGNAL, er geen alert wordt gegeven omdat de transactie niet wordt vastgelegd (na het aanmaken van de alert wordt geen commit gegeven).
De boodschap in een alert (datatype is VARCHAR2 en de omvang is maximaal 1800 bytes) kan worden gebruikt om te voorkomen dat een wachtende sessie de database gaat benaderen voor bepaalde gegevens. Deze gegevens kunnen via de alert naar één of meerdere sessies worden verstuurd. Daardoor kan één sessie de gegevens uit de database ophalen voor andere sessies.

Wanneer een alert meerdere keren is opgetreden voordat de procedures WAITANY of WAITONE zijn aangeroepen wordt de meest recente alert gebruikt. De eerdere alerts worden niet opgevangen, maar genegeerd.

Nadat de procedure SIGNAL is aangeroepen en een alert is verstuurd (na het geven van een commit) worden sessies die in deze alert zijn geïnteresseerd gewaarschuwd. Wanneer een geïnteresseerde sessie op dat moment in de wachtstand (WAITANY of WAITONE) staat wordt deze weer actief gemaakt.

Wanneer de sessies niet actief zijn krijgen ze de eerstvolgende keer dat ze gestart worden een waarschuwing.

Voorbeeld

```
create trigger t_alert_voorbeeld after update on pl_werknemers
begin
 dbms_alert.signal('werknemers_alert','de tabel pl_werknemers is aangepast !');
end;
/
```

Deze trigger gaat af wanneer de tabel PL_WERKNEMERS wordt aangepast (UPDATE) en zorgt er voor dat een alert met de naam 'werknemers_alert' en de inhoud 'De tabel PL_WERKNEMERS is aangepast !' wordt aangemaakt. (Triggers worden in het volgende hoofdstuk behandeld.) Sessies die belangstelling hebben voor deze alert moeten dit kenbaar maken door zich te registeren als belangstellende:

```
execute dbms_alert.register('werknemers_alert')
```

Wanneer nu hierna de tabel PL_WERKNEMERS wordt aangepast en deze aanpassing wordt bevestigd met een commit, dan zullen de geregistreerde sessies hiervan op de hoogte worden gebracht.

```
update pl_werknemers                          --we wijzigen niets maar de trigger gaat af
set sal = sal+0;
```

Nu is de alert nog niet aangemaakt en moet de aanpassing eerst worden bevestigd met commit:

```
commit;
```

Nu is de alert 'werknemers_alert' aangemaakt en is de huidige sessie hiervan op de hoogte gesteld omdat deze geregistreerd is als belangstellende voor deze alert.
Om dit te controleren kan het volgende PL/SQL blok worden uitgevoerd:

```
DECLARE
  v_boodschap varchar2(50);
  v_dummy integer;
begin
  dbms_alert.waitone('werknemers_alert', v_boodschap, v_dummy, 10);
  dbms_output.put_line ('Test: '||v_dummy||'; '||v_boodschap);
end;
/

Test: 0; De tabel PL_WERKNEMERS is aangepast !
```

Er wordt gewacht op een specifieke alert met de naam 'werknemers_alert'. Wanneer deze aanwezig is, is de return waarde van de procedure WAITONE gelijk aan 0 en wordt de boodschap in de alert getoond. Tevens wordt de return waarde getoond.

In de aanroep van de procedure WAITONE is een wachtperiode opgegeven van 10 seconden. Wanneer er nu geen alert is zal na 10 seconden het wachten worden afgebroken en is de return waarde gelijk aan 1.

Dit is te zien wanneer het bovenstaande PL/SQL blok nog een keer wordt uitgevoerd zonder dat de tabel PL_WERKNEMERS weer is aangepast. Dan zal het volgende op het scherm verschijnen:

```
Test: 1;
```

6.7 De package DBMS_PIPE

Wanneer een applicatie geen alerts nodig heeft die gebaseerd zijn op transacties, kan de package DBMS_PIPE een goed alternatief zijn voor de DBMS_ALERT package. Deze package maakt het mogelijk dat sessies, binnen één database instance, met elkaar kunnen communiceren.

Door een sessie wordt een boodschap eerst in een lokale buffer geplaatst. Wanneer de boodschap compleet is, wordt de inhoud van de lokale buffer in een zogenaamde PIPE geplaatst (een verbinding die beschouwd kan worden als een algemene buffer die is opgedeeld in stukjes met een bepaalde naam). Een andere sessie binnen dezelfde instance kan de boodschap weer vanuit de pipe in zijn lokale buffer plaatsen. Vervolgens kan de boodschap worden uitgepakt en kunnen de gegevens eventueel gebruikt worden voor verdere verwerking. Een overzicht van de actieve pipes is terug te vinden in de datadictionary view V$DB_PIPES.

6.7.1 De procedures en functies

De package DBMS_PIPE bevat een aantal procedures en functie. Via describe DBMS_PIPE kunnen deze opgevraagd worden. We zullen de belangrijkste procedures en functies hier nader toelichten.

6.7.1.1 CREATE_PIPE

Deze functie maakt expliciet een verbinding aan, publiek of privé. Deze blijft bestaan totdat hij expliciet verwijderd wordt met REMOVE_PIPE. Als de functie 0 retourneert, is de verbinding succesvol aangemaakt.

Syntax

```
CREATE_PIPE(<pipenaam>, <grootte>, <private>)
```

pipenaam:	naam van de verbinding, deze naam is uniek: er kan dus geen publieke en private verbinding zijn met dezelfde naam
grootte:	de maximale grootte in bytes van de verbinding. De standaard grootte is 8192 bytes.
private:	boolean die aangeeft of de verbinding privé (TRUE) of publiek is (FALSE). Default is TRUE.

Voorbeeld

```
DBMS_PIPE.CREATE_PIPE('publieke_verbinding',8192,FALSE);
```

6.7.1.2 PACK_MESSAGE

Deze procedure zet een boodschap in de lokale boodschap buffer.

Syntax

```
PACK_MESSAGE(<informatie>)
```

informatie:	de boodschap die in de buffer moet worden geplaatst. Het datatype van de boodschap kan VARCHAR2, NUMBER of DATE zijn.

Voorbeeld

```
DBMS_PIPE.PACK_MESSAGE(boodschap);
```

6.7.1.3 SEND_MESSAGE

Deze functie zendt een boodschap op de opgegeven verbinding. Deze verbinding wordt impliciet public aangemaakt als hij niet bestaat. Hierbij wordt de inhoud van de lokale boodschap buffer in de verbinding geplaatst.
De return waarde geeft het volgende aan:

- 0 succes
- 1 opgegeven periode verstreken
- 3 onderbroken

Syntax

```
SEND_MESSAGE(<naam>, <timeout>, <grootte>)
```

naam:	de naam van de verbinding waarin de boodschap moet worden geplaatst.
timeout:	de periode waarin geprobeerd wordt de boodschap in de verbinding te plaatsen. Dit kan worden bemoeilijkt doordat geen lock op de verbinding kan worden verkregen of doordat de verbinding steeds te vol is. Standaard is de time-out ingesteld op 86.400.000 seconden (= 1000 dagen).
grootte:	de maximale omvang van de verbinding in bytes. De standaard grootte is 8192 bytes. De totale omvang van alle boodschappen in de verbinding mag niet groter zijn dan deze waarde.

Voorbeeld

```
DBMS_PIPE.SEND_MESSAGE('publieke_verbinding',10,8192);
```

6.7.1.4 RECEIVE_MESSAGE

Deze functie haalt een boodschap uit de opgegeven verbinding en plaatst deze in de lokale boodschap buffer. Als een boodschap wordt opgehaald uit een impliciet aangemaakte pipe en het is de laatste boodschap dan zal RECEIVE_MESSAGE de pipe verwijderen. Als return waarde bestaan de volgende mogelijkheden:

- 0 succes
- 1 opgegeven periode verstreken
- 2 boodschap in de verbinding te groot voor de buffer.
- 3 onderbroken.

Syntax

```
RECEIVE_MESSAGE(<naam>, <timeout>)
```

naam:	naam van de verbinding waaruit de boodschap moet worden ontvangen.
timeout:	de periode in seconden waarin gewacht wordt op de boodschap. Standaard is dit de waarde van de constante MAXWAIT die gedefinieerd is als 86400000 seconden (1000 dagen).

Voorbeeld

```
DBMS_PIPE.RECEIVE_MESSAGE('publieke_verbinding',10);
```

6.7.1.5 NEXT_ITEM_TYPE

Deze functie retourneert het type van de zojuist opgehaalde boodschap in de lokale buffer als gevolg van een RECEIVE_MESSAGE. Hierdoor kun je bij het 'unpacken' van de boodschap in een variabele het correcte type gebruiken. Het datatype van de volgende boodschap wordt geretourneerd als een nummer. Dit nummer heeft de volgende betekenis:

- 0 Een lege boodschap..
- 6 NUMBER
- 9 VARCHAR2
- 12 DATE

Syntax

```
NEXT_ITEM_TYPE
```

Voorbeeld:

```
v_return_waarde := dbms_pipe.receive_message('publieke_verbinding',10);
msg_type := dbms_pipe.next_item_type;
if msg_type = 6 then ......
```

6.7.1.6 UNPACK_MESSAGE

Deze procedure haalt een boodschap uit de lokale boodschap buffer. Deze boodschap is eerst in de buffer geplaatst door RECEIVE_MESSAGE.

Syntax

```
UNPACK_MESSAGE(<informatie>) /* dit is een OUT parameter */
```

informatie:	de boodschap die uit de buffer wordt gehaald. Deze kan de datatypen VARCHAR2, NUMBER of DATE hebben.

Voorbeeld

```
DBMS_PIPE.UNPACK_MESSAGE(boodschap);
```

6.7.1.7 RESET_BUFFER

Deze procedure zet PACK_MESSAGE en UNPACK_MESSAGE weer op 0. Normaal gesproken is het niet nodig deze procedure aan te roepen.

Syntax

```
RESET_BUFFER
```

6.7.1.8 PURGE

Deze procedure maakt de inhoud van de opgegeven verbinding leeg. Het kan zijn dat de inhoud van de lokale buffer wordt overschreven door boodschappen die zich nog in de verbinding bevinden wanneer deze wordt leeggemaakt.

Syntax

```
PURGE(<naam>)
```

naam:	de naam van de verbinding die moet worden leeggemaakt.

Voorbeeld

```
DBMS_PIPE.PURGE('publieke_verbinding');
```

6.7.1.9 UNIQUE_SESSION_NAME

Deze functie retourneert een naam van maximale lengte 30, die uniek is in alle sessies die met de database verbonden zijn. Deze functie kan gebruikt worden voor een naam voor een verbinding, die nog niet in gebruik is.

Syntax

```
UNIQUE_SESSION_NAME
```

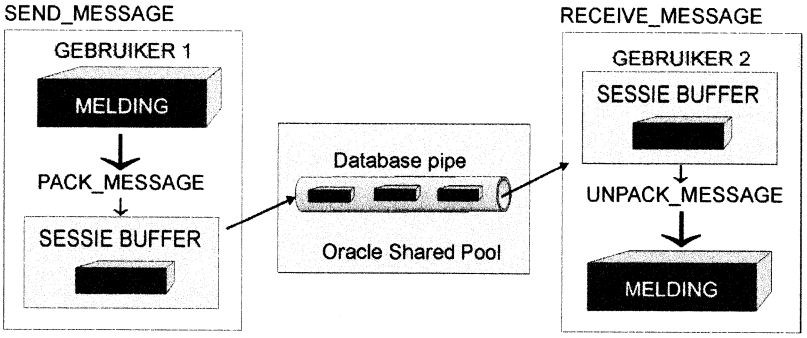
6.7.2 Het gebruik van DBMS_PIPE

Het versturen van boodschappen met behulp van de package DBMS_PIPE is niet afhankelijk van transacties, zoals wel het geval is met boodschappen die met behulp van de package DBMS_ALERT worden verstuurd.

Een verbinding wordt automatisch gestart de eerste keer dat deze wordt aangehaald in een aanroep van de package DBMS_PIPE. Elke verbinding kan door meerdere gebruikers worden gebruikt om te verzenden en te ontvangen.
De volgorde waarin de procedures en functies in de package worden aangeroepen is in het algemeen als volgt:

PACK_MESSAGE:	plaatst de boodschap in de lokale buffer.
SEND_MESSAGE:	plaatst de boodschap vanuit de lokale buffer in de verbinding.
RECEIVE_MESSAGE:	haalt de boodschap uit de verbinding naar de lokale buffer.
UNPACK_MESSAGE:	haalt de boodschap uit de lokale buffer.

In een schema ziet dit er als volgt uit:



.

Voorbeeld

```
declare
        v_dummy       integer;
begin
  dbms_pipe.pack_message('mijn gebruikersnaam is '||user);
  v_dummy := dbms_pipe.send_message('pipe_nwg29');
end;
/
```

```
declare
 v_boodschap  VARCHAR2(50);
 v_dummy      integer;
begin
  v_dummy := dbms_pipe.receive_message ('pipe_<gebruikersnaam>',5);
  dbms_pipe.unpack_message (v_boodschap);
  dbms_output.put_line (v_boodschap);
end;
/
Mijn gebruikersnaam is <gebruikersnaam>
```

In de aanroep van de functie RECEIVE_MESSAGE is een periode van 5 seconden opgenomen waarin moet worden gewacht op een boodschap. Wordt deze periode niet ingevuld en er is geen boodschap in de verbinding, dan zal 1000 dagen worden gewacht voordat een melding verschijnt.

6.8 Vergelijking DBMS_ALERT en DBMS_PIPE

We hebben als laatste in dit hoofdstuk kennis gemaakt met een tweetal communicatieve packages die veel overeenkomsten hebben. Toch zijn er wel verschillen op te merken. De belangrijkste noemen we hieronder:

Alert

- Eenrichtingsverkeer
- Specifieke cliënt adressering
- Reageert pas na commit

Pipes

- Tweerichtingsverkeer
- Wordt door iedere sessie ontvangen

7 DML-triggers

7.1 Inleiding

Database triggers zijn als het ware in de database opgeslagen procedures die impliciet uitgevoerd worden wanneer er een bepaalde actie plaatsvindt. In oudere versies van Oracle bestaan alleen DML-triggers, die afgaan op INSERT-, UPDATE- en DELETE-statements. Dit zijn voor de meeste toepassingen echter nog steeds de belangrijkste triggers. In Oracle bestaan ook INSTEAD OF-triggers, die afgaan op DML statements op views. Daarnaast bestaan in Oracle sinds enige tijd tevens DDL- en system event-triggers. Deze triggers gaan onder meer af op een grote verscheidenheid aan DDL-statements en op het starten en afsluiten van de database. DDL- en system event-triggers worden in hoofdstuk 8 behandeld.

7.2 Toepassing van DML-triggers

DML-triggers hebben een groot aantal toepassingen, zoals :

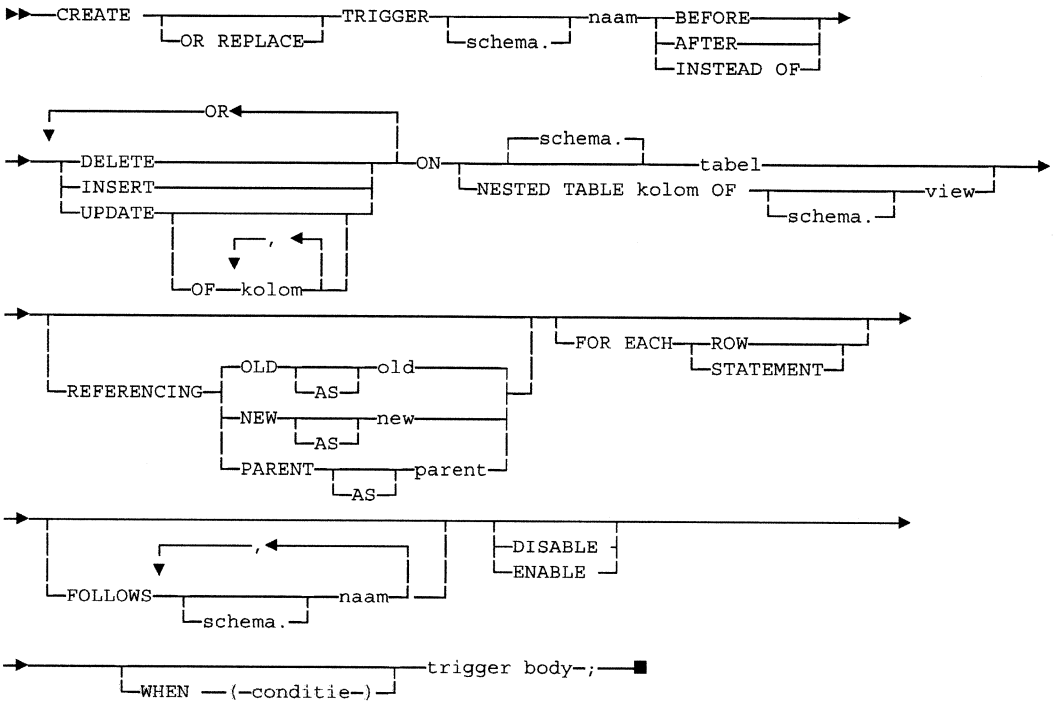
- Het definiëren van complexe beveiligingen, zoals garanderen dat een tabel alleen tijdens kantooruren gewijzigd kan worden.
- Het automatisch vullen van kolommen aan de hand van wijzigingen in andere kolommen.
- Het definiëren van business rules (bedrijfsregels) die niet met de door Oracle gedefinieerde constraints afgedwongen kunnen worden. Indien mogelijk verdient het gebruik van de door Oracle gedefinieerde constraints de voorkeur, met het oog op de performance. Een belangrijk verschil tussen constraints die met triggers afgedwongen worden en de door Oracle gedefinieerde constraints is dat de door Oracle gedefinieerde constraints altijd geldig zijn. Triggers controleren alleen de gegevens van een tabel als deze worden gewijzigd. Het is dus mogelijk dat een tabel inconsistente gegevens bevat, namelijk gegevens die reeds toegevoegd waren voordat de trigger gedefinieerd werd.
- Het verzamelen van statistische gegevens omtrent welke gebruikers welke tabellen benaderen, dit noemen we auditing.
- Het bijhouden van een tabel die een exacte kopie van een andere tabel moet zijn.

Net als bij procedures en functies het geval is kunnen ook triggers in meerdere talen gemaakt worden. Naast PLSQL mag ook gebruik worden gemaakt van C en van Java. Deze procedures maakt u los aan en binnen de trigger roept u deze routine aan met een CALL. Deze call ziet er hetzelfde uit als het CALL statement in hoofdstuk 2. Echter u kunt geen INTO gebruiken. Dit betekent dus dat u alleen procedures kunt aanroepen en geen functies. Tevens kan er geen gebruik worden gemaakt van bind variables. Wel kunt u gewoon gebruik maken dan de :new en :old variabelen (deze worden verderop in dit hoofdstuk uitgelegd).

7.3 DML-triggers

DML-triggers gaan alleen af bij bewerkingen op database tabellen. Bij het aanmaken van een trigger moet aangegeven worden bij welk DML-statement (INSERT, DELETE of UPDATE) de trigger moet worden uitgevoerd. Een trigger kan ook bij meerdere DML-statements afgaan. Triggers worden gedefinieerd met behulp van het CREATE TRIGGER statement.

Syntax



BEFORE, AFTER, INSTEAD OF

Geeft aan wanneer de trigger moet worden uitgevoerd: voor of na de operatie, of in plaats van de operatie. Een INSTEAD OF trigger kan alleen op een view worden geplaatst. Op tabellen zijn deze triggers niet toegestaan.

DELETE, INSERT, UPDATE

De operatie waarop de trigger wordt gedefinieerd. Als bij update geen kolommen worden gespecificeerd dan gaatde trigger op elke kolom af.

FOR EACH ROW/ STATEMENT

Geeft aan of het om een ROW-trigger gaat (die voor iedere rij die wordt gemanipuleerd afgaat) of een STATEMENT-trigger (die voor één statement slechts één keer afgaat). Default is een trigger een statement trigger behalve voor instead of triggers dat zijn altijd row triggers.

FOLLOWS	Zorgt ervoor dat een bepaalde trigger pas afgaat nadat de trigger uit de FOLLOWS clausule is afgegaan. De triggers moeten aangemaakt worden op dezelfde tabel en de leidende trigger dient foutloos gecompileerd te zijn maar deze hoeft niet *enabled* te zijn. Deze clausule is specifiek voor Oracle11*g*.
ENABLE/DISABLE	Een trigger kan ENABLED (default) of DISABLED aangemaakt worden. In het laatste geval is de trigger na het aanmaken niet actief. Deze clausule is specifiek voor Oracle11*g*.
WHEN (conditie)	Geeft een eventuele conditie waaraan voldaan moet worden om de trigger te laten afgaan.
Trigger body	De PL/SQL code van de trigger.

Binnen de verschillende triggertypen wordt nog een onderscheid gemaakt in de tijd waarop deze afgaan:

- BEFORE statement trigger
- AFTER statement trigger

Een BEFORE INSERT-trigger gaat af voordat het INSERT-statement is uitgevoerd. We kunnen de waarden die worden ingevoerd controleren of wijzigen.
Bij een AFTER INSERT kunnen de gegevens niet meer gewijzigd worden, aangezien de rij al is toegevoegd. Een AFTER- trigger wordt vaak gebruikt om in een andere tabel vast te leggen dat iets gebeurd is.

Daarnaast is er nog een onderscheid in hoe vaak de trigger afgaat:

Rij-triggers	Deze triggers worden uitgevoerd voor iedere rij die door een DML-statement gewijzigd, toegevoegd of verwijderd wordt. Het aantal keren dat de trigger wordt uitgevoerd hangt af van het aantal rijen dat door het statement wordt verwerkt. Bij het definiëren van de trigger wordt FOR EACH ROW opgegeven, zoals we hebben gezien in het vorige voorbeeld.
Statement-trigger	Deze triggers worden uitgevoerd voor ieder statement dat uitgevoerd wordt en dat betrekking heeft op een bepaalde tabel. De trigger gaat ook af als er geen rijen worden gewijzigd. Deze triggers worden maximaal één keer uitgevoerd per DML-statement.

Het is mogelijk om meerdere rij/statement triggers van hetzelfde type op een tabel aan te maken. Zo kunnen er dus meerdere before insert triggers op dezelde tabel gedefineerd worden. Op deze manier kunnen bedrijfsregels van verschillende applicaties dus als losse triggers aangemaakt worden en hoeft bestaande trigger-code dus niet aangepast en overschreven te worden.

De volgorde waarin de triggers afgaan is als volgt:

1. BEFORE statement trigger
   Voor elke rij uit het statement
   a. BEFORE rij trigger
   b. Het statement dat wordt uitgevoerd
   c. AFTER rij trigger
8. AFTER statement trigger

Het moment van afgaan wordt vaak als afkorting in de naam van de trigger gebruikt: een BRUI -trigger is een **B**efore each **R**ow **U**pdate en **I**nsert, een ASD -trigger een **A**fter **S**tatement **D**elete trigger enz.

7.3.1 Statement triggers

Indien de clausule FOR EACH ROW uit het CREATE TRIGGER statement niet gebruikt wordt, wordt de betreffende trigger een statement trigger. Dit betekent dat de trigger slechts eenmaal wordt uitgevoerd onafhankelijk van het aantal rijen dat wordt verwerkt.

Bij een statement trigger heeft het specificeren van de optie BEFORE tot gevolg dat de trigger wordt uitgevoerd, voordat een statement uitgevoerd wordt. De optie AFTER heeft in dit geval tot gevolg dat de trigger wordt uitgevoerd, nadat een statement uitgevoerd is (en nadat eventuele tabelconstraints gecontroleerd zijn). Wanneer naar aanleiding van het statement een rij in een andere tabel moet worden toegevoegd, is het beter een AFTER trigger te gebruiken. Hierdoor worden eventuele constraints gecontroleerd voordat de INSERT op de andere tabel wordt uitgevoerd. Als de INSERT dan vanwege constraint controle niet kan plaats vinden, wordt ook de rij niet aan de andere tabel toegevoegd.
Een BEFORE trigger is beter wanneer een gebruiker geen wijzigingen in de tabel mag aanbrengen. Er worden geen waarden in de tabel gezet en het controleren van constraints is dan overbodig.

7.3.2 Rij triggers

Bij het definiëren van een rij trigger wordt de FOR EACH ROW clausule gebruikt, hiermee geven we aan dat de trigger voor elke rij uit het statement moet afgaan.
Bij een rij trigger heeft het specificeren van de optie BEFORE tot gevolg dat de trigger altijd uitgevoerd wordt voordat een rij verwerkt wordt. De optie AFTER daarentegen heeft tot gevolg dat de trigger altijd wordt uitgevoerd nadat een rij verwerkt is (en nadat eventuele constraints gecontroleerd zijn).

Indien de optie UPDATE gebruikt wordt (om de trigger uit te laten voeren voor iedere rij die gewijzigd wordt), kan eventueel een aantal kolommen gespecificeerd worden. De trigger wordt in dat geval alléén uitgevoerd als de waarde van een van de betreffende kolommen wordt veranderd.

Bij rij triggers kan gebruik gemaakt worden van :new en :old, om aan waarden van kolommen te refereren die op dat moment worden bewerkt. Bij het uitvoeren van een rij trigger worden er 2 "records" aangemaakt , :new en :old, van het type TRIGGER_TABEL%ROWTYPE. Alle kolommen van de tabel zijn hierin aanwezig, niet alleen de waarden die zijn gewijzigd. Het "record" is geen echt record, we kunnen namelijk alleen aan de losse elementen van deze records refereren.

We kunnen bij rij triggers een voorwaarde opgeven wanneer de trigger moet afgaan, hiervoor gebruiken we de WHEN clausule.De voorvoegsels new en old moeten, bij de WHEN clausule, zonder dubbele punt (:) worden opgegeven. De voorwaarde bij de WHEN clausule moet altijd tussen haken staan.

Bij before triggers mogen de new waarden worden aangepast.

Voorbeeld

```
create or replace trigger verlaging_bru_sal before update of sal on p_werknemers
for each row when (new.sal < old.sal)
begin
  dbms_output.put_line('Het salaris van '||:old.naam||' is verlaagd!!!');
end;
```

Deze trigger gaat alleen af als het nieuwe salaris kleiner is dan het oude salaris.

7.3.3 Instead-of triggers

Instead-of triggers (=in-plaats-van triggers) kunnen alleen worden gedefinieerd op views. Deze triggers worden uitgevoerd in plaats van het DML-statement dat de trigger aanroept. Het DML-statement wordt niet meer uitgevoerd.
De instead-of triggers zijn nodig als een view is gebaseerd op een join, of als in de view een groepsfunctie of een GROUP BY wordt gebruikt. Met een instead-of trigger kunnen we er voor zorgen dat de wijziging toch wordt doorgevoerd.

7.4 Beperkingen van DML-triggers

Binnen een DML-trigger kunnen de statements ROLLBACK, COMMIT en SAVEPOINT niet worden gebruikt. Ook DDL-statements zijn niet toegestaan, aangezien die een impliciete COMMIT teweeg brengen. Deze beperking geldt natuurlijk ook voor procedures en functies die door de trigger worden aangeroepen.

De reden hiervoor is dat gegarandeerd moet blijven dat een transactie altijd als geheel slaagt of mislukt.
Stel we willen uit een tabel 10 rijen verwijderen, dan moet het DELETE statement voor alle 10 rijen slagen of mislukken.
Bij het verwijderen blijkt dat de vijfde rij niet kan worden omdat er nog een FOREIGN KEY naar verwijst. Als er per rij een COMMIT zou worden gedaan zouden er 4 rijen definitief worden verwijderd, terwijl de rest van de rijen niet wordt verwijderd. De transactie is dan maar gedeeltelijk geslaagd.

Een autonome trigger heeft de bovenstaande beperkingen niet. Door "PRAGMA AUTONOMOUS_TRANSACTION" toe te voegen aan de declaratiesectie van een trigger, wordt binnen de hoofdtransactie een autonome transactie gestart.

Hierdoor kunnen COMMIT, ROLLBACK, SAVEPOINT en met behulp van dynamische SQL ook DDL-statements binnen de trigger gebruikt worden.

Binnen de trigger body kunnen geen LONG of LONG RAW variabelen worden gedeclareerd. Bij LONG kolommen kunnen de voorvoegsels old en new niet worden gebruikt.

Er is een mooie volgorde genoemd voor before en after en statement en row triggers. Helaas bestaat er geen volgorde voor triggers op hetzelfde moment. Als u op een en dezelfde tabel bijvoorbeeld twee before row twee triggers legt één op insert en delete en de andere op insert en update en u voert een insert uit dan zal het volkomen willekeurig zijn welke trigger eerst uitgevoerd word. U kunt ook gewoon bijvoorbeeld twee BRI triggers maken op dezelfde tabel Oracle vindt dat prima. Als de code persé in een specifieke volgorde moet worden afgewerkt dan bent u dus verplicht om voor elk moment maar een trigger te maken.

7.4.1 Beperkingen rij triggers

Een rij trigger mag niet lezen of schrijven in een mutating table. Een rij trigger mag geen kolommen wijzigen in een constraining table waarop de primary, foreign, of unique key is gedefinieerd.

Een mutating table is de tabel die op dat moment wordt gewijzigd door een DML-statement. Als er uit een child tabel rijen worden verwijderd omdat de FOREIGN KEY met DELETE CASCADE optie is ingesteld, noemen we deze tabel ook een mutating table. Een trigger mag niet lezen of schrijven in deze tabellen. Als dat wel gebeurt, verschijnt bij het uitvoeren van de trigger de volgende foutmelding:

```
ORA-04091: table <USER>.P_WERKNEMERS is mutating, trigger/function may not see it
```

Een constraining table is een tabel die door de trigger wordt gelezen omdat er een verwijzing bestaat naar deze tabel door een FOREIGN KEY. De trigger mag de kolommen waarnaar wordt verwezen, de PRIMARY KEY of UNIQUE KEY kolommen, niet wijzigen.
Aangezien bij wijzigingen van die kolommen er in de mutating table wordt gecontroleerd of er nog verwijzingen bestaan. De mutating table mag echter niet worden gelezen.

Op de tabel P_WERKNEMERS is een trigger gedefinieerd, dat is de **mutating table**.

Kantnr	Persnr	Naam
10	5810	Heuvel
10	6221	Kraay
10	8222	Manders

Foreign key relatie

Kantnr	Naam	Plaats
10	Boekhouding	Amsterdam
20	Onderzoek	Utrecht
30	Verkoop	Den Haag
40	Productie	Arnhem

De tabel P_KANTOREN is de **constraining table**. Wijzigingen van de kolom KANTNR in deze tabel hebben tot gevolg dat gecontroleerd wordt in P_WERKNEMERS of er geen referentie is naar die rij.

7.5 Gebruik van DML-triggers

Net als procedures en packages wordt de code van de trigger gecompileerd opgeslagen in de datadictionary. Zodra de tekst van een trigger te groot wordt, verdient het de aanbeveling om deze in aparte procedures op te slaan en deze procedures aan te roepen in de trigger.

Als een trigger wordt gedefinieerd op meerdere DML-statements kunnen we met behulp van de functies INSERTING, UPDATING en DELETING bepalen op welk statement de trigger is afgegaan, de functies geven een BOOLEAN terug.

7.5.1 Auditing

We kunnen triggers gebruiken om bepaalde acties op tabellen te auditen.
Binnen Oracle kunnen, met het AUDIT statement, allerlei acties worden geregistreerd, niet alleen DML maar ook DDL-statements. Dit wordt over het algemeen alleen door de database beheerder gedaan. Het is echter niet mogelijk om met het AUDIT statement per rij wijzigingen bij te houden. Als dit wel wenselijk is moet er een rij trigger worden gedefinieerd, die de wijzigingen bijhoudt.
Audit alleen als het echt noodzakelijk is, er wordt veel informatie verzameld die veel ruimte in beslag kan nemen.

7.5.2 Integriteit van gegevens

Daarnaast kunnen we triggers gebruiken om de integriteit van gegevens binnen de database te bewaren. Hier geldt dat als het mogelijk is om integriteit constraints aan te maken dat de voorkeur heeft boven het definiëren van een trigger.
Zijn er echter complexe controles of bedrijfsregels (business rules) die moeten worden ingesteld, dan kunnen die met behulp van een trigger worden afgedwongen.

7.6 Cascading triggers

De mogelijkheid bestaat dat een statement uit een trigger ervoor zorgt dat een andere trigger aangeroepen wordt (omdat bijvoorbeeld een rij aan een tabel toegevoegd wordt, waarop een INSERT rij trigger gedefinieerd is). Dit soort triggers noemen we cascading. Het gebruik van cascading triggers moet zeer zorgvuldig gebeuren. Indien bijvoorbeeld een BEFORE INSERT rij trigger een rij toevoegt aan de tabel waarop hijzelf gedefinieerd is, wordt een loop gestart.

Triggers kunnen met het DROP TRIGGER statement worden verwijderd.

Syntax

```
>>── DROP TRIGGER ── triggernaam ──><
```

Triggers worden automatisch ingeschakeld nadat zij gecreëerd zijn. Triggers kunnen uitgeschakeld en vervolgens weer ingeschakeld worden.

Syntax

```
>>── ALTER TRIGGER ── triggernaam ─┬─ ENABLE ──┬──><
                                   └─ DISABLE ─┘
```

Eventueel kunnen ook in één keer alle triggers uit- of ingeschakeld worden die betrekking hebben op een bepaalde tabel.

Syntax

```
>>── ALTER TABLE ── tabelnaam ─┬─ ENABLE ──┬─ ALL TRIGGERS ──><
                               └─ DISABLE ─┘
```

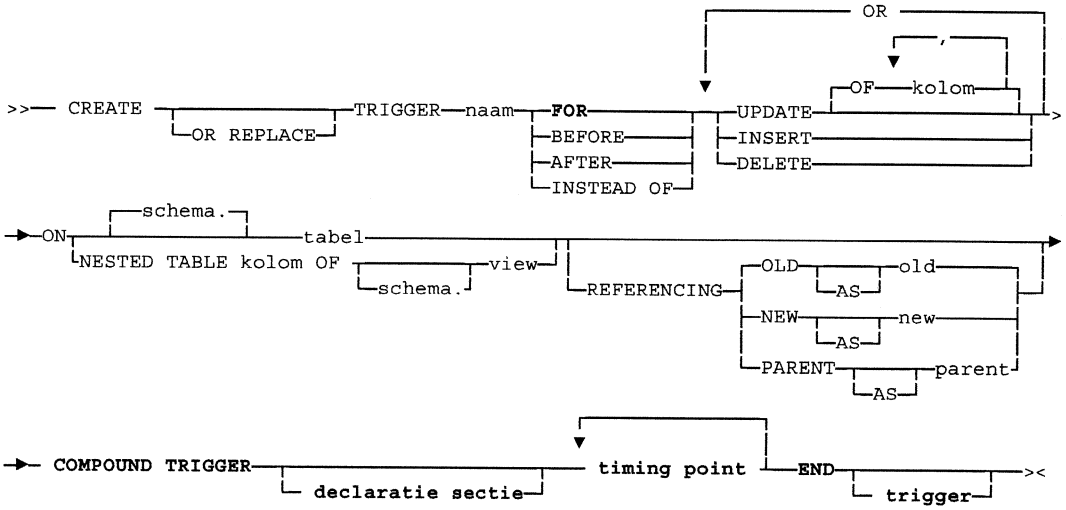
7.7 Nieuw in Oracle 11g

In deze paragraaf zullen we een aantal zaken met betrekking tot triggers bespreken die alleen toepasbaar zijn op een Oracle11*g* database. Oracle heeft in deze versie van de database een nieuw type DML trigger toegevoegd en daarnaast twee nieuwe opties toegevoegd CREATE TRIGGER clausule.

7.7.1 Compound triggers

Een nieuw type trigger in Oracle11*g* is de zogenaamde compound triggers (compound=samengesteld). Met behulp van een compound trigger is het mogelijk om de code van verschillende triggers te groeperen in één trigger. Een compound trigger kan dus bestaan uit een willekeurige combinatie van de bestaande triggers die afgaan op Before Statement, Before Row, After Row en After Statement.

Syntax



In de CREATE TRIGGER clausule zien we met betrekking tot de compound trigger bovenin de nieuwe FOR clausule die aan het einde wordt gevolgd door de COMPOUND TRIGGER clausule. Een compound trigger heeft een optionele declaratie sectie en minstens één timing-point sectie. In de declaratie sectie (dit is altijd de eerste sectie) worden variabelen en sub-programma's gedeclareerd die gebruikt kunnen worden in de timing-point secties. Wanneer een trigger afgaat zal allereerst de declaratie sectie uitgevoerd worden daarna zullen de timing-point secties volgen. De volgende timing point secties kunnen in een compound trigger op een tabel worden opgenomen:

- before statement
- before each row
- after each row
- after statement

Wanneer een compound trigger op een view wordt aangemaakt dan zal deze trigger alleen een INSTEAD OF EACH ROW timing-point sectie bevatten, een dergelijke trigger mag geen andere timing-point secties bevatten.

Timing-point sectie moeten in een vast volgorde gedefinieerd worden. Wanneer een bepaalde timing-point sectie niet in de trigger is opgenomen dan zal er dus ook niets uitgevoerd worden tijdens het betreffende timing point. Iedere sectie kan de functies Inserting, Updating, Deleting en Applying bevatten.
Doordat alle timing point secties toegang hebben tot de variabelen en sub-programma's uit de declaratie sectie kan data tussen de verschillende secties gedeeld worden en kunnen we bijvoorbeeld lopende totalen bijhouden. Deze zogenaamde common state wordt bereikt wanneer het triggering statement wordt gestart en wordt weer vrijgegeven wanneer het triggering statement wordt afgesloten, zelfs wanneer het triggering statement een foutmelding oplevert.

Door het gebruik van een compound trigger zijn er veel meer mogelijkheden en is een ontwikkelaar veel flexibeler bij ingewikkelde transacties. Doordat variabelen en subprogramma's hergebruikt kunnen worden, behoren constructies met bijvoorbeeld globale package variabelen tot het verleden.

Een compound trigger heeft vooral performance voordelen ten opzichte van normale triggers wanneer gebruik wordt van de FOR ALL en BULK COLLECT clausules. Meer hierover in de cursus: Oracle Database 11g: PL/SQL geavanceerd

Door gebruik te maken van compound triggers kunnen we eenvoudig de Mutating-Table Errors voorkomen. Variabelen zijn bij een dergelijke trigger gedefinieerd in de algemene declaratie sectie en kunnen hierdoor hergebruikt worden in de before en after each row timing secties.

Voorbeeld:

```
create or replace trigger comp_bu_prijs for update of prijs on mutating_cursussen
compound trigger
  cursor c_haal_max is
    select max(prijs)
    from mutating_cursussen;
v_max_prijs number;

before statement is
begin
  open c_haal_max;
  fetch c_haal_max into v_max_prijs;
  close c_haal_max;
end before statement;

before each row is
begin
  if :new.prijs > 1.10*:old.prijs then
    raise_application_error(-20007, 'Verhoging mag niet meer dan 10% zijn');
  elsif :old.prijs <> v_max_prijs and :new.prijs > v_max_prijs
     then raise_application_error(-20008, 'De prijs is te hoog');
 end if;
end before each row;
end comp_bu_prijs;
```

Wanneer we onderstaand update statement uitvoeren:

```
update mutating_cursussen
set prijs =4000
where naam='SQL 5 dagen';
```

Dan zal de volgende foutmelding verschijnen:

```
Error report:
SQL Error: ORA-20007: Verhoging mag niet meer dan 10% zijn
ORA-06512: at "NWG806.COMP_BU_PRIJS", line 17
ORA-04088: error during execution of trigger 'NWG806.COMP_BU_PRIJS'
```

7.7.2 Disabled Triggers

De tweede vernieuwing in het gebruik van triggers is het *disabled* creëren van een trigger. Tot aan Oracle10*g* waren triggers bij het aanmaken direct actie en konden ze alleen achteraf uitgeschakeld worden door de trigger te *disabled* te maken. Vanaf Oracle11*g* is het ook mogelijk een trigger *disabled* aan te maken. Dit kan bijvoorbeeld handig zijn als de programmeur een trigger, die naar behoren werkt in de testomgeving, wil gaan toevoegen aan een High Availability productie omgeving. Wanneer de trigger door een fout in de code niet compileert zal deze de werking van een bestaande tabel verstoren en wellicht nog veel meer processen.

Om dit risico tegen te gaan kan een trigger disabled worden aangemaakt. De trigger wordt wel aangemaakt en gecompileerd maar zal niet actief zijn. Wanneer de trigger eenmaal foutloos is aangemaakt dan kan deze alsnog met een ALTER TRIGGER statement *enabled* worden.
Als een trigger als disabled wordt gebouwd, kan de programmeur eerst alle mogelijke fouten eruit halen voordat deze live wordt gezet.

Voorbeeld

```
create or replace trigger fout_1
before update on p_werknemers for each row
disable
begin
 :new.naam:=lower(:new.naam);
end;
```

Wanneer de code gecompileerd is dan kunnen we de trigger inschakelen:

```
alter trigger fout_1 enable;

Trigger altered.
```

7.7.3 Ordered Execution

De laatste toevoeging in het gebruik van triggers is de Ordered Execution. We hebben reeds eerder aangegeven dat het mogelijk is om op één tabel meerdere triggers van hetzelfde type aan te maken. De volgorde waarin Oracle deze triggers uitvoert is normaal gesproken willekeurig. Vanaf Oracle11*g* is het echter mogelijk om zelf de volgorde van uitvoeren aan te geven wanneer er op dezelfde tabel meerdere triggers op één timing point bestaan. Hiervoor is de FOLLOWS clausule aan het CREATE TRIGGER statement toegevoegd:

Voorbeeld:

```
create or replace trigger vb_trigger_1
before update on p_werknemers
for each row
begin
... trigger code ...
end;

create or replace trigger vb_trigger_2
before update on p_werknemers
for each row
follows vb_trigger_1
begin
... trigger code ...
end;
```

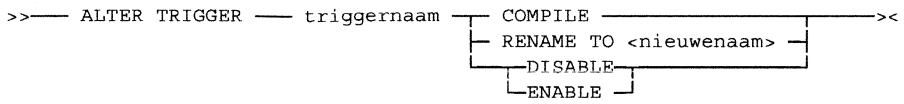
Dit voorbeeld zorgt ervoor dat de before update trigger VB_TRIGGER_2 afgaat nadat de before update trigger VB_TRIGGER_1 is uitgevoerd. Hierbij geldt dat de trigger die wordt genoemd in de FOLLOWS clausule moet bestaan en van hetzelfde type moet zijn. Daarnaast moeten de triggers aangemaakt worden op dezelfde tabel en dient de leidende trigger foutloos gecompileerd te zijn maar hoeft deze niet *enabled* te zijn.

De FOLLOWS clausule kan ook verwijzen naar een compound trigger. In dat geval wordt de trigger uitgevoerd nadat het gelijknamige timing point in de compound trigger is uitgevoerd. Andersom kan een compound trigger ook verwijzen naar een normale trigger. De betreffende timing point sectie wordt dan pas uitgevoerd nadat de gelijksoortige normale trigger is uitgevoerd.

7.8 Datadictionary views

In de datadictionary view USER_TRIGGERS kan de code van triggers worden opgevraagd. In deze view kan ook bekeken worden of een trigger "aan" staat (enabled) of niet. Om te zien of een trigger valid is moet gebruik worden gemaakt van de kolom status van user_objects. Net als bij procedures en functies bestaat ook bij triggers de mogelijkheid om ze te compileren, daarnaast is het mogelijk om triggers een andere naam te geven of de trigger aan of uit te zetten.

Syntax



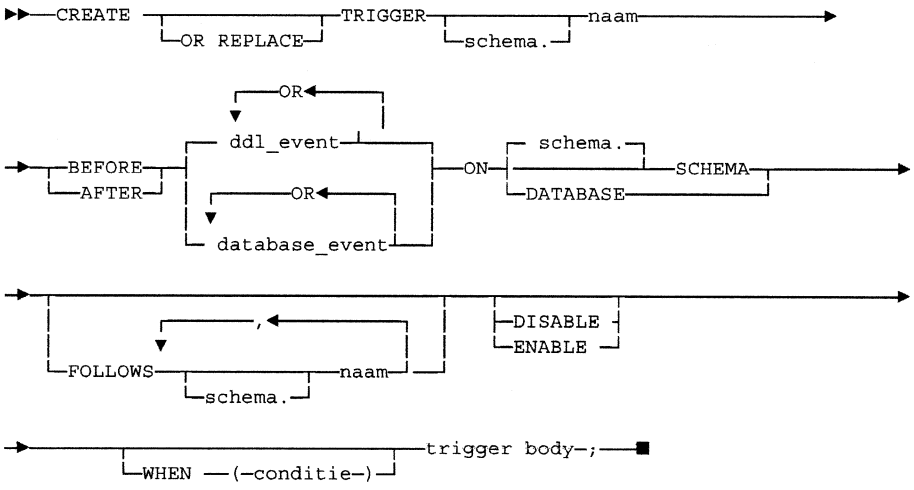
8 System Event en DDL-triggers

8.1 Inleiding

Naast de reeds lang in Oracle bekende DML-triggers kunnen vanaf Oracle8*i* ook triggers worden gemaakt op bepaalde DDL-statements. Hiermee kan bijvoorbeeld worden achterhaald wie een bepaalde tabel heeft verwijderd.

Tevens kunnen er zogenaamde system event triggers worden afgevuurd. Een system event trigger kan onder meer afgaan bij het starten en stoppen van de database, waardoor bepaalde acties direct automatisch kunnen worden uitgevoerd. Ook bij het in- of uitloggen kunnen system event triggers afgaan. Hiermee kan bijvoorbeeld autorisatie worden geregeld of kan een logboek worden bijgehouden. Als laatste kunnen system event triggers afgaan wanneer een bepaalde server error plaatsvindt.

In dit hoofdstuk zullen deze system event en DDL- triggers worden besproken.



8.2 Triggers op DDL-statements

Naast de bekende DML-triggers kunnen er ook triggers worden gemaakt op DDL-statements. Deze triggers kunnen voor de hele database worden gedefinieerd of voor een bepaald schema.

Voorbeeld

We maken een trigger DROPTRIGGER die afgaat als er binnen uw schema geprobeerd wordt de tabel DROPNIET te verwijderen.

```
create or replace trigger droptrigger before drop on <gebruikersnaam>.schema
 when (sys.dictionary_obj_name='DROPNIET')
 begin
   raise_application_error(-20001,'Je mag deze tabel niet verwijderen');
 end;
 /

DROP TABLE dropniet;

Error starting at line 1 in command:
```

```
DROP TABLE dropniet
Error report:
SQL Error: ORA-00604: error occurred at recursive SQL level 1
ORA-20001: Je mag deze tabel niet verwijderen
ORA-06512: at line 2
00604. 00000 -  "error occurred at recursive SQL level %s"
*Cause:    An error occurred while processing a recursive SQL statement
           (a statement applying to internal dictionary tables).
*Action:   If the situation described in the next error on the stack
           can be corrected, do so; otherwise contact Oracle Support.
```

DDL-triggers kunnen ook op databaseniveau worden gedefinieerd. Hiervoor is het Administer database trigger privilege nodig.

```
create or replace trigger nieuwe_tabel after create on database
when (sys.dictionary_obj_type='TABLE')
begin
  insert into tab_aangemaakt values (sys.login_user, sys.dictionary_obj_name
                                     , sysdate);
end;
```

De drie belangrijkste DDL-trigger events zijn CREATE, ALTER of DROP. Ze gaan af voor clusters, functies, indexen, packages, procedures, roles, sequences, synoniemen, tabellen, tablespaces, triggers, types, views en users. Naast de drie genoemde DDL events zijn er nog een paar statements die een trigger kunnen laten afgaan. Deze events zijn eigenlijk alleen belangrijk zijn voor de DBA zoals ANALYZE. Het gebruik van die events is hetzelfde als bij CREATE ALTER en DROP.
Als een DDL-trigger afgaat, worden er verschillende attributen gevuld. Welke precies is afhankelijk van welke trigger af is gegaan.

De belangrijkste attributen zijn:

sysevent	de actie die is uitgevoerd.
database_name	de naam van de database.
login_user	de gebruiker die de actie uitvoert.
dictionary_obj_type	het objecttype.
dictionary_obj_name	de objectnaam.
dictionary_obj_owner	het schema waarbinnen het object is aangemaakt.

In oude releases bestaat in het schema van sys een functie met dezelfde naam als het attribuut, die de waarde van het attribuut teruggeeft. Vanaf Oracle9*i* zijn er ook public synoniemen voor deze functies beschikbaar. Deze synoniemen heten allemaal ora_<attribuut>. Alleen <dictionary> is afgekort tot <dict> in de synoniemen.Hetzelfde geldt voor de attributen die bij de volgende soorten triggers worden beschreven.

8.3 System event triggers

System event triggers zijn triggers die afgaan op database startup of database shutdown ,bij server errors of bij aan en uitloggen. Database startup, LOGON en server error triggers zijn altijd AFTER en een shutdown database trigger en een LOGOFF trigger zijn altijd BEFORE.

8.3.1 Triggers op user logon en logoff

Naast de DDL-triggers zijn er triggers die bij het aanloggen of uitloggen van een gebruiker afgaan. De logon en logoff triggers kunnen op database of schemaniveau worden aangemaakt.

Voorbeeld

We maken twee triggers OPSTARTEN en AFSLUITEN, hiermee wordt geregistreerd hoelang uw eigen sessies duren. De gegevens slaan we op in de tabel TIJDEN.

```
create table tijden (datum date, tijdsduur number);

create or replace trigger opstarten after logon on <gebruikersnaam>.schema
 begin
   insert into tijden (datum) values (sysdate);
 end;
/

create or replace trigger afsluiten before logoff on <gebruikersnaam>.schema
begin
 update tijden set tijdsduur=sysdate-datum
  where tijdsduur is null;
end;
/
```

Wanneer de gebruiker nu een connectie maakt met de database, dan zal de begintijdstip van de nieuwe sessie door de trigger OPSTARTEN vastgelegd worden in de tabel TIJDEN.

```
SELECT *
FROM tijden;

DATUM                     TIJDSDUUR
------------------------- ----------------------
19-FEB-08
```

We kunnen nu bijvoorbeeld ook een trigger AFSLUITEN maken, die van een sessie de tijdsduur berekent. We gaan er hieronder vanuit dat de huidige sessie wordt gekenmerkt door het feit dat er nog geen tijdsduur is ingevuld:

```
create or replace trigger afsluiten before logoff on <gebruikersnaam>.schema
begin
  update tijden set tijdsduur=sysdate-datum
  where tijdsduur is null;
end;
/
```

Wanneer een sessie wordt afgesloten dan kunnen we bekijken hoelang die sessie duurde:

```
SELECT datum
,      round(tijdsduur*(24*60),2) minuten
FROM tijden;

DATUM      MINUTEN
-------- ---------
21-08-00      0.72
```

Attributen bij logon en logoff triggers zijn:

sysevent	Per definitie logon of logoff.
login_user	De naam van de gebruiker die inlogt
database_name	De naam van de database waarop wordt ingelogd.

Logon en logoff triggers kunnen ook voor de hele database worden aangemaakt, hiervoor is weer het Administer database triggers privilege nodig.

8.3.2 Servererror triggers

Servererror triggers reageren op fouten die gemaakt worden bij de interactie met de database. Het gaat dus om fouten die gebruikers maken, zoals het opvragen van een tabel die niet bestaat, of acties uitvoeren waarvoor de gebruiker geen rechten heeft.

Voorbeeld

We willen vaststellen hoe vaak de fout "insufficient privileges" voorkomt. De errorcode van deze foutmelding is 1031. SYS.SERVER_ERROR(1) is een functie die de errorcode van de opgetreden fout retourneert. De trigger is voor de gehele database gedefinieerd, als we geen errorcode zouden opgeven dan registreert de trigger alle fouten van elke gebruiker.

```
create or replace trigger geenrechten after servererror on database
  when (sys.server_error(1)=1031)
begin
  insert into rechten values(sys.login_user, 'Gebruiker heeft geen rechten');
end;
/
```

Vanaf dit moment zal voor iedere keer dat de fout met nummer 1031 (insufficient privileges) optreedt, een rij worden toegevoegd aan de tabel RECHTEN.
De onderstaande trigger is een trigger die elke fout van één gebruiker vastlegt in een tabel:

```
create or replace trigger foutencursist after servererror
 on gebruikersnaam.schema
begin
  insert into fouten values(sys.login_user, sys.server_error(1));
end;
/
```

Hieronder voeren we twee fouten statements uit om het resultaat van de trigger FOUTENCURSIST te bekijken:

```
SELECT *
FROM bestaatniet;

   Error starting at line 1 in command:
   SELECT *
   FROM bestaatniet
   Error at Command Line:2 Column:5
   Error report:
   SQL Error: ORA-00942: table or view does not exist
   00942. 00000 -  "table or view does not exist"
   *Cause:
   *Action:


CREATE TABLE fouten (fout number);

   Error starting at line 1 in command:
   CREATE TABLE fouten (fout number)
   Error at Command Line:1 Column:13
   Error report:
   SQL Error: ORA-00955: name is already used by an existing object
   00955. 00000 -  "name is already used by an existing object"
   *Cause:
   *Action:

SELECT *
FROM fouten;
```

```
GEBRUIKER                                                    FOUTNR
---------------------------------------------------------- ---------
<gebruikersnaam>                                               942
<gebruikersnaam>                                               955
```

Servererror trigger kunnen dus zowel op databaseniveau als op schemaniveau worden gedefinieerd. Wees voorzichtig met het definiëren van deze triggers: ze zullen vaak afgaan, en als ze tabellen vullen kunnen deze tabellen zeer groot worden.
Let erop, dat alleen databasemeldingen worden afgevangen door deze Servererror triggers. Foutboodschappen van bijvoorbeeld SQL Developer, zoals 'unknown command beginning..." kunnen niet door een trigger worden verwerkt.

8.3.3 Shutdown en startup database triggers

Shutdown en startup database triggers kunnen alleen op databaseniveau worden aangemaakt. De shutdown database trigger gaat alleen af als de database op een normale manier wordt afgesloten. Als door een fout de database wordt afgesloten gaat de trigger niet af.

Voorbeeld

Met de triggers DB_SHUTDOWN en DB_STARTUP wordt geregistreerd hoelang de database niet gebruikt kan worden, doordat deze is afgesloten. De tijdsduur wordt in minuten geregistreerd.

```
create or replace trigger db_shutdown before shutdown on database
begin
  insert into db_plat values(sys.database_name, sysdate, null);
end;
/

create or replace trigger db_startup after startup on database
begin
  update db_plat set tijdsduur=(sysdate-tijdstip)*24*60
  where tijdsduur is null;
end;
/
```

Als de database wordt afgesloten wordt het tijdstip vastgelegd in de tabel DB_PLAT en na het opstarten wordt de tijdsduur berekend.
Bij shutdown en startup triggers kunnen geen condities worden opgegeven.

9 Werken met LOBs

9.1 Inleiding

In dit hoofdstuk gaan we werken met Large Objects. Hieronder worden verstaan alle soorten gegevens, character based of binary based, met een grote tot zeer grote omvang. We moeten hierbij denken aan bestanden met afbeeldingen foto's, geluid, video of bestanden uit andere applicaties zoals MS Word, AutoCad, Pagemaker enz.
De manieren waarop dit kan gebeuren en de aandachtspunten hiervan zullen in dit hoofdstuk aan de orde komen.

9.2 LONG en RAW datatypes

De Oracle database heeft al langer de LONG en RAW datatypes ter beschikking maar deze zijn verouderd, hebben specifieke beperkingen en zullen op termijn volledig vervangen gaan worden door de LOB datatypes.
Een paar kenmerken:

- RAW kan maximaal 2000 bytes *binaire* gegevens bevatten met vooraf bepaalde omvang.
- LONG kan een variabele hoeveelheid *character* gegevens bevatten van maximaal 2 Gigabyte.
- LONG RAW kan een variabele hoeveelheid *binaire* gegevens bevatten van maximaal 2 Gigabyte.
- De RAW, LONG en LONG RAW datatypes kunnen alleen maar in de tabel zelf worden opgeslagen.

9.3 Kenmerken van LOBs

- LOBs kunnen opgeslagen worden binnen(Intern) of buiten(Extern) de database
- LOBs kunnen *binnen* de database, in de tabel(In-Line)of buiten de tabel(Out of Line) worden opgeslagen
- Er kunnen meerdere LOB objecten voorkomen in één tabel.
- Voor iedere LOB kolom kunnen we afzonderlijke specifieke opslageisen instellen.
- De maximumomvang voor een LOB in Oracle11*g* ligt in de orde van terabytes en is afhankelijk van de databaseblokgrootte. In eerdere versies was de maximumomvang 4 gigabyte.

  De maximumomvang van LOBs in Oracle11*g* kan als volgt berekend worden:

  (4GB – 1 byte) * (database block size).

Voor specifieke informatie over opslag: volg onze DBA cursussen of raadpleeg de helpbestanden met betrekking tot de Storage Clauses van het CREATE TABLE- en het CREATE TABLESPACE statement.

9 Werken met LOBs

9.4 Interne LOBs

Interne LOBs worden in de database zelf opgeslagen. Dit komt overeen met de gebruikelijke manier van gegevensopslag zoals ook voor de LONG, RAW en LONG RAW kolommen in een tabel. Bij interne LOBs kunnen we daarbij ook nog onderscheid maken tussen *In-Line* en *Out of Line* opslag van de gegevens. Deze gegevens maken dus deel uit van de database en reageren dus ook op database transacties. Ook worden deze gegevens in een database back-up meegenomen. Automatisch worden er indexes aangemaakt voor deze interne LOBs.

9.4.1 In line en Out of Line Storage

Bij het In-Line opslaan van de gegevens worden deze in het record opgeslagen, dus in hetzelfde segment. Het omslagpunt is een grootte van 4kB. Objecten van meer dan 4kB worden niet opgeslagen in het record maar worden daarbuiten opgeslagen in een eigen segment.
Bij out of line storage is er altijd sprake van opslag buiten de tabel in een eigen segment en tablespace. Het segment kan in dezelfde tablespace als de tabel worden opgeslagen, maar eventueel ook in een andere tablespace. In de tabel wordt alleen een referentie opgenomen naar de locatie van het segment. Het opgeven van de in-line of out of line opslag gebeurt tijdens het aanmaken van de tabel middels de storage clausule. De totale LOB bestaat dan dus uit twee dingen: de gegevens zelf buiten het record en de LOB locator (een soort pointer) naar de gegevens in het record. Hierover later meer.

9.4.2 Datatypes

De datatypes welke gebruikt kunnen worden voor een interne LOB zijn de volgende:

BLOB:	Binary Large Object en in het gebruik vergelijkbaar met LONG RAW.
CLOB:	Character Large Object en in gebruik vergelijkbaar met LONG.
NCLOB:	NLS Character Large Object.

Realiseer je dat deze 'datatypes' niet de gegevens zelf bevatten maar een *pointer* zijn naar de werkelijke gegevens.

9.5 Externe LOBs

Kenmerk van een externe LOB is dat dit een apart *bestand* is op een andere locatie dan *in* de database. Omdát deze externe LOBs zich dus niet in de database bevinden, kunnen ze niet worden beïnvloed door database transacties. Voordat we de Externe LOBs kunnen gebruiken moeten we een aantal voorbereidingen treffen. Deze voorbereidingen zullen later aan bod komen. Voor het gebruiken van Externe LOBs hebben we de beschikking over het BFILE datatype.

9.5.1 Datatype BFILE

Het BFILE type wordt gebruikt voor de *verwijzing* naar bestanden met binaire gegevens zoals afbeeldingen,video, geluid. Deze dienen opgeslagen te zijn in het bestandssysteem van het OS. Het is goed te realiseren dat BFILE LOBs alleen maar gelezen kunnen worden. Het onderhoud aan de BFILE kolommen, het verwijzen naar externe objecten, gebeurt in de database.

Het onderhoud aan de bestanden kan niet vanuit de database plaatsvinden. Hiervoor dienen we gebruik te maken van applicaties die deze bestanden kunnen bewerken of onderhouden.

9.6 Voorbereidende werkzaamheden

Om te kunnen werken met BFILE LOBs moet er een Oracle Directory aanwezig zijn. Dit is een object in de database die een pad aangeeft waar de database mag lezen en schrijven. Uiteraard heeft u zelf dan lees en schrijf rechten op die Oracle Directory nodig.

9.6.1 Oracle directory aanmaken

In een voorgaande paragraaf hebben we gelezen dat er een verschil kan worden gemaakt tussen interne en externe LOBs. Indien we gaan werken met Externe objecten zijn deze dus altijd van het type BFILE en worden deze buiten de database opgeslagen.
De locatie waar deze bestanden opgeslagen worden zal bekend moeten zijn in de database anders kunnen we er niet naar verwijzen. We maken deze locatie kenbaar aan de database door er een alias voor aan te maken.

De map waar de BFILE objecten in worden opgeslagen dient zich op dezelfde server te bevinden als de database.

Informeer altijd bij uw DBA over eventuele bestaande *Oracle directories* en de eventuele rechten die u daar al op hebt. We kunnen daarnaast de view DBA_ of ALL_DIRECTORIES raadplegen voor al bestaande directories en waar deze naar toewijzen. Onderstaande waarden kunnen afwijken van de werkelijkheid.

Voorbeeld

```
SELECT *
FROM all_directories;

OWNER DIRECTORY_NAME           DIRECTORY_PATH
----- ---------------------- ------------------------------------------------
SYS   DIR_LOGBESTANDEN         c:\cur\share\log
SYS   DIR_VOOR_CURSIST         c:\cur\share\route
SYS   IDR_DIR                  c:\app\oracle\diag\rdbms\nwg11\nwg11\ir
SYS   DIR_GEGEVENS             c:\cur\share
SYS   AUDIT_DIR                /tmp/
SYS   DATA_PUMP_DIR            /opt/app/oracle/admin/nwg11/dpdump/
SYS   ORACLE_OCM_CONFIG_DIR    /opt/app/oracle/product/11.1.0/db_1/ccr/state

7 rows selected
```

We zien aan de gegevens dat er een directory bestaat met de naam DIR_VOOR_CURSIST en dat deze zich bevindt op c:\cur\share\route. Let wel dit is dus op de *server*. De eigenaar van deze en alle andere Oracle directories is altijd SYS, ook al maken we de Oracle directory zelf aan.

Het maken van een directory gebeurt met het CREATE DIRECTORY statement.

```
>>—CREATE—┬──────────────┬—DIRECTORY—directory—AS—'path name'—><
          └OR REPLACE────┘
```

Voorbeeld

```
create directory dir_voor_cursist as 'e:\share';
```

Om dit statement uit te mogen voeren zullen we hiervoor ook over de juiste rechten moeten beschikken. Dit recht kan worden toegekend met het Grant Create Any Directory statement en dient door de DBA uitgevoerd te worden. Op zelf gemaakte directories mag u alles.

Oracle "onthoudt" onderhuids wie de maker was. Andere gebruikers kunnen lees (READ) en schrijf (WRITE) rechten krijgen op directories.

Voorbeeld

```
grant read on directory dir_voor_cursist to public;
```

Let op het gebruik van het keyword DIRECTORY. Dit is verplicht, aangezien Oracle anders denkt dat DIR_VOOR_CURSIST een tabel is. In het voorbeeld worden de rechten toegekend aan PUBLIC, hetgeen neerkomt op aan iedereen.

Zoals u uit bovenstaande wellicht begrepen heeft, heeft u leesrechten op de directory DIR_VOOR_CURSIST, Deze verwijst naar de directory route in de share 'SHARE' op de server. Deze directory gaan we tijdens de voorbeelden gebruiken.

9.7 Aanmaken tabel met LOB datatypes.

Het maken van een tabel met LOB datatypes is erg eenvoudig.

Voorbeeld

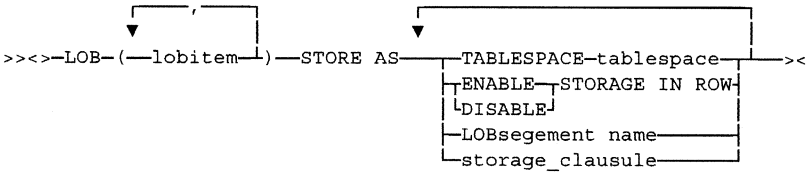
```
CREATE TABLE LOB_kantoren
( kantnr    NUMBER  primary key
, naam      VARCHAR2(50)
, plaats    VARCHAR2(50)
, route     LONG
, foto      BLOB
, kaart     BFILE );
```

Deze tabel is nu aangemaakt met de standaard instellingen met betrekking tot de LOBs.

9.7.1 Specificatie CLOB / BLOB kolom

We hebben zojuist gezien dat de LOB kolommen zonder enige specificatie zijn aangemaakt in de tabel. Uiteraard is het mogelijk een LOB kolom van specificaties te voorzien maar dit is meer het terrein van de DBA.
Hieronder een deel van de LOB storage

Syntax



De definitie kan aan het CREATE TABLE statement worden toegevoegd na de sluithaak van de kolomdefinities. In ons geval willen we de kolom FOTO 'Out-Of-Line' laten opslaan ook als het om minder van 4k aan data gaat. Standaard is STORAGE IN ROW enabled en wordt alleen data groter dan 4k Out-Of-Line opgeslagen.
De aanpassing ziet er als volgt uit:

```
LOB (foto) STORE AS foto_segment
      (TABLESPACE User_Data
       STORAGE (INITIAL 100K NEXT 100K)
       DISABLE STORAGE IN ROW)
```

Hierbij dient opgemerkt te worden dat de Storage Clause optioneel is.
Als de wijziging is uitgevoerd zijn de opslag gegevens als volgt terug te zien:

```
select substr(segment_name,1,20) segname
,      segment_type
,      tablespace_name
from user_segments
where segment_name in('LOB_KANTOREN','FOTO_SEGMENT');

SEGNAME              SEGMENT_TYPE        TABLESPACE_NAME
-------------------- ------------------  ---------------------------
LOB_KANTOREN         TABLE               USER_DATA
FOTO_SEGMENT         LOBSEGMENT          USER_DATA
```

9.7.2 Specificatie BFILE kolom

Het aanmaken van een BFILE kolom bestaat uit het benoemen van een kolom met het datatype BFILE. Tijdens het invoegen van een record dienen we aan te geven in welke Oracle Directory de informatie zich bevindt.

9.8 Invoeren gegevens

De uitgangspositie van onderstaande voorbeelden is de tabel LOB_KANTOREN met een viertal records in de tabel LOB_KANTOREN toegevoegd. De LOB kolommen zijn met NULL gevuld en de LONG kolom bevatten een aantal regels tekst

De bedoeling is dat we de kolom FOTO gaan vullen met een afbeelding van het hoofdkantoor van de betreffende firma. Deze bestanden staan op de share en zijn te herkennen aan de letters HQ aan het einde van de bestandsnaam.
De afbeeldingen die we gaan gebruiken voor de BFILE zijn de plattegronden van de omgeving van het kantoor. Deze zijn herkenbaar aan het woordje MAP in de bestandsnaam.

9.8.1 Invoeren gegevens BFILE LOBs

Het eenvoudigst is het werken met de BFILE datatypes. Zoals al eerder gezegd is het niet mogelijk de inhoud van dit datatype te bewerken vanuit Oracle.
De syntax voor het vullen van een BFILE kolom is de volgende:

```
INSERT INTO tabelnaam VALUES ( BFILENAME ('Oracle_directory_naam', 'bestandsnaam'));
```

of :

```
UPDATE tabelnaam
SET kolomnaam = BFILENAME ('Oracle_Directory_naam', 'bestandsnaam') WHERE condities;
```

We gaan voor kantoor 10, Apple office, de plattegrond van de omgeving van het kantoor toevoegen. Dat doen we met het volgende statement:

Voorbeeld

```
update lob_kantoren
set kaart = bfilename('DIR_VOOR_CURSIST', 'AppleMap.gif')
where kantnr= 10;
```

Het resultaat kunnen we controleren door het script `VB_H9_1.sql` te starten. Deze maakt een procedure aan, waarmee kan worden gecontroleerd of het gevraagde bestand aanwezig is en wat de omvang daarvan is. De inhoud van het script staat hieronder vermeldt. In deze procedure maken we hiervoor gebruik van de functies `FileExists()` en `GetLength()`.
De Oracle directorynaam dient in HOOFDLETTERS opgegeven te worden! Dit wordt helaas niet afgevangen door de functie zelf.

Voorbeeld

```
create or replace procedure bestaat_bestand( p_oradir varchar2,
                                             p_binbestand varchar2) is

v_locatie bfile := null;
v_omvang number:=null;

begin
  v_locatie := bfilename(p_oradir,p_binbestand);
  if dbms_lob.fileexists(v_locatie)=1 then
     v_omvang:=dbms_lob.getlength(v_locatie);
     dbms_output.put_line('bestand is aanwezig en heeft een omvang van: '
     ||v_omvang||' bytes.');
  else
     dbms_output.put_line('bestand kan niet gevonden worden !');
  end if;
end;
/

exec bestaat_bestand('DIR_VOOR_CURSIST','AppleMap.gif')
Bestand is aanwezig en heeft een omvang van: 17532 bytes.
```

9.9 Toevoegen externe gegevens aan interne BLOBs

Het laden van externe gegevens (BFILE) in een BLOB of CLOB is een geheel ander verhaal en kan niet in SQL worden uitgevoerd. Hiervoor dienen we een procedure te schrijven die ervoor zorgt dat deze gegevens worden opgehaald en toegekend worden aan de BLOB of CLOB kolom. Een van de meest in het oog springende elementen is het gebruik van een *locator* voor het record waarin het LOB object geplaatst wordt.

```
create or replace procedure laadblob( p_oradir in varchar2
                                    , p_verwijzing_blob in varchar2
                                    , p_kantnr in number ) is
v_verwijzing_bfile bfile;
v_verwijzing_foto  blob;
v_bfilesize pls_integer;
e_bestaatniet EXCEPTION;
BEGIN
   IF DBMS_LOB.FILEEXISTS(bfilename(p_oradir,p_verwijzing_blob)) = 0 then
         RAISE E_BESTAATNIET;
   ELSE  -- bepalen locator en toekennen aan BFILE variabele.
         v_verwijzing_bfile := bfilename(p_oradir,p_verwijzing_blob);
         -- openen van het Bfile bestand
         DBMS_LOB.FILEOPEN(V_VERWIJZING_BFILE);
   END IF;
   -- bepalen omvang van Bfile bestand
   v_bfilesize := dbms_lob.getlength(v_verwijzing_bfile);
   -- bepalen locatie(record/kolom) in tabel
   SELECT foto INTO v_verwijzing_foto FROM lob_kantoren
   WHERE kantnr = p_kantnr FOR UPDATE;
   -- uiteindelijke update van record
   DBMS_LOB.LOADFROMFILE (v_verwijzing_foto , v_verwijzing_bfile , v_bfilesize);
   -- Sluiten Bfile bestand
   DBMS_LOB.FILECLOSE(V_VERWIJZING_BFILE);
   commit;
EXCEPTION
   WHEN E_BESTAATNIET THEN DBMS_OUTPUT.PUT_LINE('BFILE bestand bestaat niet!');
   WHEN OTHERS THEN raise_application_error (-20001,sqlerrm);
END;
/
```

Een aantal zaken vallen op in het script:

`v_verwijzing_bfile := BFILENAME(oradir,blobbestand);`

Met deze opdracht wordt de locatie van het externe bestand als BFILE variabele toegekend.

`DBMS_LOB.FILEOPEN(V_VERWIJZING_BFILE);`

Indien we met behulp van `DBMS_LOB.LOADFROMFILE()` de inhoud van een BFILE in een BLOB of CLOB-kolom willen plaatsen dan moet vooraf eerst de BFILE geopend worden.

`foto INTO v_verwijzing_foto`

Hier wordt de verwijzing naar de FOTO van het opgegeven kantoor (p_kantnr) opgehaald.

`FOR UPDATE`

Om het record te kunnen vullen dienen we deze te 'locken'. Dit gebeurt met de FOR UPDATE clause.

`DBMS_LOB.LOADFROMFILE (v_foto_blob , v_verwijzing_bfile , v_bfilesize);`

De procedure `DBMS_LOB.LOADFROMFILE` kopieert data uit een externe LOB (BFILE) of een gedeelte daarvan naar een BLOB of CLOB kolom. De eerste parameter verwijst naar het doelbestand van de kopieeractie, de tweede parameter naar de bron en de derde geeft aan hoeveel bytes er uit de BFILE gelezen worden.

`DBMS_LOB.FILECLOSE(V_VERWIJZING_BFILE)`

Nadat het record voorzien is van de gegevens zal het bestand van waaruit de gegevens komen ook weer afgesloten dienen te worden.

Als de procedure LAADBLOB in de database is aangemaakt en we voeren vervolgens een update uit op de FOTO kolom van Apple (10) dan zal ditl een foutmelding opleveren!

```
exec laadblob('DIR_VOOR_CURSIST','AppleHQ.gif',10)
ORA-06502: PL/SQL: numeric or value error: invalid LOB locator specified: ORA-22275
```

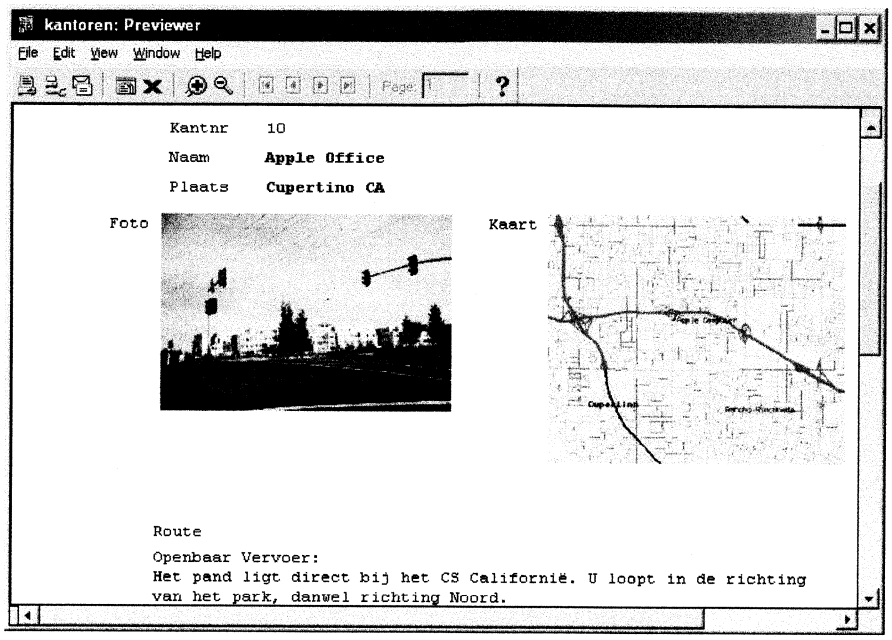
We dienen namelijk eerst de fotokolom te initialiseren met behulp van de functie EMPTY_BLOB():

```
update lob_kantoren set foto = empty_blob()
where kantnr = 10;
```

Daarna kunnen we de procedure LAADBLOB wel uitvoeren:

```
exec laadblob('DIR_VOOR_CURSIST','AppleHQ.gif',10)
```

Het resultaat van de kolommen KAART en FOTO kunnen we controleren in SQL Developer of bijvoorbeeld met een rapport dat is gemaakt in Oracle Reports.



9.10 Toevoegen externe gegevens in Interne CLOBs

In de tabel LOB_KANTOREN hebben we de beschikking over een LONG kolom met daarin een routebeschrijving. Om eenvoudig met het CLOB datatype te gaan werken kunnen we deze kolom gaan converteren naar een CLOB datatype.

9.10.1 LONG kolom converteren

Voorbeeld:

```
desc LOB_KANTOREN
Name                                                      Null?    Type
--------------------------------------------------------- -------- ---------
KANTNR                                                    NOT NULL NUMBER
NAAM                                                               VARCHAR2(50)
PLAATS                                                             VARCHAR2(50)
ROUTE                                                              LONG
FOTO                                                               BLOB
KAART                                                              BINARY FILE LOB

ALTER TABLE LOB_KANTOREN
MODIFY route clob;

desc LOB_KANTOREN
 Name                                                     Null?    Type
 --------------------------------------------------------- -------- ---------
 KANTNR                                                   NOT NULL NUMBER
 NAAM                                                              VARCHAR2(50)
 PLAATS                                                            VARCHAR2(50)
 ROUTE                                                             CLOB
 FOTO                                                              BLOB
 KAART                                                             BINARY FILE LOB
```

De kolom wordt met ALTER TABLE.. MODIFY van datatype veranderd. Hierbij gelden de volgende regels.

- Een LONG kan worden veranderd in een CLOB of een NCLOB, een LONG RAW kolom kan worden veranderd in een BLOB. Alle gedefinieerde constraints en triggers worden geerfd. Als deze veranderd of verwijderd moeten worden moet u daar vervolgens zelf zorg voor dragen.
- Als er domain indexen bestaan op de long kolom moeten deze voor de conversie verwijderd worden. En na de conversie moeten alle andere indexen op de tabel opnieuw opgebouwd worden (ALTER INDEX index REBUILD).

9.10.2 DBMS_LOB.LOADFROMFILE

Een CLOB kolom wordt op een soortgelijke wijze gevuld als de BLOB kolom. Er zijn wel een paar kleine verschillen op te merken. Het is prettig om bij een CLOB aan te kunnen geven wat de beginpositie en de omvang van de op te halen tekst dient te zijn.
We gaan de procedure DBMS_LOB.LOADFROMFILE eens beter bekijken. Zoals reeds besproken kopieert deze procedure een externe LOB (BFILE) of een gedeelte daarvan naar een interne LOB.

Syntax

```
DBMS_LOB.LOADFROMFILE
      ( DEST_LOB     IN/OUT    BLOB|CLOB
      , SRC_LOB         IN     BFILE
      , AMOUNT          IN     INTEGER
      , DEST_OFFSET     IN     INTEGER:=1
      , SRC_OFFSET      IN     INTEGER:=1 );
```

DEST_LOB	Locator voor het doel CLOB/BLOB object.
SRC_LOB	Bestandslocator voor het externe bronobject.
AMOUNT	Omvang op te halen data van extern object.
DEST_OFFSET	Beginpositie in doel object; CLOB(in char), BLOB(in byte).
SRC_OFFSET	Beginpositie in bron object; CLOB(in char), BLOB(in byte).

We hebben de argumenten DEST_LOB, SRC_LOB en AMOUNT al gebruikt bij het inlezen van de BLOB objecten in de procedure LAADBLOB. Met deze argumenten benoemen we respectievelijk:

- Locator voor het doel CLOB/BLOB object.
- Het LOB object.
- De omvang van de data.

De omvang kunnen we laten bepalen met de functie DBMS_LOB.GETLENGTH() .

Over de argumenten DEST_OFFSET en SRC_OFFSET kunnen we melden dat deze optioneel zijn. Indien deze argumenten niet gebruikt worden, wordt *alles* uit het bronbestand geplaatst op positie 1 van de CLOB kolom. We dienen er rekening mee te houden dat de tekst die ingevoegd wordt bestaande tekst overschrijft.

9.11 Nuttige functies in DBMS_LOB

We hebben tot op heden kennisgemaakt met een aantal essentiële procedures en functies uit de package DBMS_LOB. Het gaat op dit moment te ver om alle daarin aanwezige functies en procedures te bespreken. Wat we wel hebben gedaan is een overzicht maken van de nog niet eerder besproken procedures en functies.

9.11.1 Wijzigen Inhoud LOB

Wijzigen van de inhoud kan alleen bij interne LOBs. BFILE LOBs zijn read-only en derhalve dus alleen maar te wijzigen met een extern programma. Voor het wijzigen van de inhoud van een LOB hebben we de beschikking over een aantal functies. In onderstaand schema een beschrijving.

DBMS_LOB.COMPARE()

- Vergelijken inhoud van twee LOB objecten.

```
(LOBLOC_1    CLOB |BLOB |BFILE IN
 LOBLOC_2    CLOB |BLOB |BFILE IN
 AMOUNT              NUMBER(38)          IN     DEFAULT
 OFFSET_1    NUMBER(38)          IN     DEFAULT
 OFFSET_2    NUMBER(38)          IN     DEFAULT)
```

DBMS_LOB.APPEND()

- Voegt de inhoud van SouRCeLOB toe aan DESTinationLOB.

```
(DESTLOB_LOC    CLOB | BLOB        IN
 SRCLOB_LOC     CLOB | BLOB        IN)
```

DBMS_LOB.COPY()

- Kopieert de inhoud van SouRCeLOB naar DESTinationLOB.

```
(DESTLOB_LOC    CLOB | BLOB        IN
 SRCLOB_ LOC    CLOB | BLOB        IN
 AMOUNT         NUMBER(38)         IN
 DEST_OFFSET    NUMBER(38)         IN     DEFAULT
 SRC_OFFSET     NUMBER(38)         IN     DEFAULT)
```

DBMS_LOB.ERASE()

- Verwijderd data in een LOB vanaf een aangegeven positie met een opgegeven omvang.

```
(LOB_LOC        CLOB|BLOB          IN/OUT
 AMOUNT         NUMBER(38)         IN/OUT
 OFFSET         NUMBER(38)         IN     DEFAULT)
```

DBMS_LOB.TRIM()

- Opschonen overtollige ruimte welke met ERASE is verwijderd.

```
(LOB_LOC        CLOB|BLOB          IN
 NEWLEN         NUMBER(38)         IN)
```

DBMS_LOB.SUBSTR() `RETURNS VARCHAR2(CLOB)|RAW(BLOB/BFILE)`

- Retourneert een deel van de data uit de LOB.

```
(LOB_LOC        CLOB|BLOB|BFILE    IN
 AMOUNT         NUMBER(38)         IN     DEFAULT
 OFFSET         NUMBER(38)         IN     DEFAULT)
```

DBMS_LOB.INSTR() `RETURNS NUMBER`

- Retourneert de startpositie van het opgegeven patroon.

```
(LOB_LOC        CLOB|BLOB|BFILE    IN
 PATTERN        VARCHAR2|RAW       IN
 OFFSET         NUMBER(38)         IN     DEFAULT
 NTH            NUMBER(38)         IN     DEFAULT) 'hoeveelste' voorkomen
```

DBMS_LOB.READ()

- Leest data uit een LOB vanaf een bepaalde positie en omvang.

```
(LOB_LOC        CLOB|BLOB|BFILE   IN/OUT
 AMOUNT         BINARY_INTEGER    IN     DEFAULT
 OFFSET         NUMBER(38)        IN     DEFAULT
 BUFFER         VARCHAR2|RAW      OUT)
```

DBMS_LOB.WRITE()

- Schrijft een bepaalde hoeveelheid data naar een LOB op een bepaalde positie.

```
(LOB_LOC        CLOB|BLOB         IN
 AMOUNT         BINARY_INTEGER    IN     DEFAULT
 OFFSET         NUMBER(38)        IN     DEFAULT
 BUFFER         Varchar2|raw      IN)
```

DBMS_LOB.WRITEAPPEND ()

- Voegt een bepaalde hoeveelheid data toe aan een LOB op een bepaalde positie.

```
(LOB_LOC        CLOB|BLOB         IN/OUT
 AMOUNT         BINARY_INTEGER    IN     DEFAULT
 BUFFER         VARCHAR2|RAW      IN)
```

9.11.2 Controle functies

DBMS_LOB.FILEEXISTS()

- Retourneert 1 of 0 indien bestand wel of niet bestaat.

```
(file_LOC       BFILE             IN)
```

DBMS_LOB.FILEGETNAME()

- Retourneert de *aliasnaam* van de Oracle Directory en de *bestandsnaam*.

```
(FILE_LOC       BFILE             IN
 Dir_ALIAS      VARCHAR 2         OUT
 FILENAME       VARCHAR 2         OUT)
```

DBMS_LOB.FILEOPEN() / DBMS_LOB.FILEISOPEN()

- Bestand openen of bepalen dat bestand geopend is. 1=open, 0=gesloten

```
(FILE_LOC       BFILE             IN)
```

DBMS_LOB.FILECLOSE()

- Sluiten BFILE bestand.

```
(FILE_LOC       BFILE             IN/OUT) (out=alleen voor FILECLOSE)
```

DBMS_LOB.FILECLOSEALL()

- Sluiten van alle geopende BFILE bestand(en).

Appendix A Tabellen

tabel P_WERKNEMERS

beschrijving

```
Name                           Null?    Type
 ------------------------------ -------- ----
 PERSNR                         NOT NULL NUMBER(4)
 NAAM                                    VARCHAR2(20)
 FUNCTIE                                 VARCHAR2(15)
 MGR                                     NUMBER(4)
 SAL                                     NUMBER(5)
 TOESLAG                                 NUMBER(5)
 KANTNR                                  NUMBER(2)
```

inhoud

PERSNR	NAAM	FUNCTIE	MGR	SAL	TOESLAG	KANTNR
3381	SMITS	KLERK	7902	2400		20
3462	ALKEMA	VERKOPER	4621	2600	300	30
3518	WALSTRA	VERKOPER	4621	2250	500	30
3930	PIETERS	MANAGER	6221	3975		20
4510	VERGEER	VERKOPER	4621	2250	1400	30
4621	KLAASEN	MANAGER	6221	3850		30
5810	HEUVEL	MANAGER	6221	3450		10
5931	SANDERS	ANALIST	3930	4000		20
6221	KRAAY	DIRECTEUR		6000		10
6500	DROST	VERKOPER	4621	2500	0	30
6681	ADELAAR	KLERK	5931	2100		20
7900	APPEL	KLERK	4621	1950		30
7902	VERMEULEN	ANALIST	3930	3900		20
8222	MANDERS	KLERK	5810	2300		10

tabel P_KANTOREN

beschrijving

```
Name                           Null?    Type
------------------------------ -------- ----
KANTNR                         NOT NULL NUMBER(4)
NAAM                                    VARCHAR2(25)
PLAATS                                  VARCHAR2(20)
```

inhoud

KANTNR	NAAM	PLAATS
10	BOEKHOUDING	AMSTERDAM
20	ONDERZOEK	UTRECHT
30	VERKOOP	DEN HAAG
40	PRODUCTIE	ARNHEM

tabel P_PERSONEN

beschrijving

```
Name                           Null?    Type
------------------------------ -------- ----
NR                             NOT NULL NUMBER(4)
VOORLETTERS                             VARCHAR2(10)
NAAM                                    VARCHAR2(20)
GESLACHT                                CHAR(1)
DOCUMENTATIE                            CHAR(1)
STRAAT                                  VARCHAR2(20)
HUISNUMMER                              NUMBER(4)
POSTCODE                                VARCHAR2(6)
PLAATS                                  VARCHAR2(20)
```

Appendix A Tabellen

inhoud

NR	VOORLETTER	NAAM	G	D	STRAAT	HUISNUMMER	POSTCO	PLAATS
1	T	Bloem	M	A	Kastanjelaan	23	3511NM	UTRECHT
2	M.	Appel	V	N	Middellaan	37	5611AM	EINDHOVEN
6	P.T.W.	Reker	M	O	Keizersgracht	565	1041AC	AMSTERDAM
8	P.	Meer	V	O	Coolsingel	312	2083BD	ROTTERDAM
9	J.	Pas	M	O	Oranjesingel	32	6511NM	NIJMEGEN
10	K.	Kraayenhof	M	A	Steenstraat	77	6809TG	ARNHEM
17	S.B.	Smeets	V	N	Spui	87	2567XX	DEN HAAG
19	J.	Brink	M	O	Boterdiep	6	9711LB	GRONINGEN

tabel P_CURSUSSEN

beschrijving

Name	Null?	Type
NR	NOT NULL	NUMBER(2)
NAAM		VARCHAR2(50)
AANT_DAG		NUMBER(2)
PRIJS		NUMBER(7,2)

inhoud

NR	NAAM	AANT_DAG	PRIJS
1	SQL 5 dagen	5	2250
2	PL/SQL	2	990
3	Oracle Forms	6	2970
4	Oracle Reports	4	1980
5	Developer casus	2	950
6	Oracle Eindgebruiker	1	495
7	Oracle Database Administrator	10	4500
8	Oracle Applicatiebouwer	20	9000

tabel P_BOEKINGEN

beschrijving

Name	Null?	Type
PER_NR	NOT NULL	NUMBER(4)
CUR_NR		NUMBER(2)
DATUM	NOT NULL	DATE

inhoud

PER_NR	CUR_NR	DATUM
1	1	06-MAR-99
1	1	07-MAR-99
1	1	08-MAR-99
1	1	09-MAR-99
1	1	10-MAR-99
2	2	13-MAR-99
2	2	14-MAR-99
10	6	03-APR-99
17	6	21-APR-99
8	2	06-FEB-99
8	2	13-FEB-99
1	2	06-FEB-99
1	2	07-FEB-99
2	1	13-FEB-99
2	1	14-FEB-99
2	1	15-FEB-99
2	1	16-FEB-99
2	1	17-FEB-99
6	6	03-MAY-99
19	3	03-MAY-99
19	3	04-MAY-99
19	3	05-MAY-99
19	3	07-MAY-99
19	3	08-MAY-99
8	4	25-APR-99
8	4	26-APR-99
8	4	27-APR-99
8	4	28-APR-99
1	5	01-JUN-99
1	5	02-JUN-99

Appendix A Tabellen

tabel P_SPELERS

beschrijving

```
Name                           Null?    Type
------------------------------ -------- ----
SPELNR                         NOT NULL NUMBER(3)
NAAM                                    VARCHAR2(20)
VOORL                                   VARCHAR2(3)
JAARTOE                                 VARCHAR2(4)
PLAATS                                  VARCHAR2(10)
BONDSNR                                 NUMBER(4)
```

inhoud

```
       SPELNR NAAM                 VOO JAAR PLAATS       BONDSNR
--------- -------------------- --- ---- ---------- ---------
        6 Honing               R   1977 Den Haag         8467
       44 Bakker               E   1980 Rijswijk         1124
       83 Martens              PK  1982 Utrecht          1608
        2 van der Wal          R   1975 Den Haag         2411
       27 Cools                DD  1983 Utrecht          2513
      104 Kok                  D   1984 Utrecht          7060
        7 Sorgdrager           GWS 1981 Den Haag
       57 O'Neal               M   1985 Den Haag         6409
      112 Winterdijk           IP  1984 Rotterdam        1319
        8 Cools                C   1980 Rijswijk         2983
```

tabel P_TEAMS

beschrijving

```
Name                           Null?    Type
------------------------------ -------- ----
TEAMNR                         NOT NULL NUMBER(2)
SPELNR                         NOT NULL NUMBER(3)
DIVISIE                        NOT NULL VARCHAR2(6)
```

inhoud

```
    TEAMNR     SPELNR DIVISI
---------- ---------- ------
         1          2 ere
         2         27 tweede
```

tabel P_BOETEBEDRAGEN

beschrijving

```
Name                           Null?    Type
------------------------------ -------- ----
NR                             NOT NULL NUMBER(2)
SPELNR                         NOT NULL NUMBER(3)
DATUM                                   DATE
BEDRAG                                  NUMBER(7,2)
```

inhoud

```
      NR    SPELNR DATUM        BEDRAG
-------- --------- --------- ---------
       1         6 07-JAN-90       200
       2        44 17-FEB-91       100
       3        27 28-NOV-93       100
       4       104 19-JUN-94        50
       5        44 07-JAN-90       200
       6         8 07-JAN-90       200
       7        44 18-OCT-92        30
       8        27 08-JAN-95        75
```

Appendix A Tabellen

tabel P_WEDSTRIJDEN

beschrijving

```
Name                           Null?    Type
------------------------------ -------- ----
TEAMNR                         NOT NULL NUMBER(2)
SPELNR                         NOT NULL NUMBER(3)
GEWONNEN                       NOT NULL NUMBER(3)
VERLOREN                       NOT NULL NUMBER(3)
```

inhoud

TEAMNR	SPELNR	GEWONNEN	VERLOREN
1	6	9	1
1	44	7	5
1	83	3	3
1	2	4	8
1	57	5	0
1	8	0	1
2	27	11	2
2	104	8	4
2	112	4	8
2	8	4	4

tabel P_PATIENTEN

beschrijving

```
Name                           Null?    Type
------------------------------ -------- ----
PATNR                          NOT NULL NUMBER(5)
NAAM                                    VARCHAR2(15)
PLAATS                                  VARCHAR2(15)
GEBDAT                                  DATE
MV                                      VARCHAR2(1)
ZIEKFNR                                 NUMBER(7)
```

inhoud

PATNR	NAAM	PLAATS	GEBDAT	M	ZIEKFNR
11321	Koopmans M.	Utrecht	11-DEC-66	M	3542764
12816	Schouten W.	Den Haag	23-APR-73	V	7466384
19381	Elbers M.	Amsterdam	01-JAN-76	V	9753728
25218	Dekker B.	Utrecht	05-NOV-54	M	8466355
30940	Lammers T.	Arnhem	12-APR-43	V	3452718
38911	Jong H.	Nijmegen	12-JAN-82	M	4656238
39410	Manders G.	Den Bosch	11-DEC-70	M	2794710
45630	Ravenhorst P.	Eindhoven	04-FEB-48	M	9872513
48220	Feenstra A.	Breda	27-FEB-77	V	3529976
50333	Horst E.	Utrecht	12-APR-64	M	1232988

tabel P_ZIEKENHUIZEN

beschrijving

```
Name                           Null?    Type
------------------------------ -------- ----
ZIEKHNR                        NOT NULL NUMBER(4)
NAAM                                    VARCHAR2(15)
PLAATS                                  VARCHAR2(15)
TELEFOON                                VARCHAR2(12)
TOTBED                                  NUMBER(4)
```

inhoud

ZIEKHNR	NAAM	PLAATS	TELEFOON	TOTBED
10	AMC	Amsterdam	020-6532617	502
15	Diaconessen	Utrecht	030-2646362	587
20	Antonius	Nieuwegein	030-6045632	412
25	Zuiderzee	Lelystad	0320-255522	845

Appendix A Tabellen

tabel P_AFDELINGEN

beschrijving

```
Name                           Null?    Type
------------------------------ -------- ----
ZIEKHNR                        NOT NULL NUMBER(4)
AFDNR                          NOT NULL NUMBER(4)
NAAM                                    VARCHAR2(15)
TOTBED                                  NUMBER(4)
```

inhoud

ZIEKHNR	AFDNR	NAAM	TOTBED
10	3	Intensive Care	21
10	6	Psychiatrie	67
15	3	Intensive Care	10
15	4	Hartafdeling	53
20	1	Hersteloord	10
20	6	Psychiatrie	118
20	2	Kinder	34
25	4	Hartafdeling	55
15	1	Hersteloord	13
25	2	Kinder	24

tabel P_STAFLEDEN

beschrijving

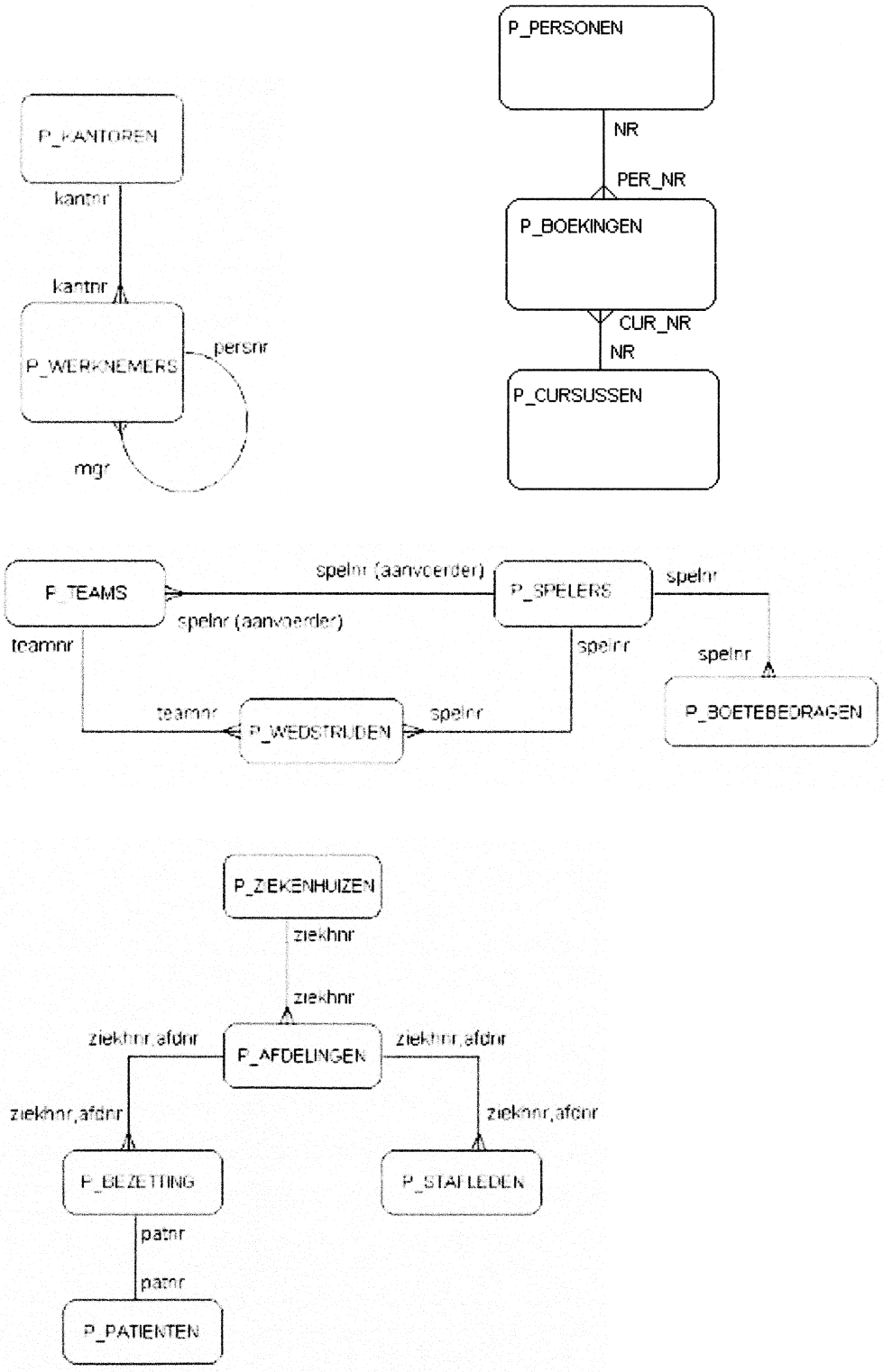
```
Name                           Null?    Type
------------------------------ -------- ----
ZIEKHNR                        NOT NULL NUMBER(2)
AFDNR                          NOT NULL NUMBER(2)
PERSNR                         NOT NULL NUMBER(4)
NAAM                                    VARCHAR2(15)
FUNCTIE                                 VARCHAR2(15)
DIENST                                  VARCHAR2(6)
SALARIS                                 NUMBER(5)
```

inhoud

ZIEKHNR	AFDNR	PERSNR	NAAM	FUNCTIE	DIENST	SALARIS
10	6	3526	Dinter B.	Verpleegster	A	17400
10	6	3198	Hursman J.	Zaalknecht	A	13500
15	4	2342	Keyzer W.	Assistent	A	18300
20	6	2315	Horst D.	Verpleegster	N	18300
20	6	8574	Beek G.	Zaalknecht	N	12600
20	2	3257	Mensink C.	Assistent	N	17000
20	1	8632	Riksen G.	Verpleegster	D	20200
20	1	5342	Coolen R.	Verpleegster	A	16300
25	4	6543	Arends R.	Assistent	D	17000
25	2	9835	Fleskes H.	Verpleegster	A	19400

Appendix A Tabellen

Appendix B Datamodel



Appendix B Datamodel

Appendix C
Repeat intervallen DBMS_SCHEDULER

Syntax: voor REPEAT_INTERVAL:
NB * betekent 0 of meer keer. Kommas en punt-kommas (uiteraard) letterlijk overnemen.

```
REPEAT_INTERVAL = frequency_clause
[; interval_clause] [; bymonth_clause] [; byweekno_clause]
[; byyearday_clause] [; bymonthday_clause] [; byday_clause]
[; byhour_clause] [; byminute_clause] [; bysecond_clause]

frequency_clause =      FREQ = frequency
frequency =             YEARLY | MONTHLY | WEEKLY | DAILY | HOURLY | MINUTELY |
                        SECONDLY

interval_clause =       INTERVAL = intervalnum
intervalnum =           1 through 99

bymonth_clause =        BYMONTH = monthlist
monthlist =             monthday ( , monthday) *
month =                 numeric_month | char_month
numeric_month =         1 | 2 | 3 ... 12
char_month =            JAN | FEB | MAR | APR | MAY | JUN | JUL | AUG | SEP | OCT |
                        NOV | DEC

byweekno_clause =       BYWEEKNO = weeknumber_list
weeknumber_list =       weekday (, weeknumber)*
week =                  [minus] weekno
minus =                 -
weekno =                1 through 53

byyearday_clause =      BYYEARDAY = yearday_list
yearday_list =          yearday (, yearday)*
yearday =               [minus] yeardaynum
yeardaynum =            1 through 366

bymonthday_clause =     BYMONTHDAY = monthday_list
monthday_list =         monthday (, monthday) *
monthday =              [minus] monthdaynum
monthdaynum =           1 through 31

byday_clause =          BYDAY = byday_list
byday_list =            byday (, byday)*
byday =                 [weekdaynum] day
weekdaynum =            [minus] daynum
daynum =                1 through 53 /* if frequency is yearly */
daynum =                1 through 5 /* if frequency is monthly */
day =                   MON | TUE | WED | THU | FRI | SAT | SUN

byhour_clause =         BYHOUR = hour_list
hour_list =             hour (, hour)*
hour =                  0 through 23

byminute_clause =       BYMINUTE = minute_list
minute_list =           minute (, minute)*
minute =                0 through 59

bysecond_clause =       BYSECOND = second_list
second_list =           second (, second)*
second =                0 through 59
```

Appendix C Repeat intervallen DBMS_SCHEDULER

Appendix D Opdrachten en uitwerkingen

Opdrachten hoofdstuk 2

Maak de opdrachten met behulp van SQL Developer. Geef bij de opdrachten de commandoscriptjes een naam waaruit blijkt bij welke opgave van welk hoofdstuk ze horen (Bijvoorbeeld: het script dat hoort bij opdracht 1 van hoofdstuk 2 heeft de naam PE21.SQL, bij opdracht 2 van hoofdstuk 2 hoort het script met de naam PE22.SQL, enz.).
Controleer van alle procedures en functies de werking door ze uit te proberen.

1. Maak een procedure VOEGTOE_CURSUS waarmee rijen toegevoegd kunnen worden aan de tabel P_CURSUSSEN.
   - Het nummer van de cursus moet automatisch gegenereerd worden met behulp van een cursor.
   - Maak gebruik van EXCEPTIONS om de volgende voorwaarden af te dwingen en zet een melding in HULPTABEL:
     - Een cursus die toegevoegd wordt, mag nooit meer dan 25 dagen duren.
     - De kolom NAAM mag geen dubbele waarden bevatten.
2. Maak een functie AANTAL_DAGEN, met behulp waarvan het aantal cursusdagen van een cursist bepaald kan worden. Gebruik hiervoor de tabellen P_PERSONEN en P_BOEKINGEN.
   - De gebruiker geeft de naam van de cursist op.
   - Een cursist kan meerdere cursussen volgen.
   - Als de naam van de opgegeven cursist niet bestaat moet de functie NULL teruggeven.

   Optioneel:
   - gebruik ook de tabel P_CURSUSSEN en bepaal niet het aantal dagen dat iemand komt, maar het totaal aantal dagen dat iemand op cursus mag komen.
3. Maak een procedure CURSISTEN_OVERZICHT die met behulp van de functie AANTAL_DAGEN voor alle cursisten uit de tabel P_PERSONEN, het aantal dagen en de naam in HULPTABEL zet
   - Als een persoon geen cursus volgt moet er een melding in HULPTABEL worden gezet.
4. Maak een operator MAAL die de volgende uitvoer geeft:
   - maal(2,3) => 6
   - maal(3,'a') => 'aaa'
   - maal('cursist',2) => 'cursistcursist'
   - maal('a','b') => Een foutmelding met de tekst: 'Dit kan niet'

Meer opdrachten op de volgende pagina.

Optionele opdrachten:

5. Laat de relatie tussen CURSISTEN_OVERZICHT en AANTAL_DAGEN zien met behulp van DEPTREE_FILL.
6. Laat zien welke tabellen door AANTAL_DAGEN benaderd worden met behulp van USER_DEPENDENCIES.

Appendix D Opdrachten en Uitwerkingen

Appendix D Opdrachten en Uitwerkingen

Opdrachten hoofdstuk 3

Maak de opdrachten met behulp van SQL Developer. Geef bij de opdrachten de commandoscriptjes een naam waaruit blijkt bij welke opgave van welk hoofdstuk ze horen (Bijvoorbeeld: het script dat hoort bij opdracht 1 van hoofdstuk 3 heeft de naam PE31.SQL, bij opdracht 2 van hoofdstuk 3 hoort het script met de naam PE32.SQL, enz.).

1. Definieer de package OPLEIDING (specificatie en body) met daarin de in hoofdstuk 2 gemaakte procedures en functie:
   - VOEGTOE_CURSUS procedure (pe21.sql)
   - AANTAL_DAGEN functie (pe22.sql)
   - CURSISTEN_OVERZICHT procedure (pe23.sql)

   Zorg ervoor dat procedures en functie ook buiten de package gebruikt kunnen worden.
2. Voeg in de packagespecificatie de constante MAX_AANTAL_DAGEN toe, met als initialisatie waarde 25. Deze constante wordt gebruikt binnen de procedure VOEGTOE_CURSUS om het op dat moment geldende maximum aantal dagen op te vragen.
3. Maak een package PACK_VOLGNR met daarin de procedure VOLGNR. Iedere keer dat de procedure VOLGNR wordt aangeroepen, moet er een nieuw volgnummer in HULPTABEL geplaatst worden. Maak hierbij gebruik van een globale variabele en NIET van een sequence.
   Naast het volgnummer moet in HULPTABEL de tijd worden gezet, waarop het nieuwe volgnummer gegenereerd is. Nadat de procedure VOLGNR 3 keer is uitgevoerd, moet het volgende in de hulptabel staan (de datum kan uiteraard afwijken):

```
   KOLOM1 KOLOM2                         KOLOM3
--------- ------------------------------ ------------------------------
        1 01 january 2006 09:47:27
        2 01 january 2006 09:47:35
        3 01 january 2006 09:47:39
```

Optionele opgave:

4. De procedure CURSISTEN_OVERZICHT moet een nieuwe procedure OVERZICHT aanroepen, dit is binnen de package OPLEIDING een lokale procedure.

   Als CURSISTEN_OVERZICHT wordt uitgevoerd zonder dat er een parameter wordt opgegeven, dan moet een overzicht gemaakt worden van alle cursisten van de tabel P_PERSONEN. Hierbij moet gebruik gemaakt worden van de nieuwe procedure OVERZICHT. Het overzicht moet in HULPTABEL worden geplaatst, bij personen die geen cursisten zijn moet een melding in HULPTABEL komen.

   Wordt bij het uitvoeren CURSISTEN_OVERZICHT wel een parameter opgegeven dan moet door de procedure OVERZICHT alleen de gegevens worden opgehaald van de genoemde cursist.

Deze opdracht gaat verder op de volgende pagina.

Als de betreffende naam niet bestaat moet een melding op het **scherm** worden gegeven.
- Gebruik een default value voor de parameter in procedure CURSISTEN_OVERZICHT.
- Maak bij de procedure OVERZICHT gebruik van overloading.

Controleer de werking van de procedures uit de package.

Appendix D Opdrachten en Uitwerkingen

Appendix D Opdrachten en Uitwerkingen

Opdrachten hoofdstuk 4

Maak de opdrachten met behulp van SQL Developer. Geef bij de opdrachten de commandoscriptjes een naam waaruit blijkt bij welke opgave van welk hoofdstuk ze horen (Bijvoorbeeld: het script dat hoort bij opdracht 1 van hoofdstuk 4 heeft de naam PE41.SQL).

1. Maak een procedure die op het scherm laat zien hoeveel rijen een op te geven tabel bevat. Zorg ervoor dat op het scherm bijvoorbeeld de volgende boodschap wordt getoond:

   ```
   De tabel P_KANTOREN bevat 4 rijen.
   ```

   a. Doe dit met behulp van DBMS_SQL, noem de procedure DBMS_TEL_RIJ.
   b. Doe dit met behulp van Native Dynamic SQL, noem de procedure NDS_TEL_RIJ.
2. Maak een procedure TABEL_KOLOM die een tabel GEGEVENS aanmaakt en daarin de waarden uit een opgegeven tabel en kolom plaatst. Voordat dit gebeurt moet door de procedure eerst worden nagegaan of de tabel GEGEVENS al bestaat, is dat het geval dan moet de tabel worden verwijderd.

   Hints:
   - Maak bij deze opgave gebruik van EXEC_DDL_STATEMENT en niet van DBMS_SQL.
   - Met behulp van de view USER_TABLES kunt u controleren of een tabel reeds bestaat.

Optionele opgave:

3. Jaarlijks krijgen werknemers een bepaald percentage van hun salaris aan salarisverhoging. Dit percentage is niet ieder jaar hetzelfde. Schrijf een procedure PE43 waarmee de salarissen in de tabel P_WERKNEMERS gewijzigd kunnen worden en de variabele verhoging (een percentage) opgegeven kan worden. Zorg ervoor dat bij het uitvoeren van deze procedure de nieuwe salarissen als volgt op het scherm worden getoond.

   ```
   SQL> exec pe43(2)
   SMITS 7200
   ALKEMA 7800
   WALSTRA 6750
   PIETERS 11925
   VERGEER 6750
   KLAASEN 11550
   HEUVEL 10350
   SANDERS 12000
   KRAAY 18000
   DROST 7500
   ADELAAR 6300
   APPEL 5850
   VERMEULEN 11700
   MANDERS 6900
   ```

   Test de procedure en maak vervolgens de wijzigingen in de tabel P_WERKNEMERS weer ongedaan.

Appendix D Opdrachten en Uitwerkingen

Appendix D Opdrachten en Uitwerkingen

Opdrachten hoofdstuk 5

Maak de opdrachten met behulp van SQL Developer. Geef bij de opdrachten de commandoscriptjes een naam waaruit blijkt bij welke opgave van welk hoofdstuk ze horen (Bijvoorbeeld: het script dat hoort bij opdracht 1 van hoofdstuk 5 heeft de naam PE51.SQL).

1. Maak een schedule PERMINUUT aan met een interval van een minuut, een begintijd die gelijk is aan de huidige tijd en zonder eindtijd.
2. Maak een programma VOLGNRPROGRAM die de procedure VOLGNR uit de package PACK_VOLGNR uitvoert. Deze procedure heeft u in hoofdstuk 3 aangemaakt en zet een volgnummer in HULPTABEL.
3. Maak een job JOB_VOLGNR aan op basis van het programma VOLGNRPROGRAM en de schedule PERMINUUT.
4. Controleer of de job daadwerkelijk wordt uitgevoerd. Laat tevens door middel van een SQL- statement zien welke actie de job uitvoert, wanneer de volgende keer is dat de job wordt uitgevoerd en wat het herhalingsinterval is.
5. Verwijder de aangemaakte job, schedule en programma **nadat** uw docent de opgaven heeft nagekeken.

Optionele opgaven:

6. Maak met behulp van de package DBMS_JOB een job die elke twee minuten een rij aan HULPTABEL toevoegt met daarin de tijd en de tekst "DBMS_JOB". Controleer of de job daadwerkelijk wordt uitgevoerd.
    1. Laat door middel van een SQL statement zien welke statement de job uitvoert en wanneer de volgende keer is dat de job wordt uitgevoerd.
    2. Verwijder de job **nadat** uw docent de opgaven heeft nagekeken.

Appendix D Opdrachten en Uitwerkingen

Appendix D Opdrachten en Uitwerkingen

Opdrachten hoofdstuk 6

Maak de opdrachten met behulp van SQL Developer. Geef bij de opdrachten de commandoscriptjes een naam waaruit blijkt bij welke opgave van welk hoofdstuk ze horen (Bijvoorbeeld: het script dat hoort bij opdracht 1 van hoofdstuk 6 heeft de naam PE61.SQL).

1. Maak een procedure RANDOM_GETAL die een random getal in HULPTABEL zet samen met de tijd waarop de procedure wordt uitgevoerd.
2. In een trigger kunt u PL/SQL-code plaatsen die uitgevoerd wordt wanneer een bepaalde gebeurtenis plaatsvindt. In het **volgende hoofdstuk** zult u een trigger maken die iedere keer dat de tabel P_KANTOREN wordt aangepast met behulp van procedures en functies uit de package DBMS_PIPE een boodschap in de verbinding 'contr_kan_<gebruikersnaam>' plaatst. Wanneer de tabel P_KANTOREN bijvoorbeeld 2 keer wordt aangepast, zullen er door de trigger dus 2 boodschappen in de verbinding worden gezet.

   In deze opgave gaat u de procedure CONTROLEER_KANTOOR maken die de boodschappen uit de verbinding 'contr_kan_<gebruikersnaam>' ophaalt. Wanneer er bijvoorbeeld 2 boodschappen worden opgehaald door deze procedure, moet op het scherm de boodschap 'De tabel is 2 keer aangepast' verschijnen. Wanneer er geen boodschappen in de verbinding aanwezig zijn, moet de boodschap 'Geen aanpassing' verschijnen.

   Maak de procedure CONTROLEER_KANTOOR aan. Om de procedure te testen kunt u even "handmatig" (via een PL/SQL blok) alvast wat boodschappen in de verbinding plaatsen. Bewaar deze code, zodat u deze kunt hergebruiken wanneer u in het volgende hoofdstuk de trigger gaat maken.

Optionele opgave:

3. Maak een procedure TOON_WERKNEMERS die de kolommen PERSNR en NAAM uit de tabel P_WERKNEMERS op de volgende manier op het scherm toont.

```
3381  SMITS          3462  ALKEMA         3518  WALSTRA
3930  PIETERS        4510  VERGEER        4621  KLAASEN
5810  HEUVEL         5931  SANDERS        6221  KRAAY
6500  DROST          6681  ADELAAR        7900  APPEL
7902  VERMEULEN      8222  MANDERS
```

Appendix D Opdrachten en Uitwerkingen

Appendix D Opdrachten en Uitwerkingen

Opdrachten hoofdstuk 7

Maak de opdrachten met behulp van SQL Developer. Geef bij de opdrachten de commandoscriptjes een naam waaruit blijkt bij welke opgave van welk hoofdstuk ze horen (Bijvoorbeeld: het script dat hoort bij opdracht 1 van hoofdstuk 7 heeft de naam PE71.SQL).

1. Zorg ervoor dat het aantal dagen dat een cursus duurt niet gewijzigd kan worden. Geef een duidelijke melding.
2. Nieuwe cursussen moeten minimaal 2 dagen duren. Verander het aantal dagen zonodig in 2, en geef een melding dat het aantal dagen is veranderd. Zorg er tevens voor dat bij nieuwe cursussen de cursusprijs altijd het aantal dagen * 495 euro is. Geef een melding als de prijs is aangepast.
3.
   a Maak een tabel SALWIJZIGING met volgende kolommen:
   OUD_SAL numeriek
   NIEUW_SAL numeriek
   WERKNEMER alfanumeriek 20 karakters
   GEBRUIKER alfanumeriek 20 karakters
   DATUM datum.
   b Maak een trigger WER_.... op de tabel P_WERKNEMERS die ervoor zorgt dat bij een salariswijziging een record wordt toegevoegd aan de tabel SALWIJZIGING. In dit record moet de volgende informatie worden opgeslagen: het oude salaris, het nieuwe salaris, de naam van de werknemer, de gebruikersnaam van degene die de wijziging doorvoert en de datum waarop dit gebeurt. De trigger moet alleen af gaan als het salaris echt gewijzigd is.
4. U gaat de trigger T_KANTOOR aanmaken die afgaat wanneer waarden in de tabel P_KANTOREN worden aangepast. Deze trigger moet in de verbinding contr_kan_<gebruikersnaam> de boodschap 'ATTENTIE' plaatsen. Deze boodschap wordt uitgelezen in de procedure CONTROLEER_KANTOOR die al gedeeltelijk in het vorige hoofdstuk is aangemaakt.
   a Maak de trigger T_KANTOOR aan die afgaat na iedere update op de tabel P_KANTOREN en de boodschap 'ATTENTIE' in de verbinding contr_kan_<gebruikersnaam> zet.
   b Pas de procedure CONTROLEER_KANTOOR uit het vorige hoofdstuk aan, zodat de boodschap in de verbinding contr_kan_<gebruikersnaam> uitgelezen wordt en er alleen geteld wordt als de boodschap in de pipe ATTENTIE is.
   c Voer nu de volgende updates uit op de tabel P_KANTOREN zonder deze via een COMMIT vast te leggen. De trigger T_KANTOOR zal nu 2 keer afgaan.

```
UPDATE p_kantoren SET naam = upper(naam) WHERE kantnr=10;
UPDATE p_kantoren SET naam = upper(naam) WHERE kantnr=20;
```

Meer opdrachten op de volgende pagina.

d Open een tweede sessie in SQL Developer en log aan onder uw eigen gebruikersnaam. Controleer of de inhoud van de tabel P_KANTOREN in deze sessie ongewijzigd is en zet de serveroutput aan. Voer vervolgens de procedure CONTROLEER_KANTOOR uit om te bekijken of en hoe vaak de tabel P_KANTOREN is aangepast in de andere sessie.

5. Maak een autonome trigger BOETE_... aan.
Deze moet afgaan wanneer aan de tabel P_BOETEBEDRAGEN een rij wordt toegevoegd. De nieuwe rijen moeten ook aan de schaduwtabel BOETEBEDRAGEN_LOG worden toegevoegd. Dit moet onafhankelijk zijn van het feit of ze daadwerkelijk zijn toegevoegd aan P_BOETEBEDRAGEN of via een rollback zijn teruggedraaid.

   Controleer de werking van de trigger door een aantal rijen toe te voegen aan de tabel P_BOETEBEDRAGEN en sommige daarvan vast te leggen en andere terug te draaien. De tabel BOETEBEDRAGEN_LOG kunt u als volgt aanmaken

```
SQL> create table boetebedragen_log as
  2  select * from p_boetebedragen
  3  where 1=0;

Table created.
```

Optionele opgave:

6. Implementeer de volgende bedrijfsregels:
   a De verhoging van een salaris mag niet hoger zijn dan 5%. Geef een duidelijke melding waarom het salaris niet kan worden verhoogd.
   b Het salaris van een werknemer mag maximaal 30% meer zijn dan het gemiddelde salaris van alle werknemers. Geef een duidelijke melding als de wijziging niet kan worden doorgevoerd. Laat in deze melding het maximaal toegestane salaris zien.
   c Voor de directeur geldt een uitzondering. Die mag wel meer gaan verdienen dan 130% van het gemiddelde salaris.

Appendix D Opdrachten en Uitwerkingen

Opdrachten hoofdstuk 8

Maak de opdrachten met behulp van SQL Developer. Geef bij de opdrachten de commandoscriptjes een naam waaruit blijkt bij welke opgave van welk hoofdstuk ze horen (Bijvoorbeeld: het script dat hoort bij opdracht 1 van hoofdstuk 8 heeft de naam PE81.SQL).

1. Maak een tabel VERWIJDERD, met vier kolommen, ACTIE, GEBRUIKER, OBJECT en TIJD.
2. Maak een trigger OBJ_VERWIJDERD, die voor ieder object dat uit uw eigen schema wordt verwijderd, een rij met de betreffende gegevens toevoegt aan de tabel VERWIJDERD.
3. Maak een tabel aan die u vervolgens weer weggooit, Controleer of er een rij wordt toegevoegd aan de tabel VERWIJDERD.
4. Maak een tweede trigger NIET_VERWIJDERD, die ervoor zorgt dat ook wordt bijgehouden wanneer geprobeerd wordt een niet bestaande tabel te verwijderen. Het is voldoende om de fout 'ORA-00942: table or view does not exists' af te vangen. De betreffende informatie moet in de tabel VERWIJDERD komen te staan.
5. Geef een DROP TABLE statement op een niet bestaande tabel. Controleer of er een rij wordt toegevoegd aan de tabel VERWIJDERD. Welke waarde ontbreekt er?
6. Probeer de tabel VERWIJDERD weg te gooien. Wat gebeurt er en waarom?
7. Verwijder de trigger OBJ_VERWIJDERD en NIET_VERWIJDERD, nadat de coach de opgaven heeft nagekeken.

Optionele opgave:

8. Maak een trigger BOETE_.... aan die afgaat wanneer de tabel P_BOETEBEDRAGEN verwijderd wordt. De trigger moet ervoor zorgen dat automatisch de schaduwtabel BOETEBEDRAGEN_LOG verwijderd wordt.

Appendix D Opdrachten en Uitwerkingen

Appendix D Opdrachten en Uitwerkingen

Opdrachten hoofdstuk 9

Maak de opdrachten met behulp van SQL Developer. Geef bij de opdrachten de commandoscriptjes een naam waaruit blijkt bij welke opgave van welk hoofdstuk ze horen (Bijvoorbeeld: het script dat hoort bij opdracht 1 van hoofdstuk 9 heeft de naam PE91.SQL).

1. Herschrijf de procedure `laadblob` en zorg dat deze, voor het vullen van de BLOB kolom met behulp van het BFILE object, in plaats van het SELECT INTO statement gebruik gaat maken van een *cursor*.

   Voeg vervolgens voor kantoor 30 het HP Office, de foto van het kantoor en de plattegrond toe. Controleer dit in SQL Developer met behulp van het Edit Value window.

2. In de Oracle Directory `DIR_VOOR_CURSIST` staat een tekst bestand: FROMFILE.txt. Open dit bestand en zoek daarin, op ongeveer de 10e regel het, in hoofdletters getypte, woord BFILE. Vanaf dit punt willen we de tekst tot aan het woordje SYNTAX selecteren en toevoegen aan de in het hoofdstuk gemaakte CLOB kolom in de tabel LOB_KANTOREN. Voeg de inhoud van dit tekstbestand vanaf positie 811(= 'BFILE') met een omvang van 364 (=tot aan 'SYNTAX') tekens, toe aan de CLOB kolom op positie 1 voor kantoor 30.

   Een paar aanwijzingen:

```
create or replace procedure leesclob( oradir in varchar2
                        , clobbestand  in varchar2
                        , refwaarde    in varchar2
                        , omvang       in integer
                        , doelpositie  in integer:=1
                        , bronpositie  in integer:=1) is
```

   Voorbeeld procedure aanroep

```
exec leesclob('DIR_VOOR_CURSIST','fromfile.txt',30,364,1,811)
```

   Waarvoor geldt:

`'DIR_VOOR_CURSIST'`	De Oracle directory.
`'fromfile.txt'`	Het betreffende bestand.
`30`	kenmerk van de kolom kantnr in lob_kantoren.
`364`	aantal karakters uit FromFile.txt.
`1`	startpositie in doel-LOB.
`811`	startpositie in FromFile.txt.

Appendix D Opdrachten en Uitwerkingen

UITWERKINGEN

PL/SQL, de procedurele extensie

De uitwerkingen ontvangt u van de coach,
na het afronden van deze cursus.

Uitwerkingen opgaven

PL/SQL, de procedurele extensie

2. Procedures en functies

1. Maak een procedure VOEGTOE_CURSUS waarmee rijen toegevoegd kunnen worden aan de tabel P_CURSUSSEN.
   - Het nummer van de cursus moet automatisch gegenereerd worden met behulp van een cursor.
     Maak gebruik van EXCEPTIONS om de volgende voorwaarden af te dwingen en zet een melding in de hulptabel.
   - Een cursus die toegevoegd wordt, nooit meer dan 25 dagen duren.
   - De kolom NAAM mag geen dubbele waarden bevatten.

```
create or replace procedure voegtoe_cursus (p_cursusnaam varchar2
                                           , p_aant_dag number
                                           , p_prijs number)
is
  cursor c_nieuw_nr is
    select max(nr)+1
    from p_cursussen;
  cursor c_cur (b_naam varchar2) is
    select 'x'
    from p_cursussen
    where naam=b_naam;
  v_cursusnr number;
  v_controle varchar2(1);
  e_te_lang exception;
  e_dubbel  exception;
begin
  if p_aant_dag > 25 then
    raise e_te_lang;
  else
    open c_cur(p_cursusnaam);
    fetch c_cur into v_controle;
    if c_cur%found then
      close c_cur;
      raise e_dubbel;
    else
      open c_nieuw_nr;
      fetch c_nieuw_nr into v_cursusnr;
      close c_nieuw_nr;
    end if;
    close c_cur;
    insert into p_cursussen values (v_cursusnr, p_cursusnaam, p_aant_dag, p_prijs);
  end if;
exception
  when e_te_lang then
     insert into hulptabel values (null, 'De cursus duurt te lang', null);
  when e_dubbel then
    insert into hulptabel values (null, p_cursusnaam,'Deze cursus komt al voor');
end;
/
```

2. Procedures en functies

2. Maak een functie AANTAL_DAGEN, met behulp waarvan het totaal aantal cursusdagen van een cursist bepaald kan worden. Gebruik hiervoor de tabellen P_PERSONEN, P_BOEKINGEN en P_CURSUSSEN.

- De gebruiker geeft de naam van de cursist op.
- Een cursist kan meerdere cursussen volgen.
- Als de naam van de opgegeven cursist niet bestaat moet de functie NULL teruggeven.
- *Optioneel:* gebruik ook de tabel P_CURSUSSEN en bepaal niet het aantal dagen dat iemand komt, maar het totaal aantal dagen dat iemand op cursus mag komen.

```
create or replace function aantal_dagen(f_naam in varchar2) return number is
  cursor c_pers (b_naam varchar2)is
    select nr
    from p_personen
    where upper(naam)=upper(b_naam);
  cursor c_ophalen_dag(b_cursistnr number) is
    select sum(aant_dag)
    from p_cursussen
    where nr in (select distinct cur_nr
                 from p_boekingen
                 where per_nr=b_cursistnr);
r_cursist c_pers%rowtype;
v_aant_dag number;
begin
  open c_pers(f_naam);
  fetch c_pers into r_cursist;
  if c_pers%notfound then
    close c_pers;
    return null;
  else close c_pers;
    open c_ophalen_dag(r_cursist.nr);
    fetch c_ophalen_dag into v_aant_dag;
    close c_ophalen_dag;
  end if;
return v_aant_dag;
end;
/show err
```

3. Maak een procedure CURSISTEN_OVERZICHT die waarde van AANTAL_DAGEN voor alle cursisten uit de tabel P_PERSONEN in de hulptabel zet, samen met de naam van de cursist. Als een persoon geen cursus volgt moet er een melding in HULPTABEL worden gezet.

```
create or replace procedure cursisten_overzicht is
cursor c_ophalen_naam is
  select naam
  from p_personen;
v_tot_dag number;
begin
for r_ophalen_naam in c_ophalen_naam loop
  v_tot_dag:=aantal_dagen(r_ophalen_naam.naam);
  if v_tot_dag is not null then
    insert into hulptabel values(null,r_ophalen_naam.naam,v_tot_dag );
  else insert into hulptabel values (null, r_ophalen_naam.naam, 'Dit is geen
cursist');
  end if;
end loop;
end;
/
show err
```

2. Procedures en functies

4. Maak een operator MAAL die de volgende uitvoer geeft:

- maal(2,3) => 6
- maal(3,'a') => 'aaa'
- maal('cursist',2) => 'cursistcursist'
- maal('a','b') => Een foutmelding met de tekst: 'Dit kan niet'

```
create or replace function getal(getal1 number,getal2 number) return number as
  totaal number;
begin
  totaal:=getal1*getal2;
  return totaal;
end;
/

create or replace function getal_tekst(getal number,tekst varchar2) return varchar2
as
  totaal varchar2(50);
begin
  for rij in 1..getal loop
   totaal:=totaal||tekst;
  end loop;
  return totaal;
end;
/

create or replace function tekst_getal(tekst varchar2,getal number) return varchar2
as
  totaal varchar2(50);
begin
  for rij in 1..getal loop
    totaal:=totaal||tekst;
  end loop;
  return totaal;
end;
/

create or replace function tekst_tekst(tekst1 varchar2,tekst2 varchar2) return
varchar2 as
  totaal varchar2(30);
begin
  totaal:='dit kan niet';
  return totaal;
end;
/

create or replace operator maal binding
            (number,number) return number using getal,
            (number,varchar2) return varchar2 using getal_tekst,
            (varchar2,number) return varchar2 using tekst_getal,
            (varchar2,varchar2) return varchar2 using tekst_tekst
/
```

Optionele opgaven:

5. Laat de relatie tussen CURSISTEN_OVERZICHT en AANTAL_DAGEN zien met behulp van DEPTREE_FILL.

```
EXECUTE DEPTREE_FILL('FUNCTION', user, 'AANTAL_DAGEN')

select * from ideptree;

DEPENDENCIES
--------------------------------------------------------------------------------
FUNCTION <user>.AANTAL_DAGEN
   PROCEDURE <user>.CURSISTEN_OVERZICHT
```

2. Procedures en functies

6. Laat zien welke tabellen AANTAL_DAGEN benaderd met behulp van USER_DEPENDENCIES.

```
select name
,      type
,      substr(referenced_name,1,20) referenced_name
,      substr(referenced_type,1,20) referenced_type
from user_dependencies
where name='AANTAL_DAGEN'
and referenced_type='TABLE';
```

```
NAME                           TYPE        REFERENCED_NAME      REFERENCED_T
------------------------------ ----------- -------------------- ------------
AANTAL_DAGEN                   FUNCTION    P_BOEKINGEN          TABLE
AANTAL_DAGEN                   FUNCTION    P_CURSUSSEN          TABLE
AANTAL_DAGEN                   FUNCTION    P_PERSONEN           TABLE
```

3. Packages

1. Definieer de package OPLEIDING met daarin de in hoofdstuk 2 gemaakte procedures en functie:
   - VOEGTOE_CURSUS procedure
   - AANTAL_DAGEN functie
   - CURSISTEN_OVERZICHT procedure

   Zorg ervoor dat de procedures en functie ook buiten de package gebruikt kunnen worden.

```
create or replace package opleiding is
  procedure voegtoe_cursus(p_cursusnaam varchar2, p_aant_dag number, p_prijs number);
  function aantal_dagen(f_naam in varchar2) return number;
  procedure cursisten_overzicht;
end;
/
show err
-- ------------------------------------------------------------------------------
create or replace package body opleiding is
  procedure voegtoe_cursus (p_cursusnaam varchar2
                          , p_aant_dag number
                          , p_prijs number)
  is
  cursor c_nieuw_nr is
    select max(nr)+1
    from p_cursussen;
  cursor c_cur (b_naam varchar2)is
    select 'x'
    from p_cursussen
    where naam=b_naam;
  v_cursusnr number;
  v_controle varchar2(1);
  e_te_lang exception;
  e_dubbel  exception;
begin
  if p_aant_dag > 25 then
    raise e_te_lang;
  else
    open c_cur(p_cursusnaam);
    fetch c_cur into v_controle;
    if c_cur%found then
      close c_cur;
      raise e_dubbel;
    else
      open c_nieuw_nr;
      fetch c_nieuw_nr into v_cursusnr;
      close c_nieuw_nr;
    end if;
    close c_cur;
    insert into p_cursussen values (v_cursusnr, p_cursusnaam, p_aant_dag, p_prijs);
  end if;
exception
  when e_te_lang then
     insert into hulptabel values (null, 'De cursus duurt te lang', null);
  when e_dubbel then
    insert into hulptabel values (null, p_cursusnaam,'Deze cursus komt al voor');
end voegtoe_cursus;
-- ------------------------------------------------------------------------------
```

```
function aantal_dagen(f_naam in varchar2) return number is
cursor c_pers(b_naam varchar2) is
    select nr
    from p_personen
    where upper(naam)=upper(b_naam);
  cursor c_ophalen_dag(b_cursistnr number) is
    select sum(aant_dag)
    from p_cursussen
    where nr in (select distinct cur_nr
                 from p_boekingen
                 where per_nr=b_cursistnr);
r_cursist c_pers%rowtype;
v_aant_dag number;
begin
  open c_pers(f_naam);
  fetch c_pers into r_cursist;
  if c_pers%notfound then
    close c_pers;
    return null;
  else close c_pers;
    open c_ophalen_dag(r_cursist.nr);
    fetch c_ophalen_dag into v_aant_dag;
    close c_ophalen_dag;
  end if;
return v_aant_dag;
end aantal_dagen;
-- ------------------------------------------------------------------------
procedure cursisten_overzicht is
cursor c_ophalen_naam is
  select naam
  from p_personen;
v_tot_dag number;
begin
for r_ophalen_naam in c_ophalen_naam loop
  v_tot_dag:=aantal_dagen(r_ophalen_naam.naam);
  if v_tot_dag is not null then
    insert into hulptabel values(null,r_ophalen_naam.naam,v_tot_dag );
  else insert into hulptabel values (null, r_ophalen_naam.naam, 'Dit is geen
cursist');
  end if;
end loop;
end cursisten_overzicht;
end;
/
show err
```

3. Packages

2. In de package specificatie wordt de constante MAX_AANTAL_DAGEN opgenomen, met als initialisatie waarde 25. Deze constante wordt gebruikt binnen de procedure VOEGTOE_CURSUS om het op dat moment gelden maximum aantal dagen op te vragen. De rest van de code blijft hetzelfde.

```
create or replace package opleiding is
  procedure voegtoe_cursus(p_cursusnaam varchar2, p_aant_dag number, p_prijs number);
  function aantal_dagen(f_naam in varchar2) return number;
  procedure cursisten_overzicht;
  max_aantal_dagen constant number:=25;
end;
/
show err
-- ------------------------------------------------------------------------------
create or replace package body opleiding is
  procedure voegtoe_cursus (p_cursusnaam varchar2
                          , p_aant_dag number
                          , p_prijs number)
is
  cursor c_nieuw_nr is
    select max(nr)+1
    from p_cursussen;
  cursor c_cur (b_cursusnaam varchar2)is
    select 'x'
    from p_cursussen
    where naam=b_cursusnaam;
  v_cursusnr number;
  v_controle varchar2(1);
  e_te_lang exception;
  e_dubbel  exception;
begin
  if p_aant_dag > opleiding.max_aantal_dagen then
    raise e_te_lang;
  else
    open c_cur(p_cursusnaam);
    fetch c_cur into v_controle;
    if c_cur%found then
      close c_cur;
      raise e_dubbel;
    else
      open c_nieuw_nr;
      fetch c_nieuw_nr into v_cursusnr;
      close c_nieuw_nr;
    end if;
    close c_cur;
    insert into p_cursussen values (v_cursusnr, p_cursusnaam, p_aant_dag, p_prijs);
  end if;
exception
  when e_te_lang then
     insert into hulptabel values (null, 'De cursus duurt te lang', null);
  when e_dubbel then
    insert into hulptabel values (null, p_cursusnaam,'Deze cursus komt al voor');
end voegtoe_cursus;
.....
etc...
```

3. Packages

3. Maak een package PACK_VOLGNR met daarin de procedure VOLGNR. Iedere keer dat de procedure VOLGNR wordt aangeroepen, moet er een nieuw volgnummer in HULPTABEL geplaatst worden. Maak hierbij gebruik van een globale variabele en NIET van een sequence.
Naast het volgnummer moet in HULPTABEL de tijd worden gezet, waarop het nieuwe volgnummer gegenereerd is. Nadat de procedure VOLGNR 3 keer is uitgevoerd, moet het volgende in de hulptabel staan (de datum kan uiteraard afwijken):

```
   KOLOM1 KOLOM2                          KOLOM3
--------- ---------------------------- ------------------------------
        1 01 january 2006 09:47:27
        2 01 january 2006 09:47:35
        3 01 january 2006 09:47:39
```

```
set serveroutput on
create or replace package pack_volgnr is
 v_volgnr number:=0;
 procedure volgnr;
end;
/
create or replace package body pack_volgnr is
procedure volgnr is
 v_ophogen number:=1;
begin
v_volgnr:=v_volgnr+v_ophogen;
insert into hulptabel values (v_volgnr, to_char(sysdate,'dd mon yyyy hh24:mi:ss'),
                              null);
end volgnr;
end;
/
```

Optionele opgave:

4. De procedure CURSISTEN_OVERZICHT moet een nieuwe procedure OVERZICHT aanroepen, dit is binnen de package OPLEIDING een private procedure.

Als CURSISTEN_OVERZICHT wordt uitgevoerd zonder dat er een parameter wordt opgegeven, dan moet een overzicht gemaakt worden van alle cursisten van de tabel PERSONEN. Hierbij moet gebruik gemaakt worden van de nieuwe procedure OVERZICHT. Het overzicht moet in HULPTABEL worden geplaatst, bij personen die geen cursisten zijn moet een melding in HULPTABEL komen.

Wordt bij het uitvoeren CURSISTEN_OVERZICHT wel een parameter opgegeven dan moet door de procedure OVERZICHT alleen de gegevens worden opgehaald van de genoemde cursist. Als de betreffende naam niet bestaat moet een melding op het **scherm** worden gegeven.
- Gebruik een default value voor de parameter in procedure CURSISTEN_OVERZICHT.
- Maak bij de procedure OVERZICHT gebruik van overloading.

```
create or replace package opleiding is
  procedure voegtoe_cursus(p_cursusnaam varchar2, p_aant_dag number, p_prijs number);
  function aantal_dagen(f_naam in varchar2) return number;
  procedure cursisten_overzicht(p_naam varchar2:='Niemand');
  pragma restrict_references(aantal_dagen, wnds,wnps,rnps);
  max_aantal_dagen constant number:=25;
end;
/
create or replace package body opleiding is
.....-- procedure voegtoe_cursus
.....-- functie aantal_dagen
-- ------------------------------------------------------------------------------
```

```
procedure overzicht is
  cursor c_ophalen_naam is
    select naam
    from p_personen;
  v_tot_dag number;
begin
  for r_ophalen_naam in c_ophalen_naam loop
    v_tot_dag:=aantal_dagen(r_ophalen_naam.naam);
    if v_tot_dag is not null then
      insert into hulptabel values(null,r_ophalen_naam.naam,v_tot_dag );
    else insert into hulptabel values (null, r_ophalen_naam.naam, 'Dit is geen
cursist');
    end if;
  end loop;
end overzicht;
-- ------------------------------------------------------------------------------
procedure overzicht(p_naam varchar2) is
  v_tot_dag number;
begin
  v_tot_dag:=aantal_dagen(p_naam);
  if v_tot_dag is not null then
    insert into hulptabel values(null,p_naam,v_tot_dag );
  else raise_application_error(-20005, p_naam||' dit is geen cursist');
  end if;
end overzicht;
-- ------------------------------------------------------------------------------
procedure cursisten_overzicht(p_naam varchar2:='Niemand') is
begin
  if p_naam='Niemand' THEN
    overzicht;
  else overzicht(p_naam);
  end if;
end cursisten_overzicht;
end;
/
show err
```

3. Packages

4. Native dynamisch SQL en standaard packages voor SQL

1. Maak een procedure die op het scherm laat zien hoeveel rijen een op te geven tabel bevat. Zorg ervoor dat op het scherm bijvoorbeeld de volgende boodschap wordt getoond:

```
De tabel P_KANTOREN bevat 4 rijen.
```

   a. Doe dit met behulp van DBMS_SQL, noem de procedure DBMS_TEL_RIJ.
   b. Doe dit met behulp van native dynamische SQL, noem de procedure NDS_TEL_RIJ.

```
create or replace procedure dbms_tel_rij(p_naam varchar2) is
 v_int integer;
 v_dummy number;
 v_aantal number;
begin
 v_int := dbms_sql.open_cursor;
 dbms_sql.parse(v_int,'select count(1) from '||p_naam,1);
 dbms_sql.define_column(v_int,1,v_aantal);
 v_dummy := dbms_sql.execute_and_fetch(v_int);
 dbms_sql.column_value(v_int,1,v_aantal);
 dbms_sql.close_cursor(v_int);
 dbms_output.put_line('het aantal rijen in de tabel '||p_naam||' is '||v_aantal);
end;
/

create or replace procedure nds_tel_rij(p_naam varchar2) is
 v_stmt varchar2(200);
 v_aantal number;
begin
 v_stmt := ('select count(1) from '||p_naam);
 execute immediate v_stmt into v_aantal;
 dbms_output.put_line('het aantal rijen in de tabel '||p_naam||' is '||v_aantal);
end;
/
```

2. Maak een procedure TABEL_KOLOM die een tabel GEGEVENS aanmaakt en daarin de waarden uit een opgegeven tabel en kolom plaatst. Voordat dit gebeurt moet door de procedure eerst worden nagegaan of de tabel GEGEVENS al bestaat, is dat het geval dan moet de tabel worden verwijderd.
Hint: Maak bij deze opgave gebruik van EXEC_DDL_STATEMENT en niet van DBMS_SQL.

```
create or replace procedure tabel_kolom(p_tabel varchar2, p_kolom varchar2) is
  cursor c_tab is
    select 'x'
    from user_tables
    where table_name='GEGEVENS';
  v_controle varchar2(1);
begin
  open c_tab;
  fetch c_tab into v_controle;
  if c_tab%found then
    dbms_utility.exec_ddl_statement('drop table gegevens');
  end if;
  close c_tab;
  dbms_utility.exec_ddl_statement('create table gegevens as select '||p_kolom||' from
'||p_tabel);
end;
/
```

4. Standaard packages

Optionele opgave:

3 Jaarlijks krijgen werknemers een bepaald percentage van hun salaris aan salarisverhoging. Dit percentage is niet ieder jaar hetzelfde. Schrijf een procedure PE43 waarmee de salarissen in de tabel P_WERKNEMERS gewijzigd kunnen worden en de variabele verhoging (een percentage) opgegeven kan worden. Zorg ervoor dat bij het uitvoeren van deze procedure de nieuwe salarissen als volgt op het scherm worden getoond.

```
SQL> exec pe43(2)
SMITS 7200
ALKEMA 7800
WALSTRA 6750
PIETERS 11925
VERGEER 6750
KLAASEN 11550
HEUVEL 10350
SANDERS 12000
KRAAY 18000
DROST 7500
ADELAAR 6300
APPEL 5850
VERMEULEN 11700
MANDERS 6900

PL/SQL procedure successfully completed.
```

Test de procedure en maak vervolgens de wijzigingen in de tabel WERKNEMERS weer ongedaan.

```
set serveroutput on

create or replace procedure pe43(p_percentage number) is
 cursor c_werknemers is
        select persnr, naam
        from p_werknemers;
 v_nieuwsal number;
 v_stmt varchar2(1000);
 v_personeel number;
begin
 for r_werknemers in c_werknemers loop
   v_personeel:=r_werknemers.persnr;
   v_stmt := 'update p_werknemers set sal = sal+(sal*:1) '||
             ' where persnr=:2 returning sal    into :3';
   execute immediate v_stmt using p_percentage,v_personeel returning into v_nieuwsal;
   dbms_output.put_line(r_werknemers.naam|| ' '||v_nieuwsal);
 end loop;
 rollback;
end;
/
```

5 Scheduling

1. Maak een externe schedule PERMINUUT aan met een interval van een minuut, een begintijd die gelijk is aan de huidige tijd en zonder eindtijd.

```
exec dbms_scheduler.create_schedule (schedule_name => 'perminuut', -
                                     repeat_interval => 'FREQ=MINUTELY')
```

2. Maak een programma VOLGNRPROGRAM die de procedure VOLGNR uit de package PACK_VOLGNR uitvoert. Deze procedure heeft u in hoofdstuk 3 aangemaakt en zet een volgnummer in HULPTABEL.

```
exec dbms_scheduler.create_program ( program_name => 'volgnrprogram', -
                                     program_type => 'stored_procedure', -
                                     program_action => 'pack_volgnr.volgnr', -
                                     enabled => true)
```

3. Maak een job JOB_VOLGNR aan op basis van het programma VOLGNRPROGRAM en de schedule PERMINUUT.

```
exec  dbms_scheduler.create_job(job_name => 'job_volgnr', -
                                program_name => 'volgnrprogram', -
                                schedule_name => 'perminuut', -
                                enabled => true)
```

4. Controleer of de job daadwerkelijk wordt uitgevoerd. Laat tevens door middel van een SQL-statement zien welke actie de job uitvoert, wanneer de volgende keer is dat de job wordt uitgevoerd en wat het herhalingsinterval is.

```
select p.program_action, j.next_run_date, s.repeat_interval
from user_scheduler_programs p
,    user_scheduler_jobs j
,    user_scheduler_schedules s
where p.program_name=j.program_name
and   s.schedule_name=j.schedule_name;
```

5. Verwijder de aangemaakte job, schedule en programma **nadat** de coach de opgaven heeft nagekeken.

```
begin
 dbms_scheduler.drop_job('job_volgnr');
 dbms_scheduler.drop_program('volgnrprogram');
 dbms_scheduler.drop_schedule('perminuut');
end;
/
```

Optionele opgave

6. Maak met behulp van de package DBMS_JOB een job die elke twee minuten een rij aan HULPTABEL toevoegt met daarin de tijd en de tekst “DBMS_JOB”. Controleer of de job daadwerkelijk wordt uitgevoerd.

```
declare
  v_jobnr binary_integer;
begin
  dbms_job.submit(v_jobnr
  , 'begin
     insert into hulptabel values (null
                                   ,to_char(sysdate,''hh24:mi:ss'')
                                   ,''DBMS_JOB''); end;'
  , sysdate
  ,'sysdate + 2/(24*60)');
end;
/
```

5. Schedulen

7. Laat door middel van een SQL statement zien welke statement de job uitvoert en wanneer de volgende keer is dat de job wordt uitgevoerd.

```
SELECT job
, what
, last_date
, last_sec
, next_date
, next_sec
FROM user_jobs;
```

8. Verwijder de job **nadat** de coach de opgaven heeft nagekeken.

```
exec dbms_job.remove(<jobnr>)
```

6. Veelzijdige standaard packages

1. Maak een procedure RANDOM_GETAL die een random getal in HULPTABEL zet samen met de tijd waarop de procedure wordt uitgevoerd.

```
create or replace procedure random_getal is
  v_getal number;
begin
  v_getal:=dbms_random.random;
  insert into hulptabel values (null,v_getal, to_char(sysdate, 'hh24:mi:ss'));
  dbms_random.terminate;  -- niet noodzakelijk is Obsolete
end;
/
```

2. In een trigger kunt u PL/SQL-code plaatsen die uitgevoerd wordt wanneer een bepaalde gebeurtenis plaatsvindt. In het **volgende hoofdstuk** zult u een trigger maken die iedere keer dat de tabel P_KANTOREN wordt aangepast met behulp van procedures en functies uit de package DBMS_PIPE een boodschap in de verbinding 'contr_kan_<gebruikersnaam>' plaatst. Wanneer de tabel P_KANTOREN bijvoorbeeld 2 keer wordt aangepast, zullen er door de trigger dus 2 boodschappen in de verbinding worden gezet.

   In deze opgave gaat u de procedure CONTROLEER_KANTOOR maken die de boodschappen uit de verbinding 'contr_kan_<gebruikersnaam>' ophaalt. Wanneer er bijvoorbeeld 2 boodschappen worden opgehaald door deze procedure, moet op het scherm de boodschap 'De tabel is 2 keer aangepast' verschijnen. Wanneer er geen boodschappen in de verbinding aanwezig zijn, moet de boodschap 'Geen aanpassing' verschijnen.

   Maak de procedure CONTROLEER_KANTOOR aan. Om de procedure te testen kunt u even "handmatig" (via een PL/SQL blok) alvast wat boodschappen in de verbinding plaatsen. Bewaar deze code, zodat u deze kunt hergebruiken wanneer u in het volgende hoofdstuk de trigger gaat maken.

```
CREATE OR REPLACE PROCEDURE controleer_kantoor AS
  v_dummy   INTEGER;
  v_aantal  NUMBER := 0;
BEGIN
  v_dummy := DBMS_PIPE.RECEIVE_MESSAGE('contr_kan_<gebruikersnaam>',1);
  WHILE dummy = 0 LOOP
    v_aantal := v_aantal + 1;
    v_dummy := DBMS_PIPE.RECEIVE_MESSAGE('contr_kan_<gebruikersnaam>',1);
  END LOOP;
  IF v_aantal > 0
    THEN DBMS_OUTPUT.PUT_LINE('De tabel P_KANTOREN is '||v_aantal||
                  ' keer aangepast');
    ELSE DBMS_OUTPUT.PUT_LINE('Geen aanpassing');
  END IF;
END;
/

DECLARE
  v_dummy   NUMBER;
BEGIN
     DBMS_PIPE.PACK_MESSAGE('attentie');
     v_dummy := DBMS_PIPE.SEND_MESSAGE('contr_kan_<gebruikersnaam>');
END;
/
```

Optionele opgave:

3. Maak een procedure TOON_WERKNEMERS die de kolommen PERSNR en NAAM uit de tabel P_WERKNEMERS op de volgende manier op het scherm toont.

```
3381  SMITS           3462  ALKEMA          3518  WALSTRA
3930  PIETERS         4510  VERGEER         4621  KLAASEN
5810  HEUVEL          5931  SANDERS         6221  KRAAY
6500  DROST           6681  ADELAAR         7900  APPEL
7902  VERMEULEN       8222  MANDERS
```

```
create or replace procedure toon_werknemers is
  cursor c_werkn is
    select persnr, naam
    from p_werknemers;
  v_teller number:=0;
begin
  for r_werkn in c_werkn loop
    dbms_output.put(r_werkn.persnr||' '||rpad(r_werkn.naam, 15));
    v_teller:=v_teller+1;
    if v_teller=3 then
      dbms_output.new_line;
      v_teller:=0;
    end if;
  end loop;
  dbms_output.new_line;
end;
/
```

7. Triggers

1. Zorg ervoor dat het aantal dagen dat een cursus duurt niet gewijzigd kan worden. Geef een duidelijke melding.

```
create or replace trigger cur_bsu_aant_dag before update of aant_dag on p_cursussen
for each row when (old.aant_dag <> new.aant_dag)
begin
  raise_application_error(-20006, 'Het aantal_dagen mag niet worden gewijzigd');
end;
/
```

De reden om hier een rij trigger voor te maken, is omdat formsapplicaties standaard alle velden in een statement opnemen (i.v.m. performance). Op deze manier kan worden voorkomen dat de trigger ten onrechte afgaat.

2. Nieuwe cursussen moet altijd minimaal 2 dagen duren. Verander het aantal dagen zonodig in 2, en geef een melding dat het aantal dagen is veranderd. Zorg er tevens voor dat bij nieuwe cursussen de cursusprijs altijd het aantal dagen* 495 euro is. Geef een melding als de prijs is aangepast.

```
create or replace trigger cur_bri before insert on p_cursussen
for each row
begin
  if :new.aant_dag < 2 then
    :new.aant_dag:=2;
    dbms_output.put_line('Het aantal dagen is veranderd in 2');
  end if;
  if :new.prijs <> :new.aant_dag*495 then
    :new.prijs:=:new.aant_dag*495;
    dbms_output.put_line('De prijs van de cursus is aangepast naar: '||:new.prijs);
  end if;
end;
/
```

3. a. Maak een tabel SALWIJZIGING met volgende kolommen:
   OUD_SAL numeriek
   NIEUW_SAL numeriek
   WERKNEMER alfanumeriek 20 karakters
   GEBRUIKER alfanumeriek 20 karakters
   DATUM datum.

```
create table salwijziging
  (oud_sal number
  , nieuw_sal number
  , naam varchar2(20)
  , gebruiker varchar2(20)
  , datum date)
/
```

   b. Maak een trigger WER_.... op de tabel P_WERKNEMERS die ervoor zorgt dat bij een salariswijziging een record wordt toegevoegd aan de tabel SALWIJZIGING. In dit record moet de volgende informatie worden opgeslagen: het oude salaris, het nieuwe salaris, de naam van de werknemer, de gebruikersnaam van degene die de wijziging doorvoert en de datum waarop dit gebeurt. De trigger moet alleen af gaan als het salaris echt gewijzigd is.

```
create or replace trigger wer_aru_sal after update of sal on p_werknemers
for each row when (new.sal <> old.sal)
begin
  insert into salwijziging values (:old.sal, :new.sal, :new.naam, user, sysdate);
end;
/
```

7. Triggers

4 U gaat de trigger T_KANTOOR aanmaken die afgaat wanneer waarden in de tabel P_KANTOREN worden aangepast. Deze trigger moet in de verbinding 'contr_kan_<gebruikersnaam>' de boodschap 'ATTENTIE' plaatsen. Deze boodschap wordt uitgelezen in de procedure CONTROLEER_KANTOOR die al gedeeltelijk in het vorige hoofdstuk is aangemaakt.

a) Maak de trigger T_KANTOOR aan die afgaat na iedere update op de tabel p_kantoren en de boodschap 'ATTENTIE' in de verbinding 'contr_kan_<gebruikersnaam>' zet.

```
CREATE OR REPLACE TRIGGER t_kantoor AFTER UPDATE ON p_kantoren
for each row
DECLARE
  v_dummy   NUMBER;
BEGIN
  DBMS_PIPE.PACK_MESSAGE('attentie');
  v_dummy := DBMS_PIPE.SEND_MESSAGE('contr_kan_<gebruikersnaam>');
END;
/
```

b) Pas de procedure CONTROLEER_KANTOOR uit het vorige hoofdstuk aan, zodat de boodschap in de verbinding 'contr_kan_<gebruikersnaam>' uitgelezen wordt en op het scherm wordt afgedrukt.

```
CREATE OR REPLACE PROCEDURE controleer_kantoor AS
  v_dummy      INTEGER;
  v_aantal     NUMBER := 0;
  v_boodschap VARCHAR2(50);
BEGIN
 WHILE  DBMS_PIPE.RECEIVE_MESSAGE('contr_kan_<gebruikersnaam>',1) = 0 LOOP
  DBMS_PIPE.UNPACK_MESSAGE(v_boodschap);
  IF v_boodschap = 'ATTENTIE' THEN
    v_aantal := v_aantal + 1;
  END IF;
  END LOOP;
   IF v_aantal > 0 THEN
     DBMS_OUTPUT.PUT_LINE('De tabel P_KANTOREN is '||v_aantal||' keer aangepast');
   ELSE DBMS_OUTPUT.PUT_LINE('Geen aanpassing');
   END IF;
END;
/
```

c) Voer nu de volgende updates uit op de tabel p_kantoren zonder deze via een COMMIT vast te leggen. De trigger T_KANTOOR zal nu 2 keer afgaan.

```
UPDATE p_kantoren SET naam = upper(naam) WHERE kantnr=10;
UPDATE p_kantoren SET naam = upper(naam) WHERE kantnr=20;
```

d) Open een tweede SQL*Plus sessie en log aan onder uw eigen gebruikersnaam. Controleer of de inhoud van de tabel P_KANTOREN in deze sessie ongewijzigd is en zet de serveroutput aan. Voer vervolgens de procedure CONTROLEER_KANTOOR uit om te bekijken of en hoe vaak de tabel P_KANTOREN is aangepast in de andere sessie.

```
SQL> set serveroutput on
SQL> execute controleer_kantoor
ATTENTIE
De tabel P_KANTOREN is 2 keer aangepast

PL/SQL procedure successfully completed.
```

7. Triggers

5 Maak een autonome trigger BOETE_... aan. Deze moet afgaan wanneer aan de tabel P_BOETEBEDRAGEN een rij wordt toegevoegd. De nieuwe rijen moeten ook aan de schaduwtabel BOETEBEDRAGEN_LOG worden toegevoegd. Dit moet onafhankelijk zijn van het feit of ze daadwerkelijk zijn toegevoegd aan P_BOETEBEDRAGEN of via een rollback zijn teruggedraaid.

Controleer de werking van de trigger door een aantal rijen toe te voegen aan de tabel P_BOETEBEDRAGEN en sommige daarvan vast te leggen en andere terug te draaien.

De tabel boetebedragen_log kunt u als volgt aanmaken

```
SQL> CREATE TABLE boetebedragen_log as
  2  SELECT *
  3  FROM p_boetebedragen
  4  WHERE 1=0;

Table created.
```

```
create or replace trigger boete_bri
before insert on p_boetebedragen for each row
declare
 pragma autonomous_transaction;
begin
 insert into boetebedragen_log
 values (:new.nr, :new.spelnr, :new.datum, :new.bedrag);
 commit;
end;
/
```

7. Triggers

Optionele opgave:

6 Implementeer de volgende bedrijfsregels:

- De verhoging van een salaris mag niet hoger zijn dan 5%. Geef een duidelijke melding waarom het salaris niet kan worden verhoogd.
- Het salaris van een werknemer mag maximaal 30% meer zijn dan het gemiddelde salaris van alle werknemers. Geef een duidelijke melding als de wijziging niet kan worden doorgevoerd. Laat in deze melding het maximaal toegestane salaris zien.
- Voor de directeur geldt een uitzondering. Die mag wel meer gaan verdienen dan 130% van het gemiddelde salaris.

```
create or replace package sal_werkn as
 cursor c_max is select avg(sal)*1.3
                 from p_werknemers;
 v_maxsal p_werknemers.sal%type;
end;
/

create or replace trigger wer_bsu before update of sal on p_werknemers
begin
 open sal_werkn.c_max;
 fetch sal_werkn.c_max into sal_werkn.v_maxsal;
 close sal_werkn.c_max;
end;
/


create or replace trigger wer_bru_sal3 before update of sal on p_werknemers
for each row when (old.sal <> new.sal)
begin
 if :new.sal > :old.sal*1.05 then
  raise_application_error(-20001,
    'Een salaris mag nooit meer dan 5% verhoogd worden. Dit is gebeurd bij: '
     ||:new.persnr);
 end if;
 if :new.functie <> 'DIRECTEUR' then
     if sal_werkn.v_maxsal < :new.sal then
        raise_application_error(-20002,
           'Een salaris mag maximaal '||sal_werkn.v_maxsal||
           'bedragen. Dit is gebeurd bij: '||:new.persnr);
     end if;
  end if;
end;
/
```

8. System Event en DDL Triggers

1 Maak een tabel VERWIJDERD, met vier kolommen, ACTIE, GEBRUIKER, OBJECT en TIJD.

```
SQL> CREATE TABLE verwijderd (actie varchar2(20)
  2                          ,gebruiker varchar2(20)
  3                          ,object varchar2(20)
  4                          ,tijd date);

Table created.
```

2 Maak een trigger OBJ_VERWIJDERD, die voor ieder object dat uit uw eigen schema wordt verwijderd, een rij met de betreffende gegevens toevoegt aan de tabel VERWIJDERD.

```
SQL> CREATE OR REPLACE TRIGGER obj_verwijderd AFTER DROP ON nwg916.schema
  2  BEGIN
  3   INSERT INTO verwijderd VALUES ('Tabel verwijderd:'
  4                                 ,ora_login_user
  5                                 ,ora_dict_obj_name
  6                                 ,sysdate);
  7  END;
  8  /

Trigger created.
```

3 Maak een tabel aan die u vervolgens weer weggooit, Controleer of er een rij wordt toegevoegd aan de tabel VERWIJDERD.

```
SQL> CREATE TABLE x (y number);

Table created.

SQL> DROP TABLE x;

Table dropped.

SQL> SELECT *
  2  FROM verwijderd;

ACTIE                GEBRUIKER            OBJECT               TIJD
-------------------- -------------------- -------------------- ---------
Tabel verwijderd:    NWG916               X                    13-OCT-03
```

4 Maak een tweede trigger NIET_VERWIJDERD, die ervoor zorgt dat ook wordt bijgehouden wanneer geprobeerd wordt een niet bestaande tabel te verwijderen. Het is voldoende om de fout 'ORA-00942: table or view does not exists' af te vangen. De betreffende informatie moet in de tabel VERWIJDERD komen te staan.

```
SQL> CREATE OR REPLACE TRIGGER niet_verwijderd AFTER SERVERERROR ON nwg916.schema
  2   WHEN (ora_server_error(1)=942)
  3  BEGIN
  4   insert into verwijderd values('Niet verwijderd:'
  5                                ,ora_login_user
  6                                ,ora_dict_obj_name
  7                                ,sysdate);
  8  END;
  9  /

Trigger created.
```

5 Geef een DROP TABLE statement op een niet bestaande tabel. Controleer of er een rij wordt toegevoegd aan de tabel VERWIJDERD. Welke waarde ontbreekt er?

```
SQL> DROP TABLE x;
DROP TABLE x
           *
ERROR at line 1:
ORA-00942: table or view does not exist
```

```
SQL> SELECT *
  2  FROM verwijderd;

ACTIE                GEBRUIKER            OBJECT               TIJD
-------------------- -------------------- -------------------- ---------
Tabel verwijderd:    NWG916               X                    13-OCT-03
Niet verwijderd:     NWG916                                    13-OCT-03
```

De kolom object wordt niet gevuld, Dit betekent dat het attribuut ora_dict_obj_name geen waarde heeft gehad.

6 Probeer de tabel VERWIJDERD weg te gooien. Wat gebeurt er en waarom?

```
SQL> select status from user_objects
  2  where object_name = 'OBJ_VERWIJDERD';

STATUS
-------
VALID

SQL> DROP TABLE verwijderd;
DROP TABLE verwijderd
           *
ERROR at line 1:
ORA-04098: trigger 'NWG916.OBJ_VERWIJDERD' is invalid and failed re-validation


SQL> select status from user_objects
  2  where object_name = 'OBJ_VERWIJDERD';

STATUS
-------
INVALID

SQL> alter trigger obj_verwijderd compile;

Trigger altered.

SQL> select status from user_objects
  2  where object_name = 'OBJ_VERWIJDERD';

STATUS
-------
VALID
```

Omdat de tabel gevuld wordt door een trigger die afgaat op het verwijderen ervan is de actie gedoemd om te mislukken. Voor Oracle9*i* kreeg u in zo'n situatie de logischere fout 'ora04020: deadlock detected'. Het als invalid markeren van de trigger wil niet zeggen dat de tabel verwijderd is, deze is nog gewoon aanwezig.

7 Verwijder de triggers OBJ_VERWIJDERD en NIET_VERWIJDERD en de tabel VERWIJDERD, nadat de coach de opgaven heeft nagekeken.

```
SQL> DROP TRIGGER OBJ_VERWIJDERD;

Trigger dropped.

SQL> DROP TRIGGER NIET_VERWIJDERD;

Trigger dropped.

SQL> DROP TABLE verwijderd;

Table dropped.
```

8. System Event en DDL Triggers

Optionele opgave

6 Maak een trigger BOETE_.... aan die afgaat wanneer de tabel p_boetebedragen verwijderd wordt. De trigger moet ervoor zorgen dat automatisch de schaduwtabel boetebedragen_log verwijderd wordt.

```
create or replace trigger boete_asu
after drop on dix004.schema
when (sys.dictionary_obj_name='P_BOETEBEDRAGEN')
begin
 execute immediate 'drop table boetebedragen_log';
end;
/
```

8. System Event en DDL Triggers

9. Werken met LOBs

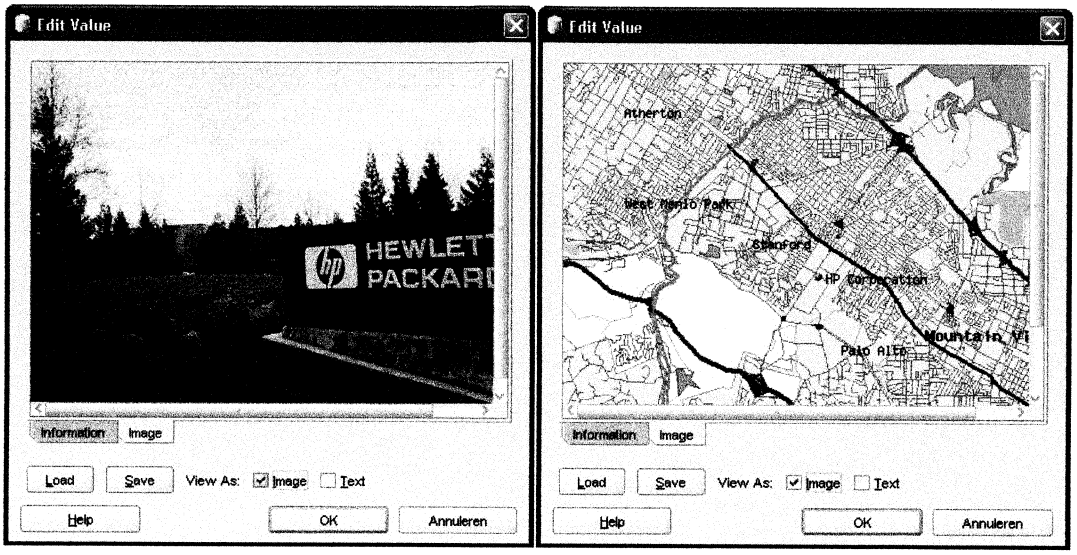
1 Herschrijf de procedure `laadblob` en zorg dat deze, voor het vullen van de BLOB kolom met behulp van het BFILE object, in plaats van het SELECT INTO statement gebruik gaat maken van een *cursor*.

Voeg vervolgens voor kantoor 30 het HP Office, de foto van het kantoor en de plattegrond toe. Controleer dit in SQL Developer met behulp van het Edit Value window.

```
create or replace procedure laadblob( p_oradir in varchar2
                                     ,p_blobbestand in varchar2
                                     ,p_refwaarde in number ) is
  cursor c_blob(a_kantnr in number) is
           select foto
           from lob_kantoren
           where kantnr = a_kantnr
           for update;
  v_blob c_blob%rowtype;
  v_lokatie bfile;
  v_recpos blob;
  v_bfilesize pls_integer;
  e_bestaatniet EXCEPTION;
begin
  open c_blob(p_refwaarde);
  fetch c_blob into v_blob;
   if c_blob%found then
      close c_blob;
      if DBMS_LOB.FILEEXISTS(bfilename(p_oradir,p_blobbestand)) = 0 then
         RAISE E_BESTAATNIET;
      else
         v_lokatie := bfilename(p_oradir,p_blobbestand);
         DBMS_LOB.FILEOPEN(V_LOKATIE);
      end if;
      v_bfilesize := dbms_lob.getlength(v_lokatie);
      DBMS_LOB.LOADFROMFILE (v_blob.foto , V_LOKATIE , v_bfilesize);
      DBMS_LOB.FILECLOSE(V_LOKATIE);
      commit;
   else
    close c_blob;
    raise e_bestaatniet;
   end if;
EXCEPTION
  WHEN E_BESTAATNIET THEN DBMS_OUTPUT.PUT_LINE('BFILE bestand bestaat niet!');
  WHEN OTHERS THEN DBMS_OUTPUT.PUT_LINE(sqlerrm);
END;
/

SQL> update lob_kantoren set foto = empty_blob()
  2  where kantnr = 30;

1 row updated.

SQL> exec laadblob('DIR_VOOR_CURSIST','HPHQ.gif',30)

PL/SQL procedure successfully completed.

SQL> update lob_kantoren
  2  set kaart = bfilename('DIR_VOOR_CURSIST', 'HPMap.gif')
  3  where kantnr= 30;

1 row updated.

SQL> commit;

Commit complete.
```

9. Werken met LOBs

In SQL Developer ziet dit er voor de twee kolommen als volgt uit:



2 In de Oracle Directory staat een tekst bestand: FROMFILE.txt.
Open dit bestand en zoek daarin, op ongeveer de 10e regel het, in hoofdletters getypte, woord BFILE. Vanaf dit punt willen we de tekst tot aan het woordje SYNTAX selecteren en toevoegen aan de in het hoofdstuk gemaakte CLOB kolom in de tabel in LOB_KANTOREN. Voeg de inhoud van dit tekstbestand vanaf positie 811(= 'BFILE') met een omvang van 364 (=tot aan 'SYNTAX') tekens, toe aan het de CLOB kolom op positie 1 voor kantoor 30.

Een paar aanwijzingen:

```
create or replace procedure leesclob( p_oradir in varchar2
                                    , p_clobbestand  in varchar2
                                    , p_refwaarde    in varchar2
                                    , p_omvang       in integer
                                    , p_doelpositie  in integer:=1
                                    , p_bronpositie  in integer:=1) is
```

Voorbeeld procedure aanroep

```
exec leesclob('DIR_VOOR_CURSIST','fromfile.txt',30,364,1,811)
```

Waarvoor geldt:

```
'DIR_VOOR_CURSIST'      De Oracle directory.
'fromfile.txt'          Het betreffende bestand.
30                      kenmerk van de kolom kantnr in lob_kantoren.
364                     aantal karakters uit FromFile.txt.
1                       startpositie in doel-LOB.
811                     startpositie in FromFile.txt.

create or replace procedure leesclob( p_oradir in varchar2
                                    , p_clobbestand  in varchar2
                                    , p_refwaarde    in varchar2
                                    , p_omvang       in integer
                                    , p_doelpositie  in integer:=1
                                    , p_bronpositie  in integer:=1) is
v_lokatie bfile;
v_recpos clob;
v_clobomvang pls_integer;
v_doelpos pls_integer;
e_bestaatniet exception;

begin
-- controle of clobfile bestaat en indien ja, inhoud ophalen.
 if dbms_lob.fileexists(bfilename(p_oradir,p_clobbestand)) = 0 then
            raise e_bestaatniet;
```

```
else
-- openen en lokatie van/in CLOB kolom bepalen
   v_lokatie := bfilename(p_oradir,p_clobbestand);
   v_clobomvang := dbms_lob.getlength(v_lokatie);
            if p_doelpositie = 0 then
               v_doelpos := v_clobomvang + 1;
            else v_doelpos := p_doelpositie;
            end if;

  dbms_lob.fileopen(v_lokatie);
  select route into v_recpos from lob_kantoren where kantnr = p_refwaarde for update;

       --syntax dbms_lob.loadfromfile():
       --DBMS_LOB.LOADFROMFILE(DEST_LOB, SRC_LOB, AMOUNT, DEST_OFFSET, SRC_OFFSET);
       dbms_lob.loadfromfile(v_recpos, v_lokatie, p_omvang, v_doelpos,
                  p_bronpositie);
       commit;
       dbms_lob.fileclose(v_lokatie);
       end if;

exception
  when e_bestaatniet then dbms_output.put_line('BFILE bestand bestaat niet!');
  when others then dbms_output.put_line(sqlerrm);
end;
/

SQL> SELECT route
  2  FROM lob_kantoren
  3  WHERE kantnr=30
  4  /

ROUTE
--------------------------------------------------------------------------------

BFILE.
Note: The input BFILE must have been opened prior to using this
procedure. No character set conversions are performed implicitly
when binary BFILE data is loaded into a CLOB. The BFILE data
must already be in the same character set as the CLOB in the
database. No error checking is performed to verify this..Summary of DBMS_LOB Sub
programs

SYNTAX
an de afrit linksaf slaan. U rijdt nu op de N224 (Californian road). Bij het 6e
.....etcetera
```

Index