CURSISTENMAP

Oracle Database:
PL/SQL voor ervaren programmeurs

Versie 2.0: januari-2013

Inhoudsopgave

1 Inleiding 1-1

1.1 Inleiding 1-1

1.2 Generaties 1-1

1.3 SQL en de generatiekloof 1-1

1.4 PL/SQL 1-2

1.5 De kracht van PL/SQL 1-2
1.5.1 Procedurele mogelijkheden 1-2
1.5.2 Verbeterde prestaties 1-3
1.5.3 Ondersteuning van Oracle tools 1-3
1.5.4 Toegesneden op Oracle 1-4

1.6 Voorbeeld 1-5

2 Basisbegrippen 2-1

2.1 Inleiding 2-1

2.2 Opbouw 2-1
2.2.1 Statements 2-2
2.2.2 Declaraties 2-4
2.2.3 Toekenningen 2-7
2.2.4 Sequences 2-7
2.2.5 %TYPE en %ROWTYPE 2-8
2.2.6 Expressies 2-9

2.3 Commentaar 2-9

2.4 NULL 2-10

3 PL/SQL Syntax 3-1

3.1 Inleiding 3-1

3.2 Condities 3-1
3.2.1 IF-THEN-ELSE statement 3-1
3.2.2 CASE-statement (9*i*) 3-5
3.2.3 CASE-expressie (9*i*) 3-7

3.3 Herhalingen 3-8
3.3.1 Sprongen maken met GOTO 3-8
3.3.2 WHILE lussen 3-9
3.3.3 FOR lussen met index 3-10
3.3.4 FOR lussen met cursor 3-11
3.3.5 Ongelimiteerde lussen 3-11
3.3.6 Voortijdig stoppen 3-12

3.4 Geneste blokken 3-13

3.5 Fouten in de syntax 3-16

4 Benaderen van de database 4-1

4.1 Inleiding 4-1

4.2 Raadpleging door middel van SELECT INTO (impliciete cursor) 4-1

4.3 Raadpleging door middel van een cursor 4-3
4.3.1 OPEN-FETCH-CLOSE 4-4
4.3.2 FOR lussen met cursor 4-5
4.4 Parameters 4-7
4.5 Controles 4-11
4.5.1 Cursorattributen 4-13
4.5.2 Controles met een lus 4-15
4.6 Zelf gedefinieerde records 4-17

5 Transacties 5-1

5.1 Inleiding 5-1
5.2 Raadpleging ten behoeve van wijzigingen 5-1
5.3 Transacties 5-3
5.3.1 Autonome transacties 5-6
5.4 Locking 5-9
5.4.1 Lees-consistentie 5-11

6 Foutafhandeling 6-1

6.1 Inleiding 6-1
6.2 Exceptions 6-1
6.3 RAISE 6-3
6.4 EXCEPTION_INIT 6-5
6.5 SQLCODE en SQLERRM 6-7

BIJLAGEN 6-11

OPDRACHTEN 1

UITWERKINGEN 2

NASLAGWERK

Oracle Database:
PL/SQL voor ervaren programmeurs

1 Inleiding

1.1 Inleiding

In deze cursus willen we een basis leggen om te kunnen programmeren met PL/SQL. Voordat we met het werkelijke programmeren gaan beginnen, gaan we eerst bekijken wat PL/SQL inhoudt en wat de voor- en nadelen zijn t.o.v. andere programmeertalen.

1.2 Generaties

Programmeertalen zijn in te delen in verschillende generaties. We kunnen hierin vijf generaties onderscheiden, die alle op een eigen niveau werken. De programmeertaal die het dichtst bij de fysieke opbouw van een computer staat, is de 1e generatietaal. Hoe hoger de generatie wordt, hoe meer een taal gaat lijken op spreektaal, hoe minder een programmeur hoeft te weten over de fysieke structuur van de computer en de gegevens daarin.

Het volgende overzicht toont de verschillen tussen de verschillende generaties, en geeft enkele voorbeelden:

- 1e generatietaal: bevindt zich op het fysieke niveau van de computer. Op dit niveau bestaan slechts nullen en enen, waardoor programmeren moeilijk is.
- 2e generatietaal: hiervoor is nog steeds kennis nodig van de opbouw van een computer, hoewel dit niet meer tot het fysieke niveau teruggaat. Er wordt geprogrammeerd in commando's die microprocessoren besturen. Voorbeeld: Assembler.
- 3e generatietaal: de taal die het meest als programmeertaal wordt gezien, ook wel *procedurele taal* genoemd. Een programma bestaat uit kleine stappen die ervoor zorgen dat het gewenste resultaat wordt bereikt. Voorbeelden: BASIC, Pascal, C++, PL/SQL.
- 4e generatietaal: lijkt meer op spreektaal. Om een bepaalde actie uit te voeren hoeft deze niet eerst te worden onderverdeeld in kleine stappen. Ook is niet meer van belang hoe en waar gegevens zijn opgeslagen. Voorbeelden: Prolog, SQL.
- 5e generatietaal: bevindt zich op het gebied van kunstmatige intelligentie. Er is niet echt meer sprake van programmeren, maar meer van modelleren. 5e generatietalen staan nog in de kinderschoenen.

1.3 SQL en de generatiekloof

SQL is een 4e generatietaal. De reden daarvan is, dat het zich onderscheidt van talen als C++ en PASCAL vanwege zijn eenvoud. Met SQL zijn geen procedures nodig. Wanneer we bijvoorbeeld gegevens willen opvragen uit een database, dan geven we een SELECT statement op waarin we bijna woordelijk aangeven welke gegevens we bedoelen. We hoeven ons niet te bekommeren over de manier waarop dat allemaal moet gebeuren: dit wordt intern geregeld door het RDBMS, dat daarvoor de benodigde procedures aanroept. Het zou een stuk ingewikkelder worden, wanneer we zelf steeds de procedures moesten schrijven.

Door die kant-en-klare afhandeling is enerzijds het programmeren enorm vereenvoudigd en daardoor minder tijdrovend, en minder voorbehouden aan specialisten. Anderzijds kent SQL een aantal tekortkomingen. Je kunt er uiteindelijk minder mee dan met 3GL (3e generatietaal, 3rd Generation Language).

1 Inleiding

SQL is en blijft een krachtige taal met een uitgebreid scala aan statements en functies. Het lezen en schrijven in de database mag trouwens alleen met SQL. Maar er kunnen zich situaties voordoen waarbij de programmeur zijn toevlucht moet nemen tot een procedurele taal wanneer SQL alleen niet toereikend is, of omdat het programma anders te omslachtig en te traag zou worden. De kunst is dan om SQL statements te nesten in een 3GL-source. Men krijgt dan EMBEDDED SQL. De 3^e generatietaal wordt de gastheertaal genoemd, ofwel HOST LANGUAGE.

1.4 PL/SQL

PL/SQL staat voor Procedural Language for SQL. Het is een procedurele taal (3GL) die in 1989 door de Oracle Corporation is ontwikkeld. Versie 2 kan uitsluitend gebruikt worden vanaf Oracle 7.

Vóór de komst van PL/SQL kon men kiezen uit de volgende gastheertalen: COBOL, Pascal, FORTRAN, C, Ada en PL/I. Volop keuze dus; waarom dan PL/SQL erbij? Het bijzondere van deze 3GL-taal is dat deze helemaal is toegespitst op SQL. De andere talen zijn dat niet, omdat ze ouder zijn dan SQL.

PL/SQL is feitelijk een uitbreiding op SQL en het gebruikt exact dezelfde functies en gegevenstypen. De SQL statements kunnen rechtstreeks in de source worden gecodeerd. Bovendien is speciaal rekening gehouden met de foutafhandeling van SQL.

Ook het ontwikkelen van PL/SQL programma's gaat soepeler. In de andere gastheertalen was het nesten van SQL minder eenvoudig: de statements moesten expliciet afgebakend worden; men moest de gegevenstypen voor database velden apart declareren en de foutafhandeling ging minder soepel. Om het object te krijgen moest de source eerst geprecompileerd worden teneinde de SQL statements in systeem-calls te vertalen. Na de compilatie moest het object tenslotte gelinkt worden met speciale libraries. Bij PL/SQL is alleen compilatie nodig, en dat gebeurt automatisch door Oracle na voltooiing van de source.

1.5 De kracht van PL/SQL

In PL/SQL kan veel eenvoudiger SQL gebruikt worden, dan in andere procedurele talen (3GL). Het volgende overzicht toont de voordelen en toepassingen van PL/SQL.

1.5.1 Procedurele mogelijkheden

Dit geldt in het algemeen voor 3GL met embedded SQL.
Typische zaken die in SQL niet kunnen worden nu mogelijk:

- gebruikmaking van hulpvariabelen en constanten;
- iteraties;
- conditionele en onconditionele sprongen;
- uitgebreide foutafhandeling.

Embedded SQL is krachtiger dan SQL of 3^e generatietalen alleen. Beide talen vullen elkaar aan. Denk bijvoorbeeld aan het gebruik van hulpvariabelen waardoor hulptabellen of complexe joins niet meer nodig zijn.

1.5.2 Verbeterde prestaties

De extra mogelijkheden, die een procedurele taal (3GL) tegenover een vierde generatietaal als SQL met zich meebrengt, komen ten goede aan de performance. PL/SQL zorgt er bovendien voor, dat er weinig gegevensverkeer hoeft te zijn tussen de gebruikersapplicatie en het RDBMS. Dit laat zich als volgt illustreren:

Als gebruiker hebben we nooit rechtstreeks contact met de database. Oracle neemt het werk tijdelijk van ons over en geeft na afloop het resultaat aan ons terug. We zouden dat kunnen vergelijken met een magazijn (de database) waar we een bestelling plaatsen (SQL statement) en de baliemedewerker (RDBMS) uw opdracht afhandelt. We mogen met SQL maar één bestelling per keer doen.

Met PL/SQL kunnen we meerdere statements in één keer doorgeven. Het aantal calls en sends kunnen drastisch gereduceerd worden; vooral in netwerken is dat gunstig.

Een programma geschreven in PL/SQL wordt een *PL/SQL-BLOK* genoemd. Zo'n blok bestaat uit procedurele statements en (niet verplicht) SQL statements. Het SQL wordt doorgegeven naar het RDBMS en de overige statements worden buiten het RDBMS om afgehandeld. In het vervolg zullen we de term PL/SQL-blok hanteren.

1.5.3 Ondersteuning van Oracle tools

PL/SQL is toepasbaar binnen verschillende Oracle tools, zoals SQL Developer, SQL*Plus, Oracle Developer en Oracle Designer en tevens als embedded PL/SQL in andere 3GL-talen.

In SQL*Plus kunnen we rechtstreeks een PL/SQL-blok intypen en runnen, net als bij een gewoon SQL statement. We kunnen ook het PL/SQL-blok bewaren als een commandobestand, en later op dezelfde manier opstarten als een SQL-commandobestand. PL/SQL-blokken kunnen ook opgenomen worden in SQL-commandobestanden. Substitutie variabelen kunnen doorgegeven worden aan een PL/SQL-blok.

In Oracle Developer en Oracle Designer wordt veel met triggers gewerkt. Triggers zijn door de ontwikkelaar geschreven routines binnen het schermprogramma. Ze worden direct of indirect geïnitieerd naar aanleiding van een bepaalde actie door de eindgebruiker. Doel van de triggers is om de acties van de gebruiker goed af te handelen en controles uit te voeren. Het gebruik van PL/SQL-routines heeft de volgende voordelen:

- De routines hoeven niet in trigger-stappen te worden uitgesplitst: SQL statements en procedurele statements vormen binnen PL/SQL een gesloten geheel.
- Voor complexe berekeningen en controles werden voorheen speciale user exits geschreven: programma's in 3GL, eventueel met embedded SQL. Men deed dat omdat zulke opdrachten binnen SQL te traag worden afgehandeld. Constructies als SELECT ... FROM DUAL dienen namelijk vermeden te worden. User exits zijn lastig te onderhouden. Vervang ze liever door PL/SQL-blokken. Want berekeningen e.d. met schermvelden kunnen ook in PL/SQL buiten de database om.
- Triggers in PL/SQL worden reeds tijdens de ontwikkelfase gecontroleerd op syntaxfouten.

Het is mogelijk om naast SQL statements ook complete PL/SQL-blokken te nesten in een 3GL taal zoals COBOL, C, etcetera. Bestaande programma's kunnen zo sterk vereenvoudigd worden en er kan gebruik worden gemaakt van bijvoorbeeld pointers en dynamisch geheugenbeheer.

1.5.4 Toegesneden op Oracle

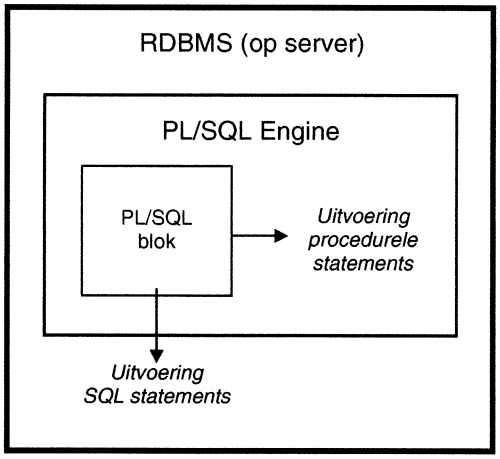
PL/SQL is door Oracle ontwikkeld en we kunnen niet anders verwachten dan dat deze taal toegesneden is op het Oracle RDBMS. PL/SQL lijkt afgezien van de procedurele statements erg veel op SQL. Het hanteert dezelfde functies, operatoren en gegevenstypen. Daarnaast kent het speciale attributen t.b.v. foutafhandeling van SQL statements.

PL/SQL is geen stand-alone programma dat je apart kunt aanroepen; het is een voorziening die kan worden toegevoegd aan een Oracle programma. Men spreekt van de PL/SQL Engine. De taak van de PL/SQL Engine is om ieder PL/SQL-blok in te lezen en daarna te ontrafelen in enerzijds de procedurele statements en anderzijds het SQL. De Engine verwerkt zelf de procedurele statements.

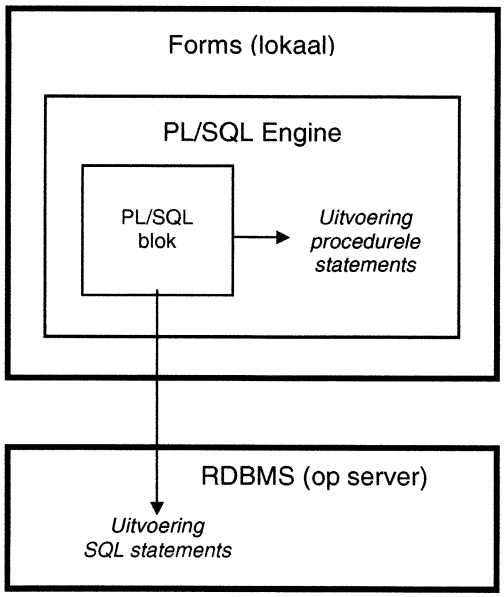
Op de server is altijd een PL/SQL Engine aanwezig. Deze draagt zorg voor de verwerking van PL/SQL dat bijvoorbeeld binnen SQL*Plus wordt gebruikt, en voor de procedures die in de database zijn opgeslagen. Op de client kan ook een PL/SQL Engine aanwezig zijn. In Forms kunnen bijvoorbeeld procedures en triggers worden gemaakt, die lokaal (op de client zelf) worden uitgevoerd. Alleen het SQL-gedeelte wordt dan op de server uitgevoerd.

Ter illustratie:

PL/SQL Engine
op de server:



PL/SQL Engine lokaal,
bijvoorbeeld binnen Forms:



Vóór Oracle9i gebruikt de PL/SQL Engine een andere SQL Parser dan die van de database. Gevolg hiervan is dat bepaalde mogelijkheden van SQL die wel vanuit SQL*Plus kunnen worden gebruikt nog niet vanuit PL/SQL gebruikt kunnen worden.
Vanaf versie 9i gebruikt de PL/SQL Engine echter dezelfde SQL Parser als de database. Hierdoor kunnen alle mogelijkheden van SQL ook vanuit PL/SQL worden gebruikt.

1.6 Voorbeeld

We gaan nu een voorbeeld bekijken, waarin het verschil tussen de 4e generatietaal SQL en de 3e generatietaal PL/SQL naar voren komt.

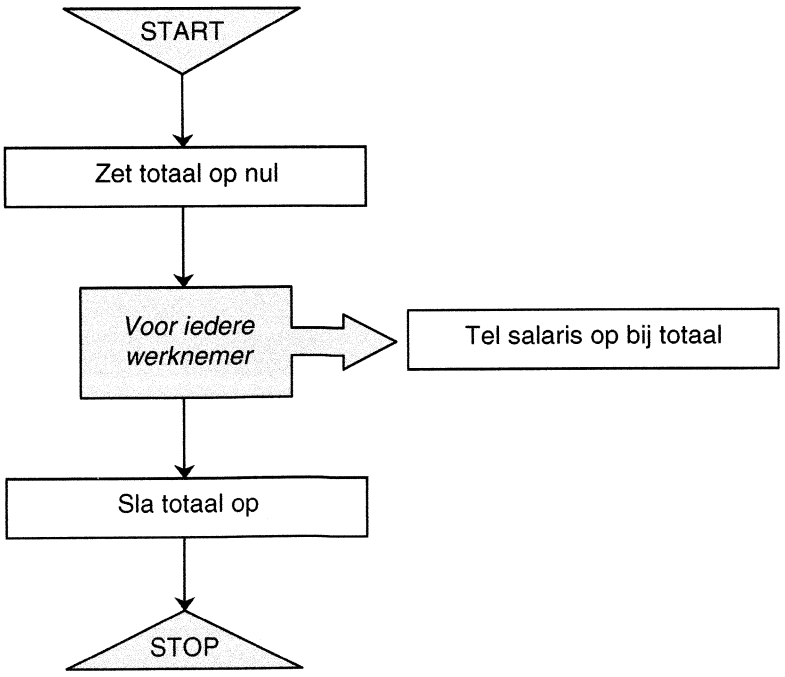
We berekenen het totaal van alle salarissen van de werknemers uit de tabel P_WERKNEMERS. Dit totaal plaatsen we vervolgens in de tabel SALARISTOTAAL. Deze tabel heeft de volgende structuur:

```
desc salaristotaal
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 TAAL                                               VARCHAR2(10)
 TOTAAL                                             NUMBER
```

In de 4e generatietaal SQL kunnen we de optelling vrij eenvoudig uitvoeren. We geven aan welke kolom we willen sommeren, waarna Oracle deze optelling zelf regelt. Het plaatsen van dit totaal in de tabel SALARISTOTAAL kunnen we in hetzelfde statement doen:

```
insert into salaristotaal
  select 'SQL'
  ,      sum(sal)
  from p_werknemers;
```

In plaats van dit ene statement, hadden we de optelling ook procedureel, in kleine stappen, kunnen uitvoeren. Van iedere werknemer wordt het salaris opgehaald, dit wordt bij elkaar opgeteld, en het resultaat wordt in de tabel gezet. We kunnen dit aan de hand van het volgende stroomschema illustreren:



In de loop van de cursus gaan we dieper in op de syntax. Hier kijken we alvast naar de globale structuur van een PL/SQL-blok:

```
declare
  v_totaal number;
  cursor c_sal is
    select sal
    from   p_werknemers;
begin
  v_totaal := 0;
  for r_sal in c_sal loop
    v_totaal := v_totaal + r_sal.sal;
  end loop;
  insert into salaristotaal
   values( 'PL/SQL'
         , v_totaal
         );
end;
/
```

Voor elke werknemer tellen we het salaris op bij het totaal. Het totaal wordt vervolgens in de tabel salaristotaal gezet.

Het blijkt dat in PL/SQL meer statements nodig zijn dan in SQL, om hetzelfde resultaat te bereiken. Dit is een kenmerkende eigenschap voor verschillende programmeertalen: hoe hoger de generatie, hoe minder code nodig is om iets te bereiken. Maar: hoe lager de generatie, hoe meer mogelijkheden er zijn om iets te bereiken.

Wanneer we nu de tabel SALARISTOTAAL bekijken, zien we dat SQL en PL/SQL tot hetzelfde resultaat zijn gekomen:

```
select *
from   salaristotaal;

   TAAL           TOTAAL
   ---------- ----------
 1 SQL             43525
 2 Totaal          43525
```

2 Basisbegrippen

2.1 Inleiding

Dit hoofdstuk beschrijft de structuur van een PL/SQL programma. Verder proberen we bij de naamgeving van de verschillende onderdelen zoveel mogelijk gebruik te maken van Oracle standaarden. Namen van variabelen laten we bijvoorbeeld altijd beginnen met voorvoegsel v_. In de loop van de cursus zullen we bij nieuwe theorieonderdelen de nieuwe naamgeving noemen.

2.2 Opbouw

Een programma in PL/SQL is volgens een bepaalde structuur opgebouwd. In het eenvoudigste geval begint een programma met BEGIN, vervolgens volgen één of meerdere statements, waarna END het programma afsluit. Elk statement wordt afgesloten door een puntkomma. Ook na het keyword END volgt een puntkomma, om het programma af te sluiten.

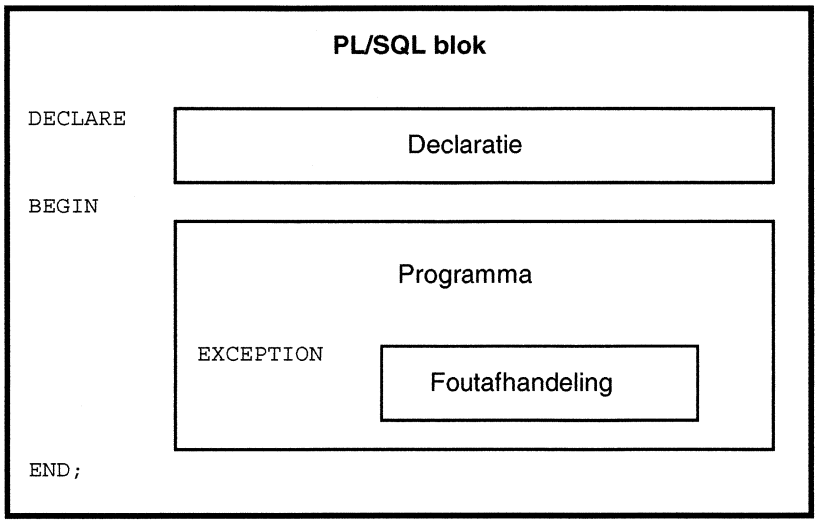
```
begin
  statement1;
  statement2;
end;
```

Met behulp van het keyword BEGIN wordt aangegeven dat er een *programmasectie* begint in PL/SQL. Naast de programmasectie bestaan er nog twee secties: de *declaratiesectie* en de *foutafhandelingssectie*. Deze hebben ook elk hun eigen keyword. De volgorde van de secties ligt vast. Dit laat zich door het volgende voorbeeld illustreren:

```
declare
  declaraties
begin
  statements
exception
  exceptions
end;
```

Alleen de programmasectie is verplicht en wordt altijd aangekondigd door het keyword BEGIN. Daarvóór kan een declaratiesectie komen, door het keyword DECLARE op te nemen. We komen hier in deze paragraaf nog op terug. Als laatste kan een foutafhandelingssectie binnen de programmasectie worden geplaatst, waarin fouten in het programmaverloop worden opgevangen. Dit wordt uitgebreid behandeld in hoofdstuk 6
Een PL/SQL programma zoals we hier hebben gezien, wordt een *PL/SQL-blok* genoemd. In een PL/SQL-blok kunnen eventueel weer andere PL/SQL-blokken worden opgenomen. Een PL/SQL-blok wordt pas uitgevoerd zodra een slash ('/') wordt gegeven na het END keyword. Deze slash moet als enige teken op een verder lege regel staan.

In de onderstaande figuur vindt u nogmaals de structuur van een PL/SQL-blok:



2.2.1 Statements

We gaan nu zelf een simpel PL/SQL-blok maken met alleen een programmasectie. In deze programmasectie plaatsen we twee statements. Omdat de meeste SQL statements ook in PL/SQL gebruikt mogen worden, mogen dit bijvoorbeeld INSERT statements zijn.

Omdat in PL/SQL geen standaard routine aanwezig is om uitvoer op het scherm te zetten, maken we nu eerst een hulptabel aan, waarin we gegevens kunnen plaatsen (zoals de uitvoer van een programma). Voer het volgende statement uit:

```
create table hulptabel
  ( kolom1 number
  , kolom2 varchar2(75)
  , kolom3 varchar2(75)
  );
```

Ons eerste PL/SQL programma voegt twee rijen toe aan de tabel HULPTABEL.
Let erop, dat elk statement met een puntkomma moet worden afgesloten. Hierbij is BEGIN geen statement, maar een keyword in het BEGIN ... END statement. Op de laatste regel wordt een slash ('/') gegeven, zodat het PL/SQL-blok onmiddellijk wordt uitgevoerd:

```
begin
  insert into hulptabel
    values ( 1
           , 'Tekst'
           , 'De eerste rij'
           );
  insert into hulptabel
    values ( 2
           , 2
           , 'De tweede rij'
           );
end;
/
```

Oracle geeft vervolgens aan dat de uitvoering van het PL/SQL-blok succesvol is verlopen. Wanneer een syntaxfout optreedt, geeft Oracle hier op een soortgelijke manier melding van als bij standaard SQL.

Wanneer we nu de tabel HULPTABEL raadplegen om te kijken wat het programma heeft uitgevoerd, zien we dat de twee rijen inderdaad zijn toegevoegd. In uw homedirectory (H:\) bevindt zich het bestandje HULP.SQL. Hiermee worden de gegevens van de HULPTABEL opgehaald en overzichtelijk weergegeven. Ook wordt de tabel na afloop leeggemaakt, zodat hij weer voor een volgende opdracht kan worden gebruikt.

Start het bestandje om het resultaat te bekijken (bekijk eventueel eerst het bestand om te kijken wat er wordt uitgevoerd):

```
@hulp.sql

   KOLOM1 KOLOM2                          KOLOM3
--------- ------------------------------ ------------------------------
        1 Tekst                          De eerste rij
        2 2                              De tweede rij
```

Merk op dat ook numerieke waarden in een varchar2 kolom kunnen worden ingevoerd. We zullen kolom2 van de hulptabel in deze cursus gebruiken om afwisselend numerieke en alfanumerieke waarden in te voeren.

Het INSERT statement, dat we in dit voorbeeld hebben gebruikt, is een standaard SQL statement. In hoofdstuk 4 komen meer statements aan de orde met betrekking tot het benaderen van de database. We zijn naast SQL statements natuurlijk ook geïnteresseerd in specifieke PL/SQL statements. We zullen hier alvast één voorbeeld van een PL/SQL statement zien: het NULL statement.

Het NULL statement doet niets. Het wordt vaak gebruikt ten behoeve van de leesbaarheid van een programma. De programmeur kan op deze manier in een programma aanduiden, dat hij expliciet de actie 'niets doen' bedoelt. Anders zou men misschien kunnen denken dat hij iets over het hoofd gezien heeft. Het NULL statement heeft overigens niets te maken met de waarde NULL.

Voorbeeld:

Een PL/SQL-blok mag géén lege programmasectie hebben. Het volgende PL/SQL-blok is daarom niet toegestaan:

```
begin
end;
/
```

Als we dit uitvoeren, zullen we zien dat Oracle hierop een foutmelding geeft. Wanneer we een PL/SQL-blok zouden willen maken dat niets uitvoert in de programmasectie, moeten we gebruik maken van het NULL statement. Hoewel dit statement niets doet, wordt het door Oracle wel als statement gezien, zodat het PL/SQL-blok correct niets uitvoert:

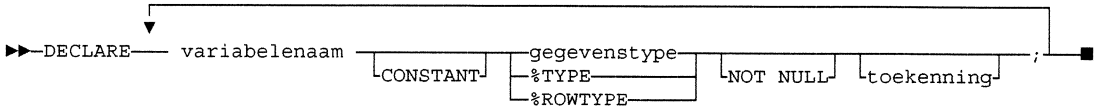
```
begin
  null;
end;
/
```

Natuurlijk kunnen we ook gebruik maken van het NULL statement om de structuur van een programma te bepalen, waarbij pas later de eigenlijke code wordt toegevoegd.

2.2.2 Declaraties

Eén van de grote voordelen van procedurele talen (3GL) is, dat gebruik kan worden gemaakt van variabelen. Hierin kan bijvoorbeeld tijdelijk een waarde worden opgeslagen. Een variabele kan in PL/SQL niet zonder meer worden gebruikt (wat bijvoorbeeld in BASIC wel kan). In PL/SQL moet iedere variabele eerst worden *gedeclareerd* (net zoals bijvoorbeeld in Pascal of C++). Het declareren van variabelen vindt plaats in de declaratiesectie.

Bij het declareren van een variabele moeten naam en gegevenstype worden opgegeven. De gegevenstypen van PL/SQL-variabelen komen (vrijwel) overeen met die van standaard SQL. In de declaratiesectie kunnen meerdere variabelen worden gedeclareerd. Deze declaraties worden gescheiden door puntkomma's (net zoals bij statements). Oracle herkent het einde van de declaratiesectie (en het begin van de programmasectie) aan het keyword BEGIN. De volledige syntax van DECLARE vinden we in het naslagwerk. We beperken ons hier tot een gedeelte ervan, zodat we variabelen van een bepaald type kunnen declareren:



De declaratiesectie van een PL/SQL-blok waarin twee numerieke variabelen worden gebruikt, kan er bijvoorbeeld als volgt uit zien:

```
declare
  v_getal  number;
  v_totaal number;
```

Wanneer bij de declaratie van een variabele het keyword CONSTANT wordt opgenomen, mag de waarde van deze variabele niet worden veranderd. Een directe waardetoekenning (in de declaratiesectie zelf) is verplicht. Bijvoorbeeld:

```
declare
  v_waarde constant varchar2(20) := 'test';
```

Wanneer een variabele als NOT NULL wordt gedeclareerd, gelden hiervoor dezelfde regels als voor een NOT NULL kolom uit een tabel: de variabele moet altijd een waarde hebben. Daarom is hier een directe waardetoekenning (in de declaratiesectie) dan verplicht.

Mogelijke gegevenstypen:

- VARCHAR2(n) — om karakterstrings van variabele lengte op te slaan. De maximale lengte is 32767 bytes;
- NUMBER(n[,m]) — voor numerieke velden, evt. gespecificeerd met grootte en decimalen. Voor de grootte wordt default waarde 38 genomen, wanneer we de grootte weglaten.

  Voorbeeld:

  ```
  declare
    v_teller number(5,2);
  ```

- DATE — voor datumvelden;

- CHAR(n) — om karakterstrings met een vaste lengte op te slaan. Zonder n wordt een default lengte van 1 aangenomen. De maximale lengte is 32767 bytes (in tegenstelling tot de maximale lengte 2000 van een CHAR kolom in een tabel - vanaf Oracle8);
- LONG — om karakterstrings van variabele lengte op te slaan. Is identiek aan VARCHAR2 met het verschil dat de maximale lengte van een LONG variabele 32760 bytes is. Let op dat data in een databasekolom van type LONG een lengte van maximaal 2 Gb kan hebben en dus niet zomaar in PL/SQL kan worden ingelezen LONG is een verouderd datatype gebruik over CLOB;
- BOOLEAN — voor Booleaanse velden: kan de waarde TRUE, FALSE of NULL krijgen. Het zijn afwijkende variabelen, we kunnen ze niet vullen vanuit de database en evenmin wegschrijven naar de database;
- RAW — om binaire data of bytestrings op te slaan. Maximale lengte is 2000 bytes. Dit datatype wordt gebruikt voor plaatjes. RAW data kunnen door PL/SQL niet geïnterpreteerd worden;
- LONG RAW — om binaire data of bytestrings op te slaan. Maximale lengte is 2 gigabytes. LONG RAW data kunnen door PL/SQL niet geïnterpreteerd worden. Ook LONG RAW is een verouderd datatype, beter kunt u BLOB gebruiken;
- %TYPE — neemt het gegevenstype over van een eerder gedeclareerd veld of een kolom uit de database. Plaats de veldnaam c.q. kolomnaam als voorvoegsel.

Voorbeeld:

```
declare
  v_naam         varchar2(20);
  v_naam_type1   v_naam%type;
  v_naam_type2   p_werknemers.naam%type;
begin
  v_naam        := 'Maximaal 20 posities';
  v_naam_type1  := 'Maximaal 20 posities';
  v_naam_type2  := 'Maximaal 20 posities';
end;
```

- %ROWTYPE — voor structuren: neemt de gegevenstypen over van de kolommen uit de database die samen een record vormen. Plaats de tabel- of viewnaam als voorvoegsel. Een tevoren gedeclareerde cursornaam mag ook, dit zien we later nog terug. Deze variabelen mogen niet beschouwd worden als één veld; het zijn samengestelde velden ofwel STRUCTURES. We kunnen wel de afzonderlijke velden eruit lichten, namelijk door de veldnaam als extensie mee te geven.

Voorbeeld:

```
declare
  r_kantoren p_kantoren%rowtype;
begin
  r_kantoren.kantnr := 11;
  r_kantoren.naam   := 'Test';
  r_kantoren.plaats := 'Utrecht';
  insert into p_kantoren values
    ( r_kantoren.kantnr , r_kantoren.naam , r_kantoren.plaats);
end;
/
```

– CLOB	Een groot object dat alleen karakters (CHAR) bevat. In Oracle 9i kan een CLOB maximaal 4 gigabyte worden. Vanaf Oracle 10g kan dit maximaal 128 terabyte worden.
– BLOB	Een groot binair object zoals een afbeelding of een Worddocument. In Oracle 9i kan een BLOB maximaal 4 gigabyte worden. Vanaf Oracle 10g kan dit maximaal 128 terabyte worden.
– BFILE	Bevat een verwijzing naar een BLOB. Deze BLOB is dan niet in de database opgeslagen, maar door de verwijzing is het wel mogelijk uit de file te lezen en ernaar te schrijven.
– BINARY_FLOAT	Nieuw datatype in Oracle10*g*. Gebroken getallen worden 32-bits opgeslagen met binaire precisie (de cijfers 0 en 1). Daardoor kunnen rekenkundige taken veel sneller uitgevoerd worden in vergelijking met het gebruik van NUMBER. BINARY_FLOAT ondersteunt ondermeer NaN (not a number) en oneindig.
– BINARY_DOUBLE	Nieuw datatype in Oracle10*g*. Vergelijkbaar met BINARY_FLOAT, behalve dat dit datatype 64-bits is.
– INTEGER	synoniem voor voor numerieke gehele getallen.
– PLS_INTEGER	kan gehele getallen bevatten tussen -2147483648 en 2147483647 of leeg (NULL) zijn.
– SIMPLE_INTEGER	kan gehele getallen tussen -2147483648 en 2147483647 bevatten, maar mag niet leeg zijn. Dit datatype bestaat alleen in Oracle11*g*, SIMPLE_INTEGER is een subtype van PLS_INTEGER.

Merk verder op dat de naam van een variabele begint met v_ en dat de naam van een rijvariabele begint met r_. Dat is niet verplicht maar bevorderd de leesbaarheid.

De volgorde waarin variabelen worden gedeclareerd is van belang wanneer een variabele in de declaratie van een andere variabele wordt gebruikt (denk bijvoorbeeld aan het type %TYPE). Bij de onderstaande declaratie geeft Oracle een foutmelding, omdat V_NAAM door V_NAAM_TYPE wordt gebruikt en dus eerder gedeclareerd dient te worden (we laten voor het gemak het PL/SQL-blok even niets uitvoeren).

```
declare
  v_naam_type v_naam%type;
  v_naam      varchar2(20);
begin
  null;
end;
/
Error starting at line 1 in command:
declare
  v_naam_type v_naam%type;
  v_naam      varchar2(20);
begin
  null;
end;
Error report:
ORA-06550: line 2, column 15:
PLS-00320: the declaration of the type of this expression is incomplete or
malformed
ORA-06550: line 2, column 15:
PL/SQL: Item ignored
06550. 00000 -  "line %s, column %s:\n%s"
*Cause:    Usually a PL/SQL compilation error.
*Action:
```

Kijk eventueel in uw PL/SQL Reference naar een uitgebreide uitleg van de datatypen. In andere versies van de Oracle database kan namelijk de maximale lengte van de verschillende datatypen afwijken.

2.2.3 Toekenningen

Het toekennen van een waarde aan een variabele noemt men een *assignment*. In PL/SQL kan dit op verschillende manieren gebeuren:

- variabele_naam := waarde;
- het SELECT INTO statement
- het FETCH INTO statement

Het SELECT INTO en het FETCH INTO statement zullen in hoofdstuk 4 aan de orde komen. Hier kijken we naar de directe toekenning van een waarde aan een variabele met behulp van ':=' (dubbele punt + isgelijkteken).

Voorbeeld:
We maken een PL/SQL-blok met een INSERT statement, dat een rij aan de tabel HULPTABEL toevoegt. Voor de waarden die in het INSERT statement worden gebruikt, maken we drie variabelen aan: twee van het type NUMBER, en één van het type VARCHAR2:

```
declare
  v_getal1 number;
  v_getal2 number;
  v_tekst  varchar2(20);
begin
  v_getal1 := 3;
  v_getal2 := v_getal1;
  v_tekst  := 'Dit zijn variabelen';
  insert into hulptabel
    values ( v_getal1
           , v_getal2
           , v_tekst
           );
end;
/
```

Nadat we dit script uitvoeren ziet de inhoud van de HULPTABEL er als volgt uit:

```
   KOLOM1 KOLOM2                         KOLOM3
--------- ------------------------------ ------------------------------
        3 3                              Dit zijn variabelen
```

Probeer variabelen altijd een zinvolle naam te geven.

2.2.4 Sequences

Vanaf Oracle11*g* is het mogelijk om in PL/SQL een variabele direct te vullen met de NEXTVAL waarde van een sequence. In vorige versies van Oracle moest hiervoor altijd een constructie gebruikt worden met een SELECT INTO statement of een expliciete cursor. In hoofdstuk 4 komt het gebruik van cursoren en het SELECT INTO statement uitgebreid aan de orde.

Voorbeeld met SELECT INTO:

```
declare
  v_getal number;
begin
select pers_seq.nextval
into   v_getal
from   dual;
insert into hulptabel
values (v_getal
       ,null
       ,null
       );
end;
/
```

Voorbeeld met directe toekenning:

```
declare
  v_getal number;
begin
v_getal:= pers_seq.nextval;
insert into hulptabel
values (v_getal
       ,null
       ,null
       );
end;
/
```

2.2.5 %TYPE en %ROWTYPE

De bijzondere gegevenstypen %TYPE en %ROWTYPE nemen het type over van een kolom of een rij uit de database. Vóór de gegevenstypen moet de naam van de kolom of tabel worden geplaatst. Deze manier van werken garandeert dat een variabele in een PL/SQL-blok van hetzelfde type is als een kolom van een tabel in de database.
Omdat %ROWTYPE een complete rij in één variabele kan definiëren, kunnen de afzonderlijke kolommen daarvan worden geselecteerd met behulp van de notatie VARIABELENAAM.KOLOMNAAM (met een punt er tussen).

Voorbeeld:

We declareren twee variabelen: V_PERSNR is van hetzelfde type als de kolom PERSNR uit tabel P_WERKNEMERS, en R_WERKNEMERS is van %ROWTYPE van een hele rij uit dezelfde tabel. Vervolgens kennen we waarden toe aan de variabelen en voegen we een rij toe aan HULPTABEL.

```
declare
  v_persnr     p_werknemers.persnr%type;
  r_werknemers p_werknemers%rowtype;
begin
  v_persnr            := 1000;
  r_werknemers.persnr := v_persnr;
  r_werknemers.naam   := 'Wolters';
  r_werknemers.sal    := 3000;
  insert into hulptabel values
    ( r_werknemers.persnr
    , r_werknemers.sal
    , r_werknemers.naam
    );
end;
/
```

Nadat we dit script uitvoeren ziet de inhoud van de HULPTABEL er als volgt uit:

```
   KOLOM1 KOLOM2                          KOLOM3
--------- ------------------------------ ------------------------------
     1000 3000                           Wolters
```

In dit voorbeeld neemt de variabele V_PERSNR het gegevenstype over van P_WERKNEMERS.PERSNR (NUMBER). De variabele R_WERKNEMERS wordt een rij, waarin alle kolommen uit de tabel P_WERKNEMERS zijn vertegenwoordigd. In het voorbeeld vullen we de componenten PERSNR, NAAM en SAL van deze variabele.

In PL/SQL bestaat nog een gegevenstype, dat niet in SQL gebruikt kan worden: BOOLEAN. Een booleaanse variabele kan slechts drie waarden bevatten: TRUE, FALSE of NULL. Een variabele van dit type is dus altijd waar, onwaar of leeg. Het resultaat van een voorwaarde (bijvoorbeeld in een conditie) is altijd van het type BOOLEAN. Variabelen van dit type kunnen niet in een tabel worden gezet, of uit een tabel worden gelezen.

2.2.6 Expressies

Een *expressie* is een omschrijving voor een waarde. In het eenvoudigste geval bestaat een expressie uit één constant getal; de waarde zelf. Expressies kunnen onder andere worden gebruikt bij waardetoekenningen (assignments).

In expressies mogen zowel constanten als variabelen worden gebruikt. Bovendien kunnen in PL/SQL voor expressies dezelfde operatoren en functies worden gebruikt als in SQL.

Voorbeeld:

```
declare
  v_getal1 number;
  v_getal2 number := 10;
  v_dag    varchar2(10);
begin
  v_getal1 := v_getal2 * 5;
  v_dag    := to_char(sysdate, 'Day');
  insert into hulptabel
    values ( v_getal1
           , v_getal2
      , v_dag
           );
end;
/
```

Nadat we dit script uitvoeren ziet de inhoud van de HULPTABEL er als volgt uit:

```
   KOLOM1 KOLOM2                          KOLOM3
--------- ------------------------------ ------------------------------
       50 10                             Wednesday
```

2.3 Commentaar

Ter verduidelijking van de programmacode kan in een programma commentaar worden opgenomen. Dit kan op twee manieren gebeuren: met één regel tegelijk of uitgesmeerd over meerdere regels.

In het eerste geval kunnen we een dubbel minteken gebruiken.

Voorbeeld:

```
begin
                              --hier begint de programmasectie
  delete p_werknemers;        --de werknemerstabel wordt leeggemaakt
  rollback ;                  --de verwijderingen worden weer ongedaan gemaakt
end;
```

Dit type commentaar loopt door tot het einde van de regel; alles wat zich rechts van de commentaarstrepen bevindt, wordt niet als programmeercode gezien.

Het tweede type geeft meer mogelijkheden: we mogen het overal plaatsen waar maar een spatie kan staan. Ook mag de tekst over meerdere regels verspreid liggen. De commentaartekst wordt aan de linkerkant begonnen met een /* en aan de rechterkant afgesloten met */. Alles wat tussen deze delimiters ligt wordt als commentaar opgevat.

Voorbeeld:

```
/* Dit PL/SQL-blok begint met commentaar */
/* Commentaar mag, bij deze notatie, ook
   over meerdere regels worden uitgesmeerd.
   Als het maar begint met slash-sterretje,
   en eindigt met sterretje-slash. */
begin
                /* hier begint de programma-sectie */
  delete        /* We maken nu de werknemerstabel leeg */   p_werknemers;
  rollback;     /* de verwijderingen worden
                   weer ongedaan gemaakt. */
end;
/
```

De code `delete p_werknemers` zal dus gewoon worden uitgevoerd.

2.4 NULL

Het NULL statement doet niets. Het wordt vaak gebruikt ten behoeve van de leesbaarheid van een programma. De programmeur kan op deze manier in een programma aangeven, dat hij expliciet de actie 'niets doen' bedoelt. Anders zou men misschien kunnen denken dat hij iets over het hoofd gezien heeft. Het NULL statement heeft overigens niets te maken met de waarde NULL.

Voorbeeld:

```
declare
 v_datum date;
 v_melding varchar2(100);
begin
v_datum:=to_date('01-JAN-2010','DD-MON-YYYY');
 if   sysdate < v_datum
 then v_melding:='Het is nog geen 2010';
 else null;
 end if;
end;
/
```

Natuurlijk kunnen we ook gebruik maken van het NULL statement om de structuur van een programma te bepalen, waarbij pas later de eigenlijke code wordt toegevoegd.

3 PL/SQL Syntax

3.1 Inleiding

In het vorige hoofdstuk is de structuur van een PL/SQL programma beschreven. In dit hoofdstuk worden de mogelijkheden met betrekking tot bijvoorbeeld condities en lussen beschreven.

Verder proberen we bij de naamgeving van de verschillende onderdelen zoveel mogelijk gebruik te maken van Oracle standaarden. Namen van variabelen laten we bijvoorbeeld altijd beginnen met voorvoegsel v_. In de loop van de cursus zullen we bij nieuwe theorieonderdelen de nieuwe naamgeving noemen.

Leerdoelen:

- Verschillende mogelijkheden met betrekking tot condities en lussen kennen
- Veel voorkomende PL/SQL foutmeldingen herkennen

3.2 Condities

We kunnen met behulp van condities regelen welke statements moeten worden uitgevoerd. Wanneer aan de opgegeven conditie wordt voldaan, worden de statements uitgevoerd. Ook is het mogelijk om aan te geven welke statements dan wel moeten worden uitgevoerd.

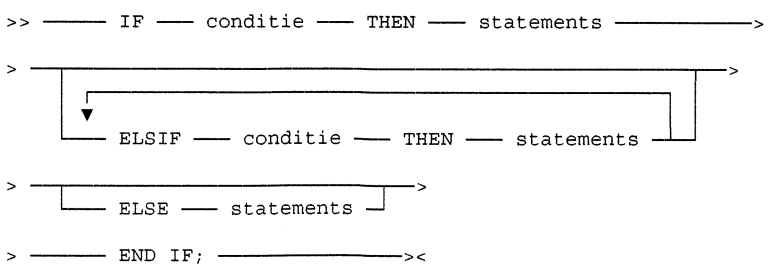
We hebben drie mogelijkheden:
- het IF-THEN-ELSE statement,
- het CASE-statement (vanaf Oracle9*i*),
- de CASE-expressie (vanaf Oracle9*i*).

In deze paragraaf zullen we ze alle drie behandelen.

3.2.1 IF-THEN-ELSE statement

Met behulp van het IF-THEN-ELSE statement kunt u regelen dat een aantal acties onder een bepaalde voorwaarde wordt opgestart zodra de voorwaarde TRUE oplevert. Het IF-THEN-ELSE statement kan vertakt worden met de clauses ELSIF en ELSE.

Syntax:



In tegenstelling tot enkele andere talen, maken we bij PL/SQL gebruik van ELSIF in plaats van ELSEIF en wordt END IF niet aan elkaar geschreven.

Na het THEN keyword worden de statements opgenomen die moeten worden uitgevoerd wanneer aan de conditie is voldaan. Deze statements worden op de normale manier gescheiden; met behulp van een puntkomma.

De uitkomst van een vergelijking is TRUE of FALSE en in sommige uitzonderingsgevallen NULL. Om compatibel te zijn met SQL geldt in PL/SQL dat NULL gelijk is aan FALSE.

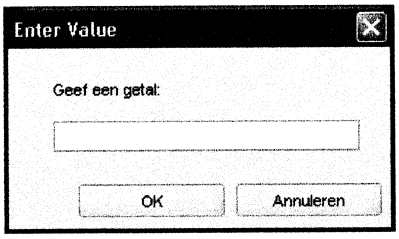
Een vergelijking (conditie) test een waarde op grond van een criterium. Dat criterium kan zijn:
- een logische operator (< , <= , = , >= , > , !=) gevolgd door een expressie;
- LIKE, een expressie die een stringpatroon oplevert;
- IS NULL of IS NOT NULL.

De uitkomst van een conditie met AND is FALSE zodra één der leden aan weerszijden de uitkomst FALSE of NULL heeft. De uitkomst van een conditie met OR is TRUE zodra één der leden aan weerszijden van OR de uitkomst TRUE heeft. U mag ronde haakjes gebruiken om de volgorde van afwerking te beïnvloeden.

We gaan nu een PL/SQL-blok maken waarin wordt bepaald of een variabele V_GETAL negatief is. Om de waarde van de variabele V_GETAL op te vragen, kunnen we gebruik maken van het SQL-commando ACCEPT. Dit commando levert een zogenaamde SQL-variabele op, waaraan in een PL/SQL-blok gerefereerd kan worden, door er een '&' (ampersand) voor te zetten Let erop, dat het ACCEPT commando geen PL/SQL is, en dus ook geen deel uitmaakt van het PL/SQL-blok. Voer met behulp van Run Script de onderstaande code uit in een SQL Worksheet:

```
accept getal number prompt 'Geef een getal: '
```

Vervolgens zal het volgende scherm verschijnen:



Geef hier bijvoorbeeld het getal –4 in, dit getal wordt vervolgens opgeslagen in de SQL variabele GETAL.

Het ACCEPT commando is geen PL/SQL code en kan dus ook geen deel uitmaken van het PL/SQL-blok

Wanneer u de code meerdere malen wilt uitvoeren met verschillende waarden voor de variabele GETAL dan moet u het ACCEPT commando opnieuw ingeven.

Wanneer we in dezelfde worksheet vervolgens onderstaande PL/SQL blok uitvoeren, dan zal dit blok de inhoud van de SQL variabele GETAL gebruiken.

```
declare
  v_getal number := &getal;                    -- v_getal krijgt waarde uit SQL-variabele getal
begin
  if v_getal < 0 then                          -- Alleen als v_getal negatief is
    insert into hulptabel
      values ( v_getal
             , null
             , 'Negatief!'
             );
  end if;                                      -- Afsluiten van het if statement
end;
/
```

Wanneer we het script uitvoeren wordt de onderstaande rij toegevoegd aan de tabel HULPTABEL:

```
   KOLOM1 KOLOM2                          KOLOM3
--------- ------------------------------ ------------------------------
       -4                                Negatief!
```

Probeer het voorbeeld zelf nogmaals uit, met een positief getal.

Onderstaand programma bevat een voorbeeld van een genest IF statement. Het programma controleert of een getal negatief, positief of gelijk aan nul is. Probeer het weer met verschillende getallen uit.

```
declare
  v_getal number := &getal;
begin
  if v_getal < 0 then
    insert into hulptabel
      values (v_getal
             , null
             , 'Negatief!'
             );
  else
    if v_getal = 0 then
      insert into hulptabel
        values (v_getal
               , null
               , 'Nul!'
               );
    else
      insert into hulptabel
        values (v_getal
               , null
               , 'Positief!'
               );
    end if;
  end if;
end;
/
```

We voeren het bovenstaande script uit en geven als getal 13 op, vervolgens runnen we het script nogmaals, maar nu met het getal 0. Het resultaat in de tabel HULPTABEL zal er nu alsvolgt uitzien:

```
      KOLOM1 KOLOM2                          KOLOM3
   --------- ------------------------------ ------------------------------
          13                                Positief!
           0                                Nul!
```

Het voorbeeld met het geneste IF statement kunnen we ook uitvoeren met behulp van ELSIF. Het ziet er dan overzichtelijker uit.

met ELSE en een nieuwe IF-conditie:

```
declare
  v_getal number := &getal;
begin
  if v_getal < 0 then
    insert into hulptabel
      values (v_getal
             , null
             , 'Negatief!'
             );
  else
    if v_getal = 0 then
      insert into hulptabel
        values (v_getal
               , null
               , 'Nul!'
               );
    else
      insert into hulptabel
        values (v_getal
               , null
                 , 'Positief!'
               );
    end if;
  end if;
end;
/
```

met alleen een ELSIF-conditie:

```
declare
  v_getal number := &getal;
begin
  if v_getal < 0 then
    insert into hulptabel
      values ( v_getal
             , null
             , 'Negatief!'
             );
  elsif v_getal = 0 then
    insert into hulptabel
      values ( v_getal
             , null
             , 'Nul!'
             );
  else
    insert into hulptabel
      values ( v_getal
             , null
             , 'Positief!'
             );
  end if;
end;
/
```

Merk op dat er een END IF keyword minder nodig is. Dit komt, doordat er nu slechts één IF statement wordt gebruikt, met daarin een ELSIF tak. Binnen een IF statement mag maar één ELSE tak en mogen meerdere ELSIF takken worden gemaakt.

Zoals we hebben gezien, staan voor de conditie de Booleaanse operatoren AND, OR en NOT ter beschikking. De uitkomst van een conditie met AND is FALSE zodra één der leden aan weerszijden de uitkomst FALSE of NULL heeft. De uitkomst van een conditie met OR is TRUE zodra één der leden aan weerszijden van OR de uitkomst TRUE heeft. U mag ronde haakjes gebruiken om de volgorde van afwerking te beïnvloeden.

Voorbeeld:

```
declare
  v_getal number := &getal;
begin
  if v_getal = trunc(v_getal)
    and (v_getal <= 0 or v_getal >= 10)
    and v_getal not between 40 and 60 then
    insert into hulptabel
      values (v_getal
             , null
             , 'Het getal voldoet'
             );
  end if;
end;
/
```

Dit voorbeeld controleert of het getal geheel is en of het niet ligt tussen 0 en 10 en niet ligt tussen 40 en 60. Let op de haakjes vanwege de OR operator!

3.2.2 CASE-statement (9*i*)

Het CASE-statement is een verkorte weergave van het IF-THEN-ELSE statement. Er bestaan twee soorten: simple en searched. Bij een simple CASE-statement wordt gebruik gemaakt van een selector in plaats van een conditie die TRUE of FALSE als resultaat heeft. Een searched CASE-statement maakt wel gebruik van condities die TRUE of FALSE als resultaat moeten hebben.

Als eerste gaan we het simple CASE-statement bespreken, daarna de searched uitvoering.

3.2.2.1 Simple

Het simple CASE-statement begint met het keyword CASE. Daarna volgt de selector. De selector is gewoonlijk een enkele variabele, maar kan ook een expressie zijn die de waarde bepaalt. Wanneer dat laatste het geval is, dan wordt de expressie éénmalig geëvalueerd om de waarde ervan te bepalen.

Na de selector volgen één of meerdere WHEN-clausules. Deze worden opeenvolgend geëvalueerd. Hierbij wordt de waarde van de selector vergeleken met de waarde die na WHEN wordt genoemd. Wanneer beide gelijk zijn, dan wordt het statement uitgevoerd dat na de bijbehorende THEN staat. Vervolgens gaat het programma verder na de END van het CASE-statement. Een CASE-statement kan één ELSE-clausule bevatten. Deze staat onder de laatste WHEN-clausules. Deze werkt vrijwel gelijk aan de ELSE in het IF-THEN-ELSE statement. En tenslotte volgt de END CASE.

Let erop dat ELSE verplicht moet worden opgegeven wanneer de selector niet gelijk is aan één van de opgegeven expressies. Wordt er in zo'n geval geen ELSE opgegeven, dan verschijnt de foutmelding `ORA-06592: CASE not found while executing CASE statement.` Bij een IF-THEN-ELSE constructie is de ELSE wel altijd optioneel.

Syntax:

```
case selector
  when expressie1 then
    statement;
  when expressie2 then
    statement;
  ...
  when expressieN then
    statement;
  [else
    statement;]
end case;
```

Voorbeeld simple CASE-statement:

```
case v_functie
  when 'Klerk' then
    v_verhoging := 1.02;
  when 'Verkoper' then
    v_verhoging := 1.06;
  else
    v_verhoging := 1.08;
end case;
```

In dit voorbeeld is de waarde van de variabele V_FUNCTIE de selector. Als eerste wordt deze vergeleken met de waarde 'Klerk'. Wanneer deze vergelijking klopt (V_FUNCTIE = 'Klerk'), wordt het bijbehorende statement uitgevoerd en krijgt de variabele V_VERHOGING de waarde 1.02. Is dit niet het geval dan wordt de volgende WHEN-clause geëvalueerd. Wanneer dan geldt dat V_FUNCTIE de waarde 'Verkoper' heeft, krijgt V_VERHOGING de waarde 1.06. Wanneer beide WHEN-clauses niet voldoen wordt het statement in de ELSE-clause uitgevoerd: V_VERHOGING krijgt de waarde 1.08. Als laatste wordt het CASE-statement afgesloten.

Hieronder staat nog eens het statement uit het hiervoor staande voorbeeld, maar nu in een IF-THEN-ELSE variant:

```
if v_functie = 'Klerk' then
  v_verhoging := 1.02;
elsif v_functie = 'Verkoper' then
  v_verhoging := 1.06;
else
  v_verhoging := 1.08;
end if;
```

3.2.2.2 *Searched*

Naast het simple CASE-statement bestaat ook het searched CASE-statement. Daarbij wordt na het keyword CASE geen selector geplaatst, maar komen direct één of meerdere WHEN-clauses. Elke WHEN-clause bevat een conditie. Wanneer deze TRUE oplevert wordt uitgevoerd wat na de bijbehorende THEN is opgenomen. Na de WHEN-clauses volgt eventueel de ELSE en als laatste END CASE.

Syntax:

```
case
  when conditie1 then
    statement;
  when conditie2 then
    statement;
  ...
  when conditieN then
    statement;
  [else
    statement;]
end case;
```

Voorbeeld searched CASE-statement:

```
declare
  v_begroeting varchar2(20);
begin
  case
    when to_char(sysdate,'hh24') < 6  then
      v_begroeting := 'Het is nacht';
    when to_char(sysdate,'hh24') < 12 then
      v_begroeting := 'Goedemorgen';
    when to_char(sysdate,'hh24') < 18 then
      v_begroeting := 'Goedemiddag';
    else
      v_begroeting := 'Goedenavond';
  end case;
  insert into hulptabel
    values
      ( null
      , v_begroeting
      , null
      );
end;
/
```

Nadat we dit script uitvoeren ziet de inhoud van de HULPTABEL er als volgt uit:

```
   KOLOM1 KOLOM2                         KOLOM3
--------- ------------------------------ ------------------------------
          Goedemiddag
```

3.2.3 CASE-expressie (9*i*)

Naast het CASE-statement is er ook de CASE-expressie. Dit is een manier om te bepalen welke waarde een variabele gaat krijgen zonder gebruik te hoeven maken van een IF-THEN-ELSE constructie of een CASE-statement. Hier is op dezelfde manier als bij het CASE-statement onderscheid te maken tussen simple en searched.

De opbouw van een CASE-expressie is vrijwel gelijk aan die van een CASE-statement. Binnen de expressie mag echter geen ';' worden gebruikt en aan het einde moet worden afgesloten met END in plaats van END CASE. Het verschil tussen statement en expressie is dat het statement zelfstandig is en de expressie links of rechts van een operator moet staan.

Voorbeeld simple CASE-expressie:

De variabele V_VERHOGING moet een waarde krijgen afhankelijk van de functie die iemand heeft. Dit kunnen we realiseren met behulp van de volgende IF-THEN-ELSE constructie en/of CASE-statement:

IF-THEN-ELSE statement::

```
if functie = 'Klerk' then
  v_verhoging := 1.02;
elsif functie = 'Verkoper' then
  v_verhoging := 1.06;
else
  v_verhoging := 1.08;
end if;
```

CASE-statement:

```
case v_functie
  when'Klerk' then
    v_verhoging := 1.02;
  when'Verkoper' then
    v_verhoging := 1.06;
  else
    v_verhoging := 1.08;
end case;
```

We kunnen dit ook met behulp van een CASE-expressie doen:

```
v_verhoging := case v_functie
                 when 'Klerk' then
                   1.02
                 when 'Verkoper' then
                   1.06
                 else
                   1.08
               end;
```

Voorbeeld searched CASE-expressie:

```
v_begroeting := case
                  when to_char(sysdate,'hh24') < 6  then
                    'Het is nacht'
                  when to_char(sysdate,'hh24') < 12 then
                    'Goedemorgen'
                  when to_char(sysdate,'hh24') < 18 then
                    'Goedemiddag'
                  else
                    'Goedenavond'
                end;
```

Het gebruik van CASE-statements of - expressies levert vaak leesbaardere en efficiëntere code op.

3.3 Herhalingen

Programmalussen ofwel iteraties zijn bedoeld om een reeks van statements een aantal malen te herhalen. In PL/SQL hebben we daarvoor verschillende mogelijkheden:

- Handmatige lus met behulp van sprongen
- WHILE LOOP met voorwaarde
- FOR LOOP met index
- FOR LOOP met cursor
- Ongelimiteerde LOOP

In deze paragraaf zullen we de verschillende mogelijkheden bespreken.

3.3.1 Sprongen maken met GOTO

In PL/SQL kan met behulp van het GOTO statement naar een willekeurige plaats in het programma worden gesprongen. Deze plaats moet worden aangegeven met behulp van een *label*. De syntax luidt als volgt:

```
>> ── GOTO ── label_naam ──><
```

Een label wordt gedefinieerd door een naam te omgeven met dubbele schuine haakjes. Labels moeten altijd op een aparte regel komen te staan. Een label mag nooit direct gevolgd worden door **end;**. Er moet minstens één statement staan tussen een label en het einde van een PL/SQL blok.

Met het GOTO statement wordt de reeks statements uitgevoerd die onder het label hangt. Labels moeten uniek zijn voor het huidige blok, maar niet voor geneste blokken (zie paragraaf 3.4). Wanneer twee labels dezelfde naam hebben zal er gesprongen worden naar het dichtstbijzijnde label.

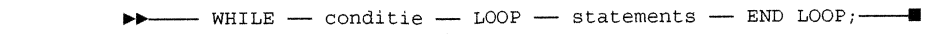
Voorbeeld:
Met labels kunnen we dus handmatig een lus maken. We plaatsen bijvoorbeeld de getallen 1 tot en met 10 in de tabel HULPTABEL, door telkens met behulp van een IF statement te controleren of het huidige getal nog kleiner of gelijk is aan 10. Zo ja, dan wordt teruggesprongen, anders is het programma klaar:

```
declare
  v_teller number := 1;
begin
  <<hierheen>>
  insert into hulptabel
    values ( v_teller
           , null
           , null
           );
  v_teller := v_teller + 1;
  if v_teller <= 10 then
    goto hierheen;
  end if;
end;
/
```

Door met sprongen te werken, is het vaak moeilijk om de structuur in een programma te ontdekken. Wanneer het gebruik van het GOTO statement niet persé noodzakelijk is, kan het beter niet gebruikt worden.

3.3.2 WHILE lussen

Een WHILE-lus is een herhaling die net zolang door blijft gaan, totdat niet meer aan de conditie wordt voldaan. De syntax ziet er als volgt uit:



Merk op, dat dit statement, net zoals het IF statement, moet worden afgesloten, in dit geval met behulp van END LOOP.

Voorbeeld 1:

Het volgende PL/SQL-blok berekent de machten van 2, kleiner dan 1000.

```
declare
  v_getal  number(4) := 1;
  v_cyclus number(2) := 0;
begin
  while v_getal < 1000  loop
    insert into hulptabel
      values ( v_getal
             , 'Twee tot de macht '||to_char(v_cyclus)
             , null
             ) ;
    v_getal := v_getal * 2 ;
    v_cyclus := v_cyclus + 1 ;
  end loop;
end;
/
```

Vergeet bij het eerste voorbeeld niet de toekenningen van V_GETAL en V_CYCLUS. Als V_GETAL geen waarde heeft wordt de hele lus overgeslagen. Het ophogen van V_CYCLUS kan ook alleen als deze een (initialisatie-)waarde heeft.

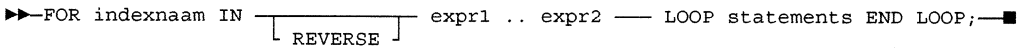
Voorbeeld 2:

Het volgende PL/SQL blok vult de tabel HULPTABEL met de getallen kleiner of gelijk aan 50, waarvan de wortel groter is dan 5.

```
declare
  v_teller number(3);
begin
  v_teller := 50;
  while sqrt(v_teller) > 5 loop                    --Zolang de wortel van teller groter is dan 5
    insert into hulptabel
      values ( v_teller
             , round(sqrt(v_teller),2)
             , 'De wortel is groter dan 5'
             );
    v_teller:=v_teller-1;
  end loop;
end;
/
```

3.3.3 FOR lussen met index

Een FOR-loop met index wordt automatisch gestuurd door een teller (de *index*). Er is geen expliciete variabele nodig: de gebruikte teller hoeft niet gedeclareerd te worden. Alleen de begin- en de eindwaarde worden opgegeven: de teller wordt dan telkens automatisch met één opgehoogd.
De syntax van het FOR statement ziet er als volgt uit (expr staat voor expressie):



De indexnaam kan worden gebruikt als variabele binnen de loop, maar hoeft *niet* gedeclareerd te worden. Het gegevenstype ervan is automatisch numeriek.

De expressies EXPR1 en EXPR2 geven respectievelijk de ondergrens en de bovengrens aan voor de index. Dit moeten beide numerieke waarden zijn. De expressies worden berekend voordat aan de herhaling wordt begonnen. Tussen de beide expressies moeten twee puntjes staan. Als ophoogwaarde wordt altijd automatisch 1 gebruikt: een andere ophoogwaarde is niet mogelijk.

Het optionele keyword REVERSE geeft aan dat de lus 'achterstevoren' doorlopen moet worden: de index krijgt in het begin de waarde van EXPR2, en wordt telkens met 1 verlaagd, totdat de waarde van EXPR1 is bereikt.

Voorbeeld:

Hetzelfde voorbeeld met de getallen 1 tot en met 10 kunnen we met behulp van een FOR-loop heel eenvoudig uitvoeren. De teller (ook index genoemd) hoeft namelijk niet te worden gedeclareerd. Bovendien wordt de teller T_TELLER automatisch opgehoogd, zodat we dit niet zelf hoeven te doen. Het PL/SQL-programma met FOR-loop ziet er dus als volgt uit:

```
begin
  for t_teller in 1..10 loop
    insert into hulptabel
      values ( t_teller
             , null
             , null
             );
  end loop;
end;
/
```

Controleer het resultaat. Let er hier weer op, dat de lus wordt afgesloten door het END LOOP keyword. Tussen LOOP en END LOOP mogen weer meerdere statements komen, waarbij gebruik gemaakt mag worden van geneste lussen.

Merk op dat de naam van een tijdelijke variabele begint met t_. Dat is geen verplichting maar bevorderd de leesbaarheid. De tijdelijke variabele van een cursorgestuurde loop krijgt meestal een naam die met r_ begint. Meer hierover in het volgende hoofdstuk.

Voorbeeld:

In het volgende voorbeeld worden 2 loops genest, in de hulptabel worden de waarden van de teller van eerste loop en van de tweede loop gezet.

```
begin
  for t_teller1 in reverse 1..5 loop
    insert into hulptabel
      values ( null
             , 'Naar de binnenste loop met'
             , 't_teller1 is '||t_teller1
             );
    for t_teller2 in 1..5 loop
      insert into hulptabel
        values ( t_teller1
               , t_teller2
               , null
               );
    end loop;
  end loop;
end;
/
```

Merk op dat de tellers weer niet gedeclareerd hoeven te worden.

3.3.4 FOR lussen met cursor

In paragraaf 1.6 zijn we een lus tegengekomen die aan de hand van een bepaalde tabel wordt doorlopen: van alle werknemers uit de tabel P_WERKNEMERS werd het salaris bij elkaar opgeteld. Het PL/SQL-programma zag er als volgt uit:

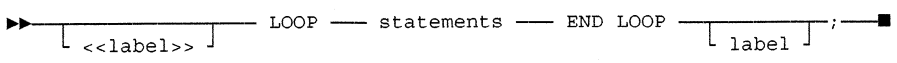
```
declare
  v_totaal number;
  cursor c_sal is
    select sal
    from   p_werknemers;
begin
  v_totaal := 0;
  for r_sal in c_sal loop
    v_totaal := v_totaal + r_sal.sal;
  end loop;
  insert into salaristotaal
    values( 'PL/SQL'
          , v_totaal
          );
end;
/
```

In dit programma wordt een FOR-loop met *cursor* gebruikt. Hiermee kan een herhaling worden gerealiseerd, die voor iedere rij van een bepaalde query wordt doorlopen. De FOR-loop met cursor komt in hoofdstuk 4 uitgebreid aan de orde.

Merk op dat we de naam van een cursor laten beginnen met c_. Ook dit is weer voor de leesbaarheid van de code.

3.3.5 Ongelimiteerde lussen

Een laatste type lus wordt gevormd door de ongelimiteerde (of oneindige) lus (infinite loop). Dit is een herhaling die nooit wordt beëindigd. Een dergelijke lus zou kunnen worden gerealiseerd door een WHILE-loop te maken, waarvan de voorwaarde altijd TRUE blijft. PL/SQL kent echter ook een speciale syntax voor ongelimiteerde lussen:



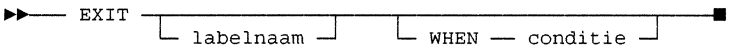
Een ongelimiteerde lus is alleen zinvol, wanneer op één of andere manier de herhaling toch kan worden gestopt. Dit kan worden gerealiseerd met behulp van GOTO, EXIT, CONTINUE of RAISE.

3.3.6 Voortijdig stoppen

Om een iteratie (en in het bijzonder een ongelimiteerde lus) voortijdig te kunnen stoppen, kunnen in PL/SQL vier statements gebruikt worden: GOTO, EXIT, CONTINUE (Oracle11g) en RAISE. De eerste van deze vier, GOTO, zijn we al eerder tegengekomen. Dit statement kan worden gebruikt om naar een willekeurig label binnen het programma te springen. Dit kan ook vanuit een lus worden gedaan. De herhaling wordt dan afgebroken. Overigens kan nooit vanaf een plaats *buiten* een bepaalde lus naar een plaats *binnen* die lus worden gesprongen. Andersom is wel mogelijk.

Zoals eerder opgemerkt, moet het gebruik van het GOTO statement zoveel mogelijk worden beperkt. Omdat in sommige situaties (bijvoorbeeld een ongelimiteerde lus) het afbreken van een herhaling toch noodzakelijk is, bestaat er een variant op het GOTO statement: EXIT. Het EXIT statement mag alleen binnen lussen worden gebruikt, en zorgt ervoor, net als het GOTO statement, dat naar een bepaald label wordt gesprongen. Wanneer geen label wordt opgegeven, wordt naar het eerste statement ná de huidige herhaling gesprongen.

Bovendien mag aan het EXIT statement een voorwaarde worden verbonden:



Op deze manier kunnen we het voorbeeld van de getallen 1 tot en met 10 uitvoeren met behulp van een oneindige lus. Het EXIT statement zorgt ervoor, dat uit de lus wordt gesprongen wanneer het laatste getal (10) is opgeslagen:

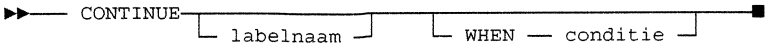
```
declare
  v_teller number := 1;
begin
  loop
    insert into hulptabel
      values ( v_teller
             , null
             , null
              );
    exit when v_teller = 10;
    v_teller := v_teller + 1;
  end loop;
end;
/
```

Controleer het resultaat. In plaats van het EXIT statement met WHEN clausule hadden we ook een IF statement kunnen gebruiken om de waarde van V_TELLER te testen:

```
if v_teller = 10 then
  exit;
end if;
```

In Oracle11g is het CONTINUE statement geïntroduceerd. Dit statement is een aanvulling op het EXIT statement. Het EXIT statement zorgt ervoor dat je uit de huidige LOOP springt. Het CONTINUE statement zorgt er juist voor dat je naar het begin van de huidige LOOP springt. Het EXIT statement blijft wel noodzakelijk om de LOOP te kunnen beëindigen.

Syntax:



Voorbeeld:
We gaan weer de getallen 1 tot en met 10 in de hulptabel zetten. Het CONTINUE statement zorgt ervoor, dat uit de lus wordt gesprongen wanneer het laatste getal (10) is opgeslagen. Vervolgens kun je nog andere statements binnen de loop uitvoeren.

```
declare
  v_teller number := 0;
begin
  loop                                    --zolang de conditie in CONTINUE voldoet komt Oracle hier terug
    v_teller := v_teller + 1;                                    --v_teller wordt met 1 verhoogd
    insert into hulptabel
      values ( v_teller
             , null, null);
    continue when v_teller < 10;
    insert into hulptabel     -- als conditie in CONTINUE niet meer voldoet gaat Oracle hier verder
      values ( null
             , 'het CONTINUE statement is afgelopen'
             , null
              );
    exit when v_teller >= 10;                  -- de EXIT blijft nodig om uit de LOOP te komen
  end loop;
end;
/
```

Controleer het resultaat. In plaats van het CONTINUE statement met WHEN clausule hadden we ook een IF statement kunnen gebruiken om de waarde van V_TELLER te testen:

```
if v_teller < 10 then
  continue;
end if;
```

Merk op, dat bij een gestructureerde manier van programmeren ook in het EXIT statement en het CONTINUE statement niet met labels gewerkt hoeft te worden.

De vierde manier om een lus voortijdig te beëindigen, is het RAISE statement. Dit wordt behandeld in hoofdstuk 6, Foutafhandeling.

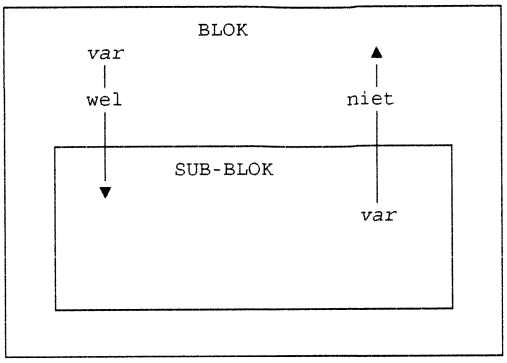
3.4 Geneste blokken

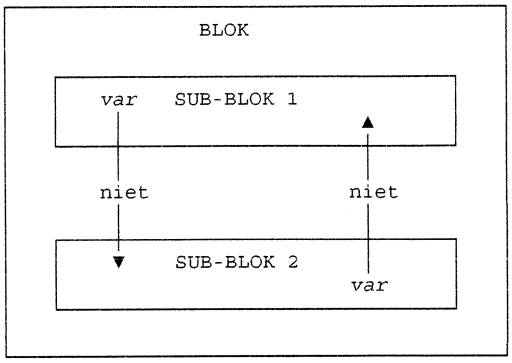
We kunnen PL/SQL-blokken nesten binnen andere PL/SQL-blokken. Men spreekt dan van SUB-BLOCKS. Ze mogen voorkomen in de programmasectie en de foutafhandelingsectie, maar niet in de declaratiesectie.

Wanneer we in zo'n heterogeen PL/SQL programma refereren aan een identifier (variabele, constante, exception, etc.) dan moeten we de reikwijdte ofwel de SCOPING van dat object kennen.

We kennen lokale en globale identifiers. Binnen het huidige blok zijn ze lokaal en naar de sub-blokken toe zijn ze globaal. Een sub-blok kan dus refereren aan een identifier van het omgevende blok, maar omgekeerd kan dat niet !
We kunnen ook te maken hebben met twee sub-blokken op gelijk niveau. Daarin kan ook niet gerefereerd worden aan elkaars identifiers.

Ter illustratie:





Deze restricties blijven gelden wanneer gebruik gemaakt wordt van labels. Labels zijn nuttig om gelijknamige identifiers in hoofdblok en sub-blok van elkaar te kunnen onderscheiden. Dit kan worden bereikt, door het hoofdblok te labelen en dat label als voorvoegsel te gebruiken bij de namen van de identifiers in het sub-blok (gescheiden door een punt).

Voorbeeld:

In dit voorbeeld vullen we de HULPTABEL met variabelen uit een hoofd- en een subblok.

```
declare
  v_tekst   varchar2(30) := 'Dit is het hoofdblok';
  v_teller  number := 0;
begin
  v_teller := v_teller + 1;
  insert into hulptabel
    values ( null
           , 'v_teller='|| v_teller
           , v_tekst );
  declare
    v_tekst varchar2(30) := 'Dit is het subblok';
  begin
    v_teller := v_teller + 1;
    insert into hulptabel
      values ( null
             , 'v_teller='||v_teller
             , v_tekst);
  end;
end;
/
```

In het sub-blok is de variabele V_TELLER globaal en de variabele V_TEKST lokaal. Wanneer we het buitenste blok hadden gelabeld en in het binnenste blok hadden gerefereerd aan LABEL_NAAM.V_TEKST dan kregen we de V_TEKST van het buitenste blok:

```
begin
<<hoofd>>
   declare
     v_tekst   varchar2(30) := 'Dit is het hoofdblok';
     v_teller  number := 0;
   begin
     v_teller := v_teller + 1;
     insert into hulptabel
       values ( null
              , 'v_teller='||v_teller
              , v_tekst
```

```
            );
        <<sub>>
        declare
          v_tekst varchar2(30) := 'Dit is het subblok';
        begin
          v_teller := v_teller + 1;
          insert into hulptabel
            values ( null
                   , 'v_teller='||v_teller
                   , hoofd.v_tekst
                   );
        end;
      end;
  end;
  /
```

3.5 Fouten in de syntax

Bij het maken van de voorbeelden bent u misschien al een aantal foutmeldingen tegen gekomen. In deze paragraaf willen we nog een aantal veel voorkomende fouten laten zien.

In vb3_1.sql staat onderstaand PL/SQL programma dat de getallen 1 tot en met 10 in de HULPTABEL zet. Er zitten echter diverse fouten in de code.

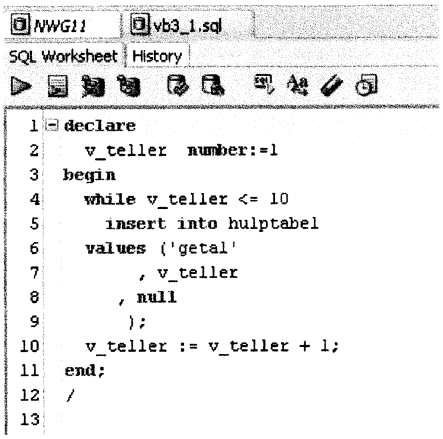
```
declare
  v_teller  number:=1
begin
  while v_teller <= 10
    insert into hulptabel
      values ('getal'
             , v_teller
             , null
              );
      v_teller := v_teller + 1;
end;
/
```

Voer vb3_1.sql uit en bekijk het resultaat.

Onderstaande foutmeldingen worden gegeven:

```
begin
*
ERROR at line 3:
ORA-06550: line 3, column 1:
PLS-00103: Encountered the symbol "BEGIN" 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 "BEGIN" to continue.
ORA-06550: line 5, column 3:
PLS-00103: Encountered the symbol "INSERT" when expecting one of the following:
* & - + / at loop mod remainder rem <an exponent (**)> and or
|| multiset
The symbol "loop" was substituted for "INSERT" to continue.
ORA-06550: line 11, column 5:
PLS-00103: Encountered the symbol ";" when expecting one of the following:
loop
```

Oracle geeft aan dat op drie verschillende regels een fout is gesignaleerd (de foutregels zijn vet afgedrukt). We zien bij iedere fout de volgende tekst staan: `: Encountered the symbol .... when expecting one of the following:..` eventueel gevolgd door een aantal suggesties wat er ingevuld zou kunnen worden om de fout op te lossen. De laatste regel geeft aan wat de meest voor de handliggende oplossing voor de fout is. Bijvoorbeeld : `The symbol ";" was substituted for "BEGIN" to continue.` Dit zijn allemaal syntactische fouten. Om te achterhalen welke code er op welke regel staat kunt u het beste de Line Numbers activeren in de SQL Worksheet.



De eerste fout staat in regel 3:

```
ORA-06550: line 3, column 1:
PLS-00103: Encountered the symbol "BEGIN" 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 "BEGIN" to continue.
```

De fout zit echter niet in regel 3 maar in regel 2, hier mist namelijk een puntkomma aan het einde van de regel. Oracle kan dit pas signaleren op regel 3, omdat in principe de puntkomma ook op de volgende regel zou kunnen staan. Aangezien hier geen puntkomma maar het key-word BEGIN staat treedt de error op. Oracle geeft zelf al aan wat de oplossing van het probleem, is namelijk het toevoegen van een puntkomma, middels de tekst:.

```
The symbol ";" was substituted for "BEGIN" to continue.
```

De tweede fout staat in regel 5:

```
ORA-06550: line 5, column 5:
PLS-00103: Encountered the symbol "INSERT" when expecting one of the following:

   * & - + / at loop mod remainder rem <an exponent (**)> and or
   || multiset
The symbol "loop" was substituted for "INSERT" to continue.
```

Ook in deze regel is niets mis, maar wel in regel 4. Daar mist het keywoord LOOP behorend bij het WHILE statement.

De derde fout staat in regel 11:

```
ORA-06550: line 11, column 4:
PLS-00103: Encountered the symbol ";" when expecting one of the following:

   loop
06550. 00000 -  "line %s, column %s:\n%s"
*Cause:    Usually a PL/SQL compilation error.
*Action:
```

Het PL/SQL programma wordt afgesloten door END;, maar we hebben de WHILE lus nog niet afgesloten met een END LOOP;.
We passen de code in de worksheet aan, zodat de fouten niet meer bestaan. Wanneer we nu het script vb3_1.sql nogmaals uitvoeren zal er nog een foutmelding verschijnen:

```
Error starting at line 1 in command:
declare
  v_teller  number:=1;
begin
  while v_teller <= 10 loop
    insert into hulptabel
      values ('getal'
             , v_teller
             , null
             );
    v_teller := v_teller + 1;
    end loop;
end;
Error report:
ORA-01722: invalid number
ORA-06512: at line 5
01722. 00000 -  "invalid number"
*Cause:
*Action:
```

Deze melding is niet eerder gesignaleerd, omdat dit geen syntactische maar een semantische fout is. Deze fout is pas te constateren als het programma uitgevoerd wordt. Oracle zegt dat er een fout zit in regel 1 en in regel 5. Welke regel is nu fout. Dat is regel 5. Met het INSERT statement proberen we de alfanumerieke waarde 'getal' in de eerste numerieke kolom van de hulptabel te stoppen. Dat gaat niet en dat veroorzaakt dus bovenstaande foutmelding.

4 Benaderen van de database

4.1 Inleiding

PL/SQL biedt ook de mogelijkheid om rijen/waarden uit een tabel te verwerken in de code. Voor het ophalen van waarden uit een tabel hebben we de beschikking over het SELECT INTO statement en over Cursoren. De bij deze mogelijkheden behorende theorie bespreken we in dit hoofdstuk.

In PL/SQL mogen we overigens geen DDL statements gebruiken. DDL staat voor Data Definition Language. Het gaat hier om statements als CREATE, GRANT, AUDIT, RENAME etc.. Toegestane statements m.b.t. SQL zijn:

Raadpleging :	Onderhoud :	Transacties :
SELECT	INSERT	COMMIT
OPEN	UPDATE	ROLLBACK
FETCH	DELETE	SAVEPOINT
CLOSE	MERGE	SET TRANSACTION READ ONLY
		LOCK TABLE

De categorie raadpleging bespreken we dus in dit hoofdstuk. De beide andere categorieën zullen in het volgende hoofdstuk worden besproken.

4.2 Raadpleging door middel van SELECT INTO (impliciete cursor)

Met het SELECT INTO statement kunnen we waarden of een complete rij uit de database ophalen. Het SELECT INTO statement moet echter precies één rij ophalen, anders ontstaat er een foutsituatie. Wanneer het statement meerdere rijen zou ophalen, verschijnt bij het uitvoeren van het PL/SQL-blok de onderstaande foutmelding:

```
Error report:
ORA-01422: exact fetch returns more than requested number of rows
ORA-06512: at line 4
01422. 00000 -  "exact fetch returns more than requested number of rows"
*Cause:    The number specified in exact fetch is less than the rows returned.
*Action:   Rewrite the query or change number of rows requested
```

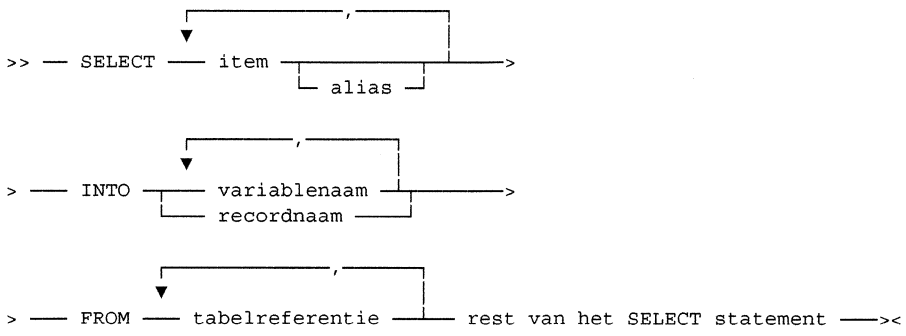
Wanneer het statement geen enkele rijen zou ophalen, verschijnt bij het uitvoeren van het PL/SQL-blok de volgende foutmelding:

```
Error report:
ORA-01403: no data found
ORA-06512: at line 4
01403. 00000 -  "no data found"
*Cause:
*Action:
```

We moeten dus van tevoren weten of en hoeveel rijen er worden opgehaald. Als we dat niet weten kunnen we beter gebruik maken van cursoren, maar hierover meer in de volgende paragraaf.

Het SELECT INTO statement is een uitbreiding op het bekende SELECT statement en wijkt dus enigszins af van de reeds bekende syntax; er is een INTO clausule bijgekomen.

Syntax:



Toelichting:

Bij de INTO-clausule moeten we aangeven waarin de opgehaalde gegevens terecht moeten komen. Dat kan een lijst van één of meerdere variabelen zijn of een zgn. rijvariabele. Deze variabelen moeten allemaal vooraf in de declaratiesectie zijn gedefinieerd.

De rijvariabele is een structuur die alle kolommen van de tabel c.q. view bevat. Een rijvariabele definiëren we door de naam van de tabel of view als gegevenstype op te geven met als achtervoegsel %ROWTYPE.

Wanneer we een lijst van variabelen opnoemen moeten we erop letten dat hun aantal en gegevenstype altijd overeenkomt met de expressies in de SELECT clausule.

In de volgende voorbeelden zal een en ander worden verduidelijkt.

Voorbeeld 1:

Dit voorbeeld plaatst het nummer en de naam van speler 44 in twee variabelen. Let op dat dus de volgorde en de datatypen van de variabelen overeenkomen met de volgorde en de datatypen van de geselecteerde kolommen.

```
declare
  v_spelnr      number(3) ;
  v_naam        varchar2(25) ;
begin
  select spelnr
  ,      naam||' '||voorl
  into   v_spelnr, v_naam
  from   p_spelers
  where  spelnr = 44;
  insert into hulptabel
    values(v_spelnr, v_naam, null);
end;
/
```

Als we regel 7 zouden vervangen door `into v_naam, v_spelnr`, zou Oracle een foutmelding geven; de datatypen van de opgehaalde waarden en de variabelen komen namelijk niet met elkaar overeen.

Voorbeeld 2:

Dit voorbeeld plaatst alle gegevens van speler 44 in een rijvariabele. Let op dat we alle kolommen in de selectie moeten meenemen, omdat deze ook allemaal in de rijvariabele zijn opgenomen. Zijn we echter niet in alle kolommen geïnteresseerd, maar willen we toch de rijvariabele gebruiken, dan biedt voorbeeld 3 een uitkomst.

```
declare
  r_spelers    p_spelers%rowtype;
begin
  select *
  into   r_spelers
  from   p_spelers
  where  spelnr = 44;
  insert into hulptabel
    values(r_spelers.spelnr, r_spelers.naam||' '||r_spelers.voorl, null);
end;
/
```

Voorbeeld 3:

Dit voorbeeld plaatst de drie gegevens van speler 44 uit de selectie (nummer, naam en voorletters) in de overeenkomende attributen uit de rijvariabele. De opbouw is gelijk aan dat van het eerste voorbeeld, maar nu vullen we de attributen uit de rijvariabele R_SPELERS.

```
declare
  r_spelers    p_spelers%rowtype;
begin
  select spelnr
  ,      naam
  ,      voorl
  into   r_spelers.spelnr
  ,      r_spelers.naam
  ,      r_spelers.voorl
  from   p_spelers
  where  spelnr = 44;
  insert into hulptabel
    values(r_spelers.spelnr, r_spelers.naam||' '||r_spelers.voorl, null);
end;
/
```

Het SELECT INTO statement moet dus precies één rij ophalen, anders ontstaat er een foutsituatie. Daarom is het handiger om gebruik te maken van een zogenaamde cursor (expliciete cursor). Hiermee ontstaan geen fouten als er meerdere of geen rijen worden gevonden.

4.3 Raadpleging door middel van een cursor

Een cursor is een kleine work area van enkele kilobytes die de taak heeft om het SQL statement te besturen. Deze leest het statement in en controleert op syntax en bevoegdheden. Daarna wordt rij voor rij afgewerkt. De cursor fungeert als een wijzer die onder andere het huidige rijnummer bijhoudt.

We kennen binnen PL/SQL twee constructies om gegevens uit de database op te halen met behulp van een cursor:
- Raadplegen met behulp van de commando's OPEN, FETCH en CLOSE.
- Raadplegen met behulp van een FOR-loop.

In de volgende subparagrafen worden beide methoden uitgelegd.

4.3.1 OPEN-FETCH-CLOSE

Een cursor die wordt uitgevoerd met de statements OPEN, FETCH en CLOSE zorgt ervoor dat er precies één rij uit de tabel wordt opgehaald. De rij die wordt opgehaald is de eerste rij die we ophalen als we het SELECT statement in SQL*Plus zouden uitvoeren.

De procedure voor het gebruik van deze statements gaat als volgt:

1) Definieer in de declaratiesectie een cursor voor de betreffende query:

```
>> — CURSOR cursornaam — IS — SELECT statement ——><
```

Het SELECT statement mag hier geen INTO regel bevatten.

2) Open in de programmasectie de cursor:

```
>> — OPEN cursornaam ——><
```

Het SELECT statement wordt door Oracle getest op juistheid en er worden voorbereidingen getroffen om de eerste rij te gaan ophalen.

3) Haal de huidige rij op en vul de variabele(n) met de opgehaalde waarde(n). Deze retrieval kan geïtereerd worden, dit zullen we later nog gaan zien (Oracle zet namelijk na een FETCH statement de wijzer bij de volgende rij):

```
>> — FETCH cursornaam — INTO variabele ——><
```

4) Sluit na afloop de cursor:

```
>> — CLOSE cursornaam ——><
```

Voorbeeld:
Dit voorbeeld haalt het totale salaris op van de verkopers en plaatst deze in de tabel SALARISTOTAAL. De constructie met OPEN-FETCH-CLOSE is hier uitstekend toepasbaar, omdat we zeker weten dat de cursor maar één waarde ophaalt.

```
declare
  v_totaal number;
  cursor c_sal is
    select sum(sal)
    from   p_werknemers
    where  functie = 'VERKOPER';
begin
  open c_sal;
  fetch c_sal into v_totaal;
  close c_sal;
  insert into salaristotaal
    values( 'Totaal'
          , v_totaal
          );
end;
/
```

Op het moment dat we gegevens uit de database willen ophalen, dan moeten we altijd gebruik maken van een cursor (ervan uitgaande dat we geen SELECT INTO willen gebruiken, omdat dat fouten kan geven).

Eigenlijk is een cursor niet meer dan een hulpmiddel om een SELECT statement binnen de PL/SQL-code uit te voeren. In de declaratiesectie krijgt een klein geheugengebied in de database (de cursor) een naam (de cursornaam) en wordt er een statement aan gekoppeld. In de body van het programma wordt de cursor pas uitgevoerd; de gegevens worden daar opgehaald.

In het volgende voorbeeld zal duidelijk worden dat we beter geen OPEN, FETCH en CLOSE kunnen gebruiken op het moment dat we met de cursor meerdere rijen willen ophalen.

Voorbeeld:
In dit voorbeeld willen we de salarissen van de personen uit de tabel P_WERKNEMERS optellen en in de hulptabel plaatsen.

```
declare
  v_totaal  number;
  v_salaris number;
  cursor c_sal is
    select sal
    from   p_werknemers;
begin
delete from salaristotaal;          -- eerdere resultaten uit de tabel verwijderen
  v_totaal := 0;
  open c_sal;                       --Select statement wordt gecontroleerd
  fetch c_sal into v_salaris;       --De eerste rij wordt opgehaald en in de variabele v_salaris gezet
  v_totaal := v_totaal + v_salaris;
  insert into salaristotaal
    values( 'PL/SQL'
          , v_totaal
          );
  close c_sal;                      --De cursor wordt gesloten
end;
/
```

Als we nu het salaristotaal opvragen, krijgen we het volgende resultaat

```
select *
from   salaristotaal;

  TAAL                    TOTAAL
  -------------------- ---------
1 PL/SQL                    2400
```

We zien dat deze code niet tot een goed resultaat leidt. De cursor heeft slechts één rij opgehaald; het salaris van de persoon SMITS, dit is de eerste rij uit de tabel. Als een cursor meerdere rijen ophaalt zullen we in de meeste gevallen een FOR-loop gebruiken. In de volgende paragraaf gaan we hier dieper op in.

4.3.2 FOR lussen met cursor

Wanneer we de code uit het vorige voorbeeld ombouwen tot onderstaand voorbeeld, voert het PL/SQL-programma wel het gewenste uit; van alle personen wordt één voor één het salaris opgehaald (regel 6 t/m 8), het salaris wordt bij de overige reeds opgehaalde salarissen opgeteld en het resultaat wordt en in de hulptabel geplaatst.

De FOR-loop stopt pas wanneer de cursor geen rijen meer ophaalt uit de database.

```
 1 declare
 2   v_totaal number;
 3   cursor c_sal is
 4     select sal
 5     from   p_werknemers;
 6 begin
 7   v_totaal := 0;
 8   for r_sal in c_sal loop
 9     v_totaal := v_totaal + r_sal.sal;
10   end loop;
11   insert into salaristotaal
12     values( 'PL/SQL'
13           , v_totaal
14           );
15 end;
16 /
```

Het openen van de cursor (OPEN) gebeurt hier impliciet evenals het ophalen van de rijen (FETCH) en het sluiten van de cursor (CLOSE) als er geen rijen meer kunnen worden opgehaald. Merk op dat volgens de standaarden de indexnaam van de FOR-loop begint met r_ (r is afkomstig van row). Dit in tegenstelling tot de indexnaam die begint met t_ bij 'gewone' FOR-loops (t is afgeleid van temporary).

Dergelijke FOR-lussen worden ook wel cursorgestuurde lussen genoemd. In PL/SQL moeten we voor elke query, waarvan het aantal rijen niet van tevoren bekend is, expliciet een cursor declareren. Aan de hand van die cursor kunnen we in de programmasectie het verloop van de selectie volgen, zoals het volgnummer van de huidige rij.

Merk op dat we niet meer met indexwaarde r_sal naar een opgehaalde waarde kunnen verwijzen als een loop eenmaal is beëindigd. De cursor is op dat moment reeds gesloten en r_sal verwijst juist naar de huidige rij die we binnen de loop hebben opgehaald. Oracle zal een foutmelding geven.

Als we bijvoorbeeld in het INSERT statement uit het voorbeeld r_sal.sal in plaats van v_totaal opnemen, krijgen we bij het uitvoeren van het PL/SQL-blok een melding:

```
PLS-00201: identifier 'R_SAL.SAL' must be declared
```

Syntax:

De vereenvoudigde syntax van de declaratie van de cursor luidt als volgt (deze is gelijk aan de syntax in de vorige paragraaf):

```
>> ── CURSOR cursornaam ── IS SELECT statement ──><
```

De FOR-loop met cursor doorloopt voor elke rij die door de query wordt opgehaald een zelfde reeks statements. De syntax van de FOR-loop luidt als volgt:

```
>> ── FOR ─ rij_variabele ─ IN ─ cursornaam ── LOOP ─ statements ─ END LOOP ──><
```

Termen:

rij_variabele	Een variabele waarin de afzonderlijke kolommen qua naam en gegevenstype vervat zijn. De variabele kan uit meerdere velden bestaan en hoeft niet gedeclareerd te worden. Ook wel indexwaarde genoemd.
cursornaam	De expliciete cursor die gedefinieerd is in de declaratie sectie.

Voorbeeld:

Het volgende voorbeeld plaatst de naam, het salaris en de functie van alle werknemers in de hulptabel.

```
declare
  cursor c_werknemers is
    select naam
    ,      sal
    ,      functie
    from   p_werknemers;
begin
  for r_werknemers in c_werknemers loop
    insert into hulptabel
      values ( r_werknemers.sal
             , r_werknemers.naam
             , r_werknemers.functie
             );
  end loop;
end;
/
```

Met het gebruik van cursoren maken we ook vaak gebruik van parameters. In de volgende paragraaf wordt uitgelegd wanneer en hoe we parameters kunnen toepassen.

4.4 Parameters

In SQL kunnen we heel gemakkelijk een voorwaarde aan een SELECT statement meegeven. Bijvoorbeeld:

```
select   teamnr
from     p_wedstrijden
where    gewonnen > 2                          -- een voorwaarde in de WHERE clausule
group by teamnr
having   count(spelnr) > 3;                    -- een voorwaarde in de HAVING clausule
```

In PL/SQL is het geen enkel probleem om statements met dergelijke voorwaarden in cursoren op te nemen. Het kan echter zijn dat een bepaalde voorwaarde nog niet bekend is op het moment dat een programma wordt geschreven, omdat bij het uitvoeren van de code die waarde door de gebruiker wordt ingevoerd, of dat die waarde pas binnen het PL/SQL-blok wordt bepaald.

De waarde binnen de WHERE- of HAVING clausule kan dan worden vervangen door een variabele. Op het moment dat zo'n waarde variabel wordt, kunnen we gebruik maken van een parameter; dit is een variabele die alleen binnen de cursor bekend is.

Hieronder hebben we het voorbeeld uit de vorige paragraaf aangepast naar een programma waarbij de door de gebruiker opgegeven werknemersfunctie via een parameter aan de cursor wordt doorgegeven:

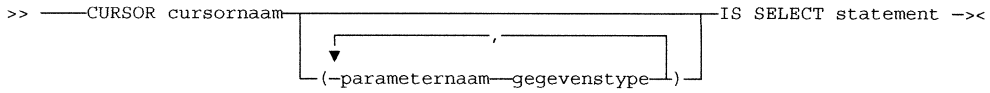
```
declare
  cursor c_werknemers(b_functie varchar2) is     --Het lokaal definiëren van de parameter
    select naam
    ,      sal
    ,      functie
    from   p_werknemers
    where  upper(functie) = upper(b_functie);    --Het lokaal toepassen van de parameter
```

```
begin
  for r_werknemers in c_werknemers('klerk') loop   -- Het toekennen van een waarde 'klerk'
    insert into hulptabel                                -- aan de parameter
      values ( r_werknemers.sal
             , r_werknemers.naam
             , r_werknemers.functie
             );
  end loop;
end;
/
```

De functie klerk wordt hier nu in de FOR-loop aan de cursor doorgegeven, middels `c_werknemers('klerk')`. Binnen de cursor neemt de parameter B_FUNCTIE deze waarde over. De parameter wordt op zijn beurt weer in de WHERE-clausule van het statement gebruikt.

We kunnen bij het declareren van een cursor altijd één of meerdere parameters opgeven. We breiden daarvoor nu de syntax van het declareren van de cursor uit:

Syntax:



Het parameterveld bestaat uit de naam plus het gegevenstype. Mogelijke gegevenstypes zijn VARCHAR2, NUMBER, DATE, waarbij we, in tegenstelling tot kolomdatatypen, géén lengte of precisie mogen meegeven. De cursorparameters gelden exclusief voor de betreffende cursor en zijn daarbuiten niet te gebruiken.

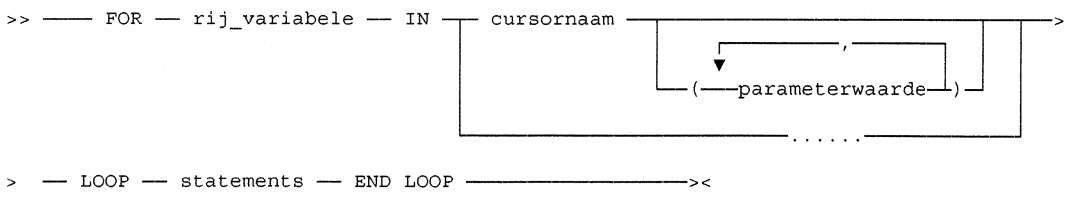
Om de code weer te standaardiseren, laten we de naam van de parameters altijd beginnen met b_. We zouden verwacht dat het voorvoegsel p_ zou zijn, maar dit is reeds gereserveerd voor parameters die binnen Procedures en Functies worden gebruikt (hiervoor verwijzen we naar onze cursus: PL/SQL, de Procedurele Extensie).

Zoals we in het begin van deze paragraaf hebben aangegeven, declareren we parameters binnen een cursor, om in het SELECT statement te kunnen verwijzen naar een variabele. Gewoonlijk zal dat dus gebeuren in de WHERE of HAVING clausule teneinde een nauwere selectie mogelijk te maken. Een gedeclareerde cursorparameter moet daadwerkelijk gebruikt worden in het SELECT statement, anders heeft deze geen zin.

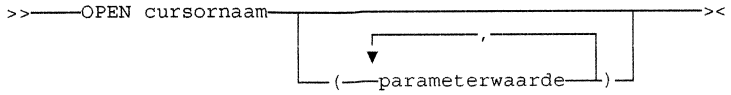
De syntax voor het doorgeven van een waarde aan de parameter bij de aanroep van de cursor luidt als volgt:

Syntax:

In de FOR-loop (de volledige syntax staat in de appendix):



Binnen het OPEN statement van de constructie OPEN-FETCH-CLOSE:



Voor de parameterwaarde vult u een constante, een programmavariabele of een expressie in. U moet net zoveel parameters meegeven als er in het DECLARE CURSOR statement voorkomen. Houd ook dezelfde volgorde aan, omdat de parameters positioneel zijn, tenzij u expliciet de naam opgeeft:

```
open cursornaam(parameternaam => parameterwaarde, ...);

for rijnaam in cursornaam(parameternaam => parameterwaarde, ...) loop;
```

Het gebruik van parameters maakt het programma vaak overzichtelijk, als de programma's groter worden. In het vorige voorbeeld hebben we een variabele gedeclareerd en die door gegeven aan de cursor. Met een parameter gebeurt iets dergelijks, alleen het definiëren van de parameter gebeurt binnen de cursor en het toekennen van een waarde gebeurt pas als de cursor wordt geopend.

We zullen in korte stappen gaan kijken wat er op de achtergrond gebeurt, wanneer we een cursor met parameter aanroepen. Om het geheel overzichtelijk te houden hebben we de onbelangrijke code weggelaten en met stippellijnen opgevuld.

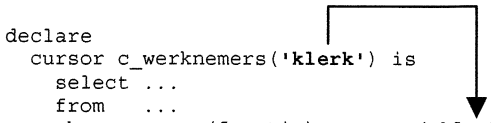
Uitgangssituatie: De cursor is gedeclareerd en in de body wordt deze aangeroepen. De waarde 'klerk' wordt aan de cursor meegegeven.

```
declare
  cursor c_werknemers(b_functie varchar2) is
    select ...
    from   ...
    where  upper(functie) = upper(b_functie);
begin
  for r_werknemers in c_werknemers('klerk') loop
    ...
  end loop;
end;
/
```

Stap 1: De cursor wordt geopend en de parameter binnen de cursor neemt de waarde 'klerk' over die bij de aanroep van de cursor wordt meegegeven.

```
declare
  cursor c_werknemers('klerk') is
    select ...
    from   ...
    where  upper(functie) = upper(b_functie);
begin
  for r_werknemers in c_werknemers('klerk') loop
    ...
  end loop;
end;
/
```

Stap 2: De parameter wordt in de WHERE-clausule gebruikt en ook deze neemt dus de waarde 'klerk' over.



```
declare
  cursor c_werknemers('klerk') is
    select ...
    from   ...
    where  upper(functie) = upper('klerk');
begin
  for r_werknemers in c_werknemers('klerk') loop
    ...
  end loop;
end;
/
```

Na stap 2 worden pas de rijen uit de tabel opgehaald die voldoen aan de voorwaarde uit de WHERE clausule. We zien nu dat er dus in principe maar twee stappen nodig zijn om de waarde 'klerk' naar de WHERE-clausule van het SELECT statement te brengen.
Probeer bij het doorgeven van waarden aan de cursor zoveel mogelijk gebruik te maken van parameters. We zien in het voorbeeld hierboven dat daardoor de code korter en overzichtelijker wordt.

Een voordeel van een parameter is, dat we dezelfde cursor met een andere voorwaarde opnieuw kunnen uitvoeren. Wanneer we bijvoorbeeld na het ophalen van de gegevens van de klerken ook de gegevens van de verkopers willen verwerken, kunnen we de code simpel uitbreiden:

```
declare
  cursor c_werknemers(b_functie varchar2) is
    select ...
    where  upper(functie) = upper(b_functie);
begin
  for r_werknemers in c_werknemers('klerk') loop
    ...
  end loop;
  for r_werknemers in c_werknemers('verkoper') loop     --dezelfde code voor de verkopers
    ...
  end loop;
end;
/
```

Na de eerste loop wordt de cursor impliciet gesloten. De genoemde stappen van de vorige pagina worden bij het uitvoeren van de tweede lus opnieuw doorlopen.
Als we overigens de functie variabel willen maken, kunnen we het ACCEPT-commando toevoegen en de aanroep van c_werknemers wijzigen. Zo kunnen we de werknemers ophalen van een door de gebruiker opgegeven functie.

```
accept a_functie char prompt 'Geef een functie: '

declare
  cursor c_werknemers(b_functie varchar2) is
    select naam
    ,      sal
    ,      functie
    from   p_werknemers
    where  upper(functie)=upper(b_functie);
```

```
begin
  for r_werknemers in c_werknemers('&a_functie') loop
    insert into hulptabel
      values ( r_werknemers.sal
             , r_werknemers.naam
             , r_werknemers.functie
             );
  end loop;
end;
/
```

De waarde uit het ACCEPT commando wordt doorgegeven aan de parameter binnen de cursor via de aanroep van de cursor, via `c_werknemers('&a_functie')`.
In de voorbeelden voeren we de cursor uit met een FOR-loop, omdat er meerdere rijen worden opgehaald. Het gebruik van parameters bij de constructie OPEN-FETCH-CLOSE werkt op een zelfde manier. We zullen hier in de volgende paragraaf voorbeelden van zien.

N.B.: Wanneer u refereert aan de parameter dient u de cursornaam als voorvoegsel te gebruiken indien de parameter dezelfde naam heeft als een kolom uit de tabel(len). Bijvoorbeeld:

```
cursor c_werknemers(functie varchar2) is
    select sal
    from   p_werknemers
    where  upper(functie)=upper(c_werknemers.functie);
```

Wanneer we ons houden aan de standaard naamgeving (parameters binnen cursoren beginnen dan met b_) hoeven we dus nooit gebruik te maken van dit voorvoegsel (of de kolom moet beginnen met b_, maar die kans is zeer klein).

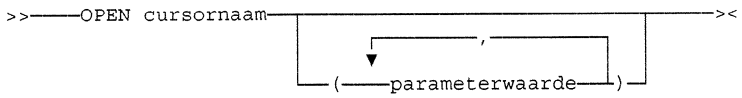
4.5 Controles

We kunnen cursoren ook gebruiken om controles uit te voeren; om te kijken of een bepaalde waarde wel of niet in een bepaalde tabel voorkomt. Hiervoor gebruiken we de statements OPEN, FETCH en CLOSE.

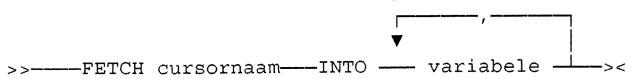
Het gebruik van een cursorgestuurde FOR-loop, zoals we zojuist hebben gezien, heeft in dit geval weinig effect. De loop wordt immers direct weer afgesloten wanneer de cursor geen gegevens ophaalt (dus als de te controleren waarde niet in de tabel voorkomt). We hebben dan geen mogelijkheid om het programma op dat moment de gewenste statements en/of commando's uit te laten voeren.

Syntax:

De volledige syntaxen luiden:



Voor de parameterwaarde vullen we een constante, een programmavariabele of een expressie in. We moeten net zoveel parameters meegeven als in het DECLARE CURSOR statement. Houd ook dezelfde volgorde aan, omdat de parameters positioneel zijn.



De variabele moet gedeclareerd zijn. Voor elke kolom of expressie die door de query-cursor wordt geselecteerd moet er een overeenkomstige variabele zijn aangewezen. De gegevenstypen moeten overeenstemmen of converteerbaar zijn. Wanneer de cursor meerdere waarden ophaalt, kunnen we in plaats van meerdere variabelen ook een rij_variabele gebruiken (zie paragraaf 2.2.5) .

```
>>———CLOSE———cursor_naam———><
```

Bij de statements FETCH en CLOSE mogen geen parameters worden meegegeven.

In het volgende voorbeeld wordt gecontroleerd of een opgegeven cursusnaam voorkomt in de tabel P_CURSUSSEN.

Voorbeeld:

```
accept a_naam char prompt 'De te controleren cursusnaam: '
declare
  cursor c_controle(b_naam varchar2) is
    select naam
    from   p_cursussen
    where  upper(naam)=upper(b_naam);
  v_cursus p_cursussen.naam%type;
  v_naam   p_cursussen.naam%type;
begin
  v_cursus := '&a_naam';
  open  c_controle(v_cursus);
  fetch c_controle into v_naam;
  if v_naam is null then
    insert into hulptabel
      values ( null
             , v_naam
             , v_cursus||' is geen cursus');
  else
    insert into hulptabel
      values (null
             , v_naam
             , v_cursus||' is een cursus');
  end if;
  close c_controle;
end;
/
```

Als de variabele V_NAAM echter al een waarde heeft bij het uitvoeren (initialisatiewaarde), gaat deze constructie fout.

We passen daarom het PL/SQL-blok zo aan, dat de variabele V_NAAM de waarde 'Leeg' krijgt toegekend:

```
accept a_naam char prompt 'De te controleren cursusnaam: '
declare
  cursor c_controle(b_naam varchar2) is
    select naam
    from   p_cursussen
    where  upper(naam)=upper(b_naam);
  v_cursus p_cursussen.naam%type;
  v_naam   p_cursussen.naam%type := 'Leeg';   --v_ophaal_naam krijgt een initialisatiewaarde
begin
  v_cursus := '&a_naam';
  open  c_controle(v_cursus);
  fetch c_controle into v_naam;
```

```
if v_naam is null then
   insert into hulptabel
     values ( null, v_naam
            , v_cursus||' is geen cursus'
            );
  else
    insert into hulptabel
      values ( null, v_naam
             , v_cursus||' is een cursus'
             );
  end if;
  close c_controle;
end;
/
```

Als we het programma uitvoeren met een niet-bestaande cursus (bv 'Frans') gaat het fout. Frans is namelijk geen cursus die voorkomt in de tabel P_CURSUSSEN, maar omdat variabele V_NAAM niet NULL is gaat het in de IF-conditie fout; niet de code na de IF, maar de code direct na de ELSE wordt uitgevoerd. Om deze situaties te vermijden zijn er een aantal cursorattributen waarmee we het ophalen van de rijen kunnen volgen.

4.5.1 Cursorattributen

Met cursorattributen kunnen we volgen wat de cursor heeft uitgevoerd. Er bestaan een viertal cursorattributen. Drie van de vier zijn Booleaans (geven waarde TRUE of FALSE).

`cursor_naam%FOUND`	TRUE wanneer de huidige FETCH een rij heeft opgehaald.
`cursor_naam%NOTFOUND`	TRUE wanneer de huidige FETCH geen rij heeft opgehaald.
`cursor_naam%ROWCOUNT`	Het aantal opgehaalde rijen tot nu toe.
`cursor_naam%ISOPEN`	TRUE wanneer de cursor nog niet gesloten is.

De eerste twee cursorattributen kunnen we toepassen bij de controle van een waarde. Omdat het voorgaande voorbeeld niet het gewenste resultaat gaf, gaan we in de code gebruik maken van het cursorattribuut cursornaam%NOTFOUND.

```
accept a_naam char prompt 'De te controleren cursusnaam: '
declare
  cursor c_controle(b_naam varchar2) is
    select naam
    from   p_cursussen
    where  upper(naam)=upper(b_naam);
  v_cursus p_cursussen.naam%type;
  v_naam   p_cursussen.naam%type := 'Leeg';
begin
  v_cursus := '&a_naam';
  open  c_controle(v_cursus);
  fetch c_controle into v_naam;
  if c_controle%notfound then                    --We gebruiken nu een cursorattribuut
    insert into hulptabel
      values ( null
             , null
             , v_cursus||' is geen cursus'
             );
  else
    insert into hulptabel
      values ( null
             , null
             , v_cursus||' is een cursus'
             );
  end if;
  close c_controle;
end;
/
```

Ondanks het feit dat variabele V_NAAM de waarde 'Leeg' heeft, wordt nu wel het gewenste resultaat gegeven. Het toepassen van cursorattributen is de gebruikelijke manier om controles uit te voeren.

De volgende regels gelden voor cursorattributen:

- de cursorattributen %NOTFOUND en %FOUND kunnen niet gebruikt worden binnen een cursorgestuurde FOR-loop, aangezien ze dan geen waarde toegekend krijgen
- cursorattribuut cursornaam%ISOPEN wordt gebruikt om te bepalen of een cursor open of gesloten is; de waarde van dit cursorattribuut kan dus op elke plaats in de code opgevraagd worden als gebruik wordt gemaakt van OPEN - FETCH – CLOSE; bij een cursorgestuurde FOR-loop kan cursornaam%ISOPEN gebruikt worden buiten de FOR-loop
- het cursorattribuut %ROWCOUNT kan zowel binnen een FOR-loop als ook bij een OPEN – FETCH – CLOSE constructie gebruikt worden;
- als de cursor eenmaal gesloten is hebben de cursorattributen %NOTFOUND, %FOUND en %ROWCOUNT geen waarde meer.

Onderstaande tabel laat zien wat op de verschillende momenten (OPEN - FETCH - CLOSE) de waarden zijn van de cursorattributen.

	voor OPEN	na OPEN	na goede FETCH	na foute FETCH	na CLOSE
%FOUND	*	null	true	false	*
%NOTFOUND	*	null	false	true	*
%ROWCOUNT	*	0	1 of n	0 of n	*
%ISOPEN	false	true	true	true	false

* Oracle zal de volgende foutmelding geven:

```
ORA-01001: invalid cursor
```

n is het aantal opgehaalde rijen.

Wanneer we een SELECT statement in SQL Developer uitvoeren geeft Oracle een melding op het moment dat er geen rijen worden opgehaald uit de geraadpleegde tabel:

```
0 rows selected.
```
met Run Script (F5)

of

All Rows Fetched met Run Statement (F9)

We kunnen dus de uitkomst TRUE van het cursorattribuut cursornaam%NOTFOUND (of FALSE van het cursorattribuut cursornaam%FOUND) vergelijken met deze melding als het SELECT statement uit de cursor geen waarden (meer) ophaalt.

Verder is het belangrijk dat we inzien dat het bij een controle niet van belang is wat er nu voor waarde wordt opgehaald door het SELECT statement. We hebben in het vorige voorbeeld dan ook variabele V_NAAM uit het INSERT statement gehaald, omdat het er helemaal niet toe doet wat na het FETCH statement de waarde van V_NAAM is. Wanneer we de code zouden vervangen door de volgende code, blijft het resultaat hetzelfde.

```
declare
  cursor c_controle(b_naam varchar2) is
    select 1                                          --we halen nu waarde 1 op
    from   p_cursussen
    where  upper(naam) = upper(b_naam);
  v_cursus varchar2(20);
  v_controle number;                                  --het datatype passen we aan
begin
  v_cursus := 'PL/SQL';
  open  c_controle(v_cursus);
  fetch c_controle into v_controle;
  if c_controle%notfound then
    insert into hulptabel
      values ( null
             , null
             , v_cursus||' is geen cursus'
             );
  else
    insert into hulptabel
      values ( null
             , null
             , v_cursus||' is een cursus'
             );
  end if;
  close c_controle;
end;
/
```

In plaats van de kolom NAAM, wordt nu de waarde 1 opgehaald voor de eerste de beste rij die voldoet aan de voorwaarde uit het SELECT statement. De waarde 1 wordt in variabele V_CONTROLE bewaard, maar heeft verder geen functie. Het maakt dus helemaal niet uit welke waarde we ophalen. In veel programma's wordt bij een controle de waarde 1, null of de waarde 'x' opgehaald.

4.5.2 Controles met een lus

We kunnen voor een controle ook een lus bouwen met een OPEN, FETCH en CLOSE. Hierbij gebruiken we ook weer een cursorattribuut, namelijk cursornaam%FOUND.

In het volgende PL/SQL-blok halen we de rijen op van werknemers die binnen een bepaalde functie meer verdienen (inclusief toeslag) dan een bepaald bedrag. De functie en het bedrag geven we door middel van parameters door. Binnen de loop moeten we de rijen ophalen, met een FETCH, totdat er geen rijen meer gevonden worden.

Voorbeeld:

```
declare
  v_persnr   number(4);
  v_functie  varchar2(10);
  v_salaris  number(5);
  cursor c_verkoop ( b_basis number
                   , b_beroep varchar2) is
    select persnr
    ,      functie
    ,      sal + nvl(toeslag,0)
    from   p_werknemers
    where  sal + nvl(toeslag,0) > c_verkoop.b_basis
    and    functie = c_verkoop.b_beroep;
begin
  open c_verkoop(2800,'VERKOPER');
  fetch c_verkoop
    into v_persnr,    v_functie,    v_salaris;
```

```
  while c_verkoop%found loop          --Zolang de cursor rijen ophaalt moet de loop doorlopen worden
    insert into hulptabel
      values ( v_persnr
             , v_salaris
             , v_functie);
    fetch c_verkoop                   --De volgende rij ophalen
      into v_persnr
      ,    v_functie
      ,    v_salaris;
    end loop;
  close c_verkoop;
end;
/
```

Een FOR-loop doet echter hetzelfde en is eenvoudiger en overzichtelijker. Een FOR-loop haalt ook alle rijen op tot er geen rij meer wordt gevonden.

```
declare
  cursor c_verkoop ( b_basis number
                   , b_beroep varchar2) is
    select persnr
    ,      functie
    ,      sal + nvl(toeslag,0) salaris
    from   p_werknemers
    where  sal + nvl(toeslag,0) > b_basis
    and    functie = b_beroep;
begin
  for r_verkoop in c_verkoop(2800,'VERKOPER') loop
    insert into hulptabel
      values ( r_verkoop.persnr
             , r_verkoop.salaris
             , r_verkoop.functie);
  end loop;
end;
/
```

De code `sal + nvl(toeslag,0)` moet hier overigens een alias krijgen, omdat een rij-variabele niet naar een expressie kan verwijzen (`r_verkoop.(sal + nvl(toeslag,0))` zal een foutmelding geven).

Met een cursorgestuurde lus (FOR ... LOOP ... END LOOP) hoeven we dus niet zelf de cursor te openen, steeds een nieuwe FETCH uit te voeren en met een cursorattribuut te controleren of er wel een rij is opgehaald.

Het gebruik van cursoren verdient de voorkeur boven SELECT INTO. Ten eerste omdat met een cursor één of meerdere rijen opgehaald kunnen worden. Ten tweede omdat met de cursorattributen foutsituaties beter af te vangen zijn. Foutsituaties in het geval van SELECT INTO zijn alleen af te vangen met behulp van Predefined (Internal) Exceptions (meer hierover in hoofdstuk 6). En deze kunnen geen onderscheid maken tussen meerdere SELECT INTO statements, zodat je niet weet welke er fout is gegaan.

4.6 Zelf gedefinieerde records

We hebben tot op heden kennis gemaakt met een aantal manieren om gegevens uit een tabel te halen:

- we kunnen de gegevens in losse variabelen plaatsen:

```
v_naam varchar2(20);
...
fetch c_cursor into v_naam;
```

- we kunnen losse variabelen baseren op tabelkolommen middels %TYPE:

```
v_naam werknemers.naam%type;
...
fetch c_cursor into v_naam;
```

- we kunnen een %ROWTYPE variabele aanmaken gebaseerd op de tabel:

```
r_kantoren p_kantoren%rowtype;
...
fetch c_cursor into r_kantoren;
```

of op de cursor:

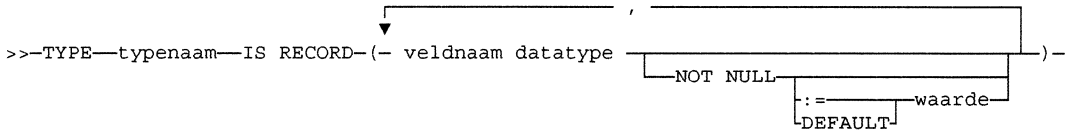
```
cursor c_kan is select kantnr, naam from p_kantoren
r_kan c_kan%rowtype;
...
fetch c_kan into r_kan;
```

Het is echter ook mogelijk in PL/SQL een variabele aan te maken die gebaseerd is op een eigen gedefinieerd recordtype. Door het aanmaken van een recordtype is het mogelijk dat de programmeur zelf de structuur bepaalt. Het aantal velden, de naam en het datatypen kunnen dan zelf worden gekozen.
Na het definiëren van een recordTYPE kan deze structuur gebruikt worden als datatype voor variabelen.

Voorbeeld:
In het onderstaande voorbeeld maken we gebruik van een zelf gedefinieerd record voor het opvangen van de waarden uit de cursor. Hierbij worden de waarden van salaris, functie en naam van de medewerker gebruikt.

```
declare
   type voorbeeld_record is record
        ( functie varchar2(30)
        , naam varchar2(20)
        , salaris number(7,2));
   r_voorbeeld voorbeeld_record;
   cursor c_werkn (b_naam varchar2) is
      select functie, naam, sal
      from p_werknemers
      where upper(naam)=upper(b_naam);
   v_naam p_werknemers.naam%type;
begin
   v_naam:= '&a_naam';
   open c_werkn (v_naam);
   fetch c_werkn into r_voorbeeld;
      if c_werkn%notfound then
         insert into hulptabel values( null
                                     , v_naam||' is geen werknemer'
                                     , null);
      else
         insert into hulptabel values( r_voorbeeld.salaris
                                     , 'Werknemer: '||r_voorbeeld.naam
                                     , null);
      end if;
   close c_werkn;
end;
/
```

Syntax:



Binnen een record kunnen verschillende datatypen worden gebruikt. Het gebruik van records als datatype, zowel zelf gedefinieerde records alsook records van het type %ROWTYPE, zijn toegestaan. We kunnen dus een record binnen een record gebruiken.

Er kunnen bij de datatypen NOT NULL constraints worden opgegeven en een DEFAULT waarde, in plaats van een waarde mag ook een expressie opgegeven worden die een waarde oplevert van het juiste datatype.

Voorbeeld:

```
declare
   type voorbeeld_record is record
      ( functie varchar2(30) not null
      , aantal number(5)
      , gemiddeld number (7,2));
   type voorbeeld_type is record
      ( rec voorbeeld_record
      , wortel number(5,2) := sqrt(69));
```

5 Transacties

5.1 Inleiding

In het vorige hoofdstuk hebben we alleen gegevens uit de database gelezen. Gegevens in de database kunnen echter ook gewijzigd worden met behulp van PL/SQL programma's. Hierbij komen begrippen als transacties en locks naar voren. Deze zullen in dit hoofdstuk uitgelegd worden.

5.2 Raadpleging ten behoeve van wijzigingen

Er is nog een andere bijzonderheid die zich bij het declareren van cursoren kan voordoen. Dat is wanneer er een speciale query wordt gebruikt, namelijk de SELECT ... FOR UPDATE query.

Voorbeeld:

```
declare
  cursor c_voorbeeld is
    select *
    from   p_werknemers
    where  kantnr = 10
    for update;
```

Als de FOR UPDATE clausule wordt gebruikt, moet deze altijd als de laatste worden genoemd. Het maakt deel uit van de standaard syntax van het SELECT statement. Met de FOR UPDATE clausule geven we aan dat we de rijen willen selecteren met het doel ze later te wijzigen (of te verwijderen). Oracle voert de select uit en plaatst een lock op de rijen die door het SELECT statement geselecteerd worden. Hierdoor kunnen andere gebruikers deze rijen alleen nog raadplegen, maar niet wijzigen.

Indien uit meerdere tabellen gelijktijdig geselecteerd wordt, en er slechts één tabel gewijzigd moet worden, kan dit met het optionele OF tabel.kolomnaam worden gespecificeerd. Het statement wordt dan als volgt:

```
cursor c_werknemers is
  select w.sal, w.functie, k.naam
  from   p_werknemers w, p_kantoren k
  where  w.kantnr = k.kantnr
  and    w.functie = 'MANAGER'
  for update of w.sal;
```

Omdat we later in het PL/SQL-blok alleen de salarissen van de managers willen aanpassen, geven we dit op door de clausule FOR UPDATE OF w.sal. Nu worden alleen de geselecteerde rijen in de tabel waarin de kolom w.sal voorkomt gelockt. Geselecteerde rijen in de tabel P_WERKNEMERS worden zodoende wel gelockt en geselecteerde rijen in de tabel P_KANTOREN niet.

Een bijkomend voordeel is dat Oracle 'onthoudt' welk record er precies met de laatste FETCH is opgehaald (met behulp van de recordnummer ofwel rowid), zodat later in het programma een update uitgevoerd kan worden met de clausule WHERE CURRENT OF cursor_naam. Deze clausule zorgt ervoor dat alleen de rij die met de laatste FETCH is opgehaald, wordt gewijzigd.

Locking wordt doorgaans automatisch geregeld door het RDBMS, maar er zijn gevallen waarbij we zelf in willen grijpen op de standaard verwerking van Oracle.

Bijvoorbeeld als we tabellen willen wijzigen die samenhangen met andere tabellen. Als eerste stap wordt een rij gefetched waarna er meerdere tabellen moeten worden geraadpleegd en eventueel gewijzigd. Gedurende deze verwerking mag het mutatierecord niet gewijzigd of verwijderd worden. Pas als laatste s tap van het programma wordt het mutatierecord aangemerkt als zijnde verwerkt en kan de update plaatsvinden. Zo wordt gegarandeerd dat de gegevens in de database consistent blijven.

De WHERE CURRENT OF clausule wordt op de volgende wijze toegevoegd aan de UPDATE en DELETE statements:

```
update .... where current of cursor_naam ;
```

en

```
delete .... where current of cursor_naam ;
```

Deze WHERE clausule refereert dan aan de huidige rij welke door de FETCH is opgehaald.

Merk op dat een UPDATE dan wel DELETE met de WHERE CURRENT OF optioneel is. Het is uiteraard niet verplicht om de geselecteerde rijen te wijzigen.

De volgende regels gelden voor de FOR UPDATE clausule:

- FOR UPDATE **moet** gebruikt worden als er in de code later WHERE CURRENT OF cursor_naam gebruikt wordt.
- FOR UPDATE OF kolom **mag** gebruikt worden als er in het SELECT statement één tabel benaderd wordt; het is dan echter documentatief.
- FOR UPDATE OF kolom(men) **moet** gebruikt worden als er in het SELECT statement meerdere tabellen benaderd worden om aan te geven uit welke tabel rijen gelocked moeten worden.
- In FOR UPDATE OF kolom(men) **moeten** de kolommen uit dezelfde tabel komen; syntactisch mogen de kolommen wel uit meerdere tabellen komen, maar Oracle voert eventuele wijzigingen op de kolommen niet uit; Oracle kan in dit geval namelijk maar uit één tabel rijen locken om te wijzigen.

Deze cursor is goed:

```
cursor c_werknemers is
  select w.sal, w.functie, k.naam
  from   p_werknemers w, p_kantoren k
  where  w.kantnr = k.kantnr
  and    w.functie = 'MANAGER'
  for update of w.sal, w.functie;
```

Maar deze cursor is fout :

```
cursor c_werknemers is
  select w.sal, w.functie, k.naam
  from   p_werknemers w, p_kantoren k
  where  w.kantnr = k.kantnr
  and    w.functie = 'MANAGER'
  for update of w.sal, k.naam;
```

In het volgende voorbeeld worden diensten van STAF gewijzigd. De dagdienst wordt avonddienst, avonddienst wordt nachtdienst en de nachtdienst wordt dagdienst.

Voorbeeld: .

```
declare
  cursor c_dienst is
    select *
    from   p_stafleden
    order by dienst
    for update of dienst;
begin
  for r_dienst in c_dienst loop
    r_dienst.dienst := upper(r_dienst.dienst);
    if r_dienst.dienst='D' then
      update p_stafleden
      set    dienst='A'
      where current of c_dienst;
    elsif r_dienst.dienst='A' then
      update p_stafleden
      set    dienst ='N'
      where current of c_dienst;
    elsif r_dienst.dienst='N' then
      update p_stafleden
      set    dienst ='D'
      where current of c_dienst;
    end if;
  end loop;
end;
/
```

Merk op dat de statements OPEN, FETCH en CLOSE ontbreken, omdat deze acties vanzelf plaatsvinden in de FOR cursor lus. Merk tevens op dat in de UPDATE statements gerefereerd wordt aan de huidige rij van de cursor, er hoeft geen conditie in de vorm van PERSNR = R_DIENST.PERSNR gebruikt te worden.

5.3 Transacties

Alle mutaties op de gegevens in de database zijn pas definitief na het COMMIT statement. Zolang die commit nog niet is gegeven zijn de wijzigingen alleen zichtbaar binnen de huidige sessie van de gebruiker. De andere gebruikers krijgen de oude gegevens te zien, alsof er nog niets is gebeurd. We kunnen de wijzigingen ook definitief ongedaan maken met een ROLLBACK.

Een transactie is een verzameling DML statements, tussen twee COMMITS of ROLLBACKS. Dus de statements waarvan de wijzigingen tegelijkertijd worden weggeschreven of teruggedraaid. Een DDL statement (zoals CREATE TABLE en DROP TABLE) zorgt altijd voor een impliciete COMMIT.

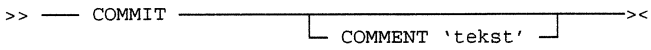
Daarnaast kunnen we ook wijzigingen tot een bepaald punt terugdraaien, door middel van savepoints.

```
>> — SAVEPOINT —— savepointnaam ——><
```

Savepoints zijn markeringen binnen transacties en worden gebruikt om grote transacties onder te verdelen in kleinere stukken. Dit stelt de programmeur in staat om een transactie-rollback in partijen af te handelen. Voor de naam gelden dezelfde afspraken als bij andere databaseobjecten, zoals tabellen.

Binnen de transactie moeten de savepoints een unieke naam hebben. Mochten we een savepoint definiëren dat al eerder bestond, dan geldt de laatste. Het aantal savepoints binnen een transactie is maximaal vijf, tenzij de Database Administrator voor een andere limiet heeft gekozen. Het gebruik van savepoints illustreren we straks met een voorbeeld.

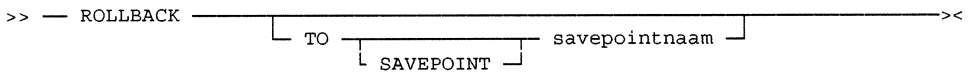
Het volgende statement zorgt ervoor dat alle wijzigingen die tijdens de huidige transactie uitstaan, definitief worden vastgelegd.



Uitstaande locks op de tabellen en rijen in kwestie worden vrijgegeven. Vanaf dit moment gelden de wijzigingen ook voor andere sessies. COMMENT specificeert eventueel het commentaar bij de huidige transactie. Deze commentaartekst mag maximaal 50 karakters lang zijn en moet tussen aanhalingstekens staan.

Wees voorzichtig met het geven van een commit terwijl er nog een cursor openstaat met een FOR UPDATE clausule, dus binnen de loop bijvoorbeeld. Elke volgende FETCH zal dan fout gaan; we moeten zulke cursoren eerst sluiten.

ROLLBACK zorgt ervoor dat alle wijzigingen die binnen de huidige transactie (sinds de laatste ROLLBACK of COMMIT) worden teruggedraaid. Nog niet vastgelegde wijzigingen gaan dus verloren. De syntax luidt als volgt:



Wanneer we geen TO SAVEPOINT clausule gebruiken, wordt de hele transactie ongedaan gemaakt en worden alle daarbinnen gedefinieerde savepoints verwijderd. Alle uitstaande locks op tabellen en rijen in kwestie worden vrijgegeven. De transactie is daarmee beëindigd net als bij het COMMIT statement. Een eventuele volgende transactie begint daarna automatisch.

Als we refereren aan een savepoint dan worden alle wijzigingen sinds dat savepoint ongedaan gemaakt. Alle savepoints gedefinieerd na dat savepoint gaan verloren.

Voorbeeld:

De werking van savepoints laat zich het makkelijkst illustreren in SQL Developer. We gaan een aantal SQL statements uitvoeren, waardoor één en ander duidelijk zal worden. We plaatsen eerst een savepoint zodat we later de transactie tot dit punt kunnen terugdraaien. Voer hiervoor onderstaand statement uit in een lege SQL Worksheet met behulp van de knop Run Script:

```
savepoint vlag_1;
```

Na afloop verschijnt de volgende melding in het Script Output window:

```
savepoint vlag_1 succeeded.
```

Vervolgens gaan we een update uitvoeren in een lege SQL Worksheet:

```
update p_kantoren
set    naam = lower(naam);
```

Als het goed is verschijnt de volgende melding in het Script Output window:

```
4 rows updated.
```

Met de onderstaande code maken we vervolgens een tweede savepoint aan:

```
savepoint vlag_2;
```

Na afloop verschijnt de volgende melding in het Script Output window:

```
savepoint vlag_2 succeeded.
```

Voer vervolgens het onderstaande PL/SQL-blok uit met daarin een aantal updates

```
declare
  cursor c_prijs is
    select *
    from   p_cursussen
    for update of prijs;
  v_verhoging number := 1;
begin
  for r_prijs in c_prijs loop
    if    r_prijs.prijs <= 500  then
      v_verhoging := 1.05;
    elsif r_prijs.prijs <= 1000 then
      v_verhoging := 1.04;
    elsif r_prijs.prijs <= 2000 then
      v_verhoging := 1.03;
    else
      v_verhoging := 1.02;
    end if;
    update p_cursussen
    set prijs = prijs * v_verhoging
    where current of c_prijs;
  end loop;
end;
/

Anonymous block completed
```

We kijken wat de updates voor resultaat hebben gegeven. De kantoornamen zijn aangepast en de prijzen zijn verhoogd:

```
select *
from   p_cursussen;
```

	NR	NAAM	AANT_DAG	PRIJS
1	1	SQL 5 dagen	5	2295
2	2	PL/SQL	2	1029.6
3	3	Oracle Forms	6	3029.4
4	4	Oracle Reports	4	2039.4
5	5	Developer casus	2	988
6	6	Oracle Eindgebruiker	1	519.75
7	7	Oracle Database Administrator	10	4590
8	8	Oracle Applicatiebouwer	20	9180

```
select *
from   p_kantoren;
```

	KANTNR	NAAM	PLAATS
1	10	boekhouding	AMSTERDAM
2	20	onderzoek	UTRECHT
3	30	verkoop	DEN HAAG
4	40	productie	ARNHEM

Door het uitvoeren van een rollback tot savepoint vlag_2 maken we het aanpassen van de prijzen ongedaan (en dus niet het aanpassen van de kantoornamen):

```
rollback to vlag_2;
```

Na afloop verschijnt de volgende melding in het Script Output window:

```
rollback completed.
```

Bekijk vervolgens de inhoud van de tabel P_CURSUSSEN en P_KANTOREN:

```
select *
from   p_cursussen;

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

select *
from   p_kantoren;

   KANTNR NAAM           PLAATS
   ------ -------------- ----------
 1     10 boekhouding    AMSTERDAM
 2     20 onderzoek      UTRECHT
 3     30 verkoop        DEN HAAG
 4     40 productie      ARNHEM
```

Zoals we zien is de update van de kantoornamen nog niet teruggedraaid.
We voeren nu een rollback uit tot savepoint vlag_1:

```
rollback to vlag_1;
```

Na afloop verschijnt de volgende melding in het Script Output window:

```
rollback completed.
```

Bekijk nogmaals de inhoud van de tabel P_KANTOREN:

```
select *
from   p_kantoren;

   KANTNR NAAM           PLAATS
   ------ -------------- ----------
 1     10 BOEKHOUDING    AMSTERDAM
 2     20 ONDERZOEK      UTRECHT
 3     30 VERKOOP        DEN HAAG
 4     40 PRODUKTIE      ARNHEM
```

Alles is nu teruggedraaid. Merk op dat door het teruggaan naar savepoint vlag_1 het savepoint vlag_2 nu niet meer bestaat.

5.3.1 Autonome transacties

Een transactie is een serie SQL statements die wordt afgesloten door een COMMIT of een ROLLBACK. Alle statements binnen een transactie worden op dat moment vastgelegd of teruggedraaid, het is niet mogelijk om één of meerdere statements daarbij uit te sluiten. Binnen applicaties kan het toch nodig zijn dat bepaalde statements altijd worden vastgelegd onafhankelijk van wat er met de andere statements in de transactie gebeurt. Vanaf Oracle8i is dat mogelijk in de vorm van autonome transacties.

Autonome transacties zijn transacties die onafhankelijk binnen een andere transactie kunnen worden aangeroepen. Met een autonome transactie is het mogelijk om tijdelijk uit de huidige transactie naar een nieuwe transactie te gaan die onafhankelijk van elkaar vastgelegd kunnen worden. Met de optie PRAGMA AUTONOMOUS_TRANSACTION wordt aangegeven dat het PL/SQL-blok als aparte transactie moet worden uitgevoerd.

Voorbeeld:

We gaan de tabel P_WERKNEMERS wijzigen en binnen dezelfde transactie een PL/SQL-blok uitvoeren dat een autonome transactie uitvoert. We starten een nieuwe transactie door de vorige af te sluiten met een COMMIT of ROLLBACK. Dit kan met een los statement, maar ook met behulp van de knoppen Commit (F11) en Rollback (F12).

Als eerste starten we dus een nieuwe transactie:

```
commit;
```

Na afloop verschijnt de volgende melding in het Script Output window:

```
commit succeeded.
```

Vervolgens maken we een nieuwe tabel:

```
create table naamfunctie
  ( naam varchar2(30)
   , functie varchar2(30)
  );
```

Na afloop verschijnt de volgende melding in het Script Output window:

```
create table succeeded.
```

Hierna zetten we de namen van de managers uit de tabel P_WERKNEMERS in kleine letters:

```
update p_werknemers
set    naam=lower(naam)
where  functie='MANAGER';
```

Na afloop verschijnt de volgende melding in het Script Output window:

```
3 rows updated.
```

Vervolgens gaan we een tweede (autonome) transactie starten. Voer hiervoor de onderstaande code uit in een lege SQL Worksheet:

```
declare
  pragma autonomous_transaction;
begin
  insert into naamfunctie
    values ( user
           , 'Cursist1'
           );
  commit;
end;
/

Anonymous block completed
```

Oracle heeft de autonome transactie direct afgesloten door de commit op regel voor de end;. Wanneer we nu uit de tabel NAAMFUNCTIE selecteren zien we dat de rij aan de tabel is toegevoegd:

```
select *
from   naamfunctie;

   NAAM                 FUNCTIE
   -------------------- --------------------
 1 <gebruikersnaam>     Cursist1
```

De eerste transactie staat nu echter nog open:

```
select *
from   p_werknemers
where  functie='MANAGER';

      PERSNR NAAM       FUNCTIE         MGR        SAL    TOESLAG     KANTNR
   --------- ---------- --------- --------- --------- --------- ---------
 1      3930 pieters    MANAGER        6221      3975                   20
 2      4621 klaasen    MANAGER        6221      3850                   30
 3      5810 heuvel     MANAGER        6221      3450                   10
```

We draaien de wijzigingen terug met een ROLLBACK.

```
rollback;
```

Na afloop verschijnt de volgende melding in het Script Output window:

```
rollback succeeded.
```

De naam van de managers staat weer in hoofdletters in de tabel en alle transactie zijn afgesloten.

```
select *
from   p_werknemers
where  functie='MANAGER';

      PERSNR NAAM       FUNCTIE         MGR        SAL    TOESLAG     KANTNR
   --------- ---------- --------- --------- --------- --------- ---------
 1      3930 PIETERS    MANAGER        6221      3975                   20
 2      4621 KLAASEN    MANAGER        6221      3850                   30
 3      5810 HEUVEL     MANAGER        6221      3450                   10

select *
from   naamfunctie;

   NAAM                 FUNCTIE
   -------------------- --------------------
 1 <gebruikersnaam>     Cursist1
```

De naam van de managers staat weer in hoofdletters in de tabel en alle transactie zijn afgesloten.

Om een autonome transactie te kunnen verlaten moet deze worden afgesloten met een COMMIT of een ROLLBACK. Als de autonome transactie wordt verlaten zonder dat deze wordt afgesloten volgt er een foutmelding en de transactie wordt teruggedraaid. We zullen nu een tweede autonome transactie starten zonder dat deze aan het eind wordt afgesloten.

```
declare
pragma autonomous_transaction;
begin
  insert into naamfunctie
    values ( user
           , 'Cursist2'
           );
end;
/
```

De volgende foutmelding zal in het Script Output window verschijnen:

```
Error starting at line 1 in command:
declare
pragma autonomous_transaction;
begin
  insert into naamfunctie
    values ( user
           , 'Cursist2'
           );
end;
Error report:
ORA-06519: active autonomous transaction detected and rolled back
ORA-06512: at line 8
06519. 00000 -  "active autonomous transaction detected and rolled back"
*Cause:    Before returning from an autonomous PL/SQL block, all autonomous
           transactions started within the block must be completed (either
           committed or rolled back). If not, the active autonomous
           transaction is implicitly rolled back and this error is raised.
*Action:   Ensure that before returning from an autonomous PL/SQL block,
           any active autonomous transactions are explicitly committed
           or rolled back.
```

Oracle heeft de autonome transactie teruggedraaid

```
select * from naamfunctie;

  NAAM                 FUNCTIE
  -------------------- --------------------
1 <gebruikersnaam>     Cursist1
```

5.4 Locking

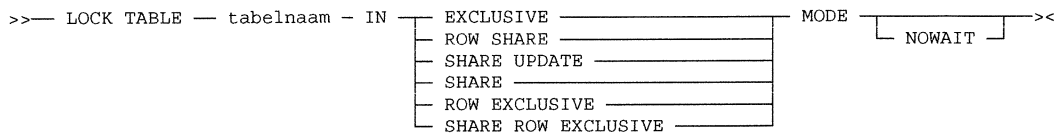
Het zou te ver voeren om de theorie van locking hier volledig te gaan bespreken. Hiervoor verwijzen we naar onze cursus 'Databasegebruik voor ontwikkelaars' of de cursus 'Oracle database: performance tuning'.

Hoewel Oracle automatisch de locking verzorgt kan de gebruiker ook expliciet locks plaatsen. De expliciete locks onderscheiden zich in strengheid.
Zo kan de ene lock verhinderen dat een andere lock wordt geplaatst. Dat is ook de essentie van locking: Wanneer een lock verhindert dat iemand anders de tabel gaat wijzigen, komt dat door verhindering van de (impliciete) lock die voorafgaat aan die wijziging.

Expliciete locks worden geplaatst m.b.v. SQL statements. We kennen er twee :
- LOCK TABLE
- SELECT ... FOR UPDATE ...

Met het LOCK TABLE statement kunnen we een tabel in zes verschillende standen (LOCK MODES) vergrendelen.

Syntax:



Uitleg van de verschillende lock-modes:

Exclusive
Om een tabel geheel voor uzelf te reserveren, zodat u de enige bent die wijzigingen mag doen. Anderen mogen hooguit raadplegen.

Row Share
Staat toe dat anderen gelijktijdig in dezelfde tabel mogen wijzigen, zolang het niet dezelfde rijen zijn. Ze mogen geen Exclusive lock plaatsen, wel andere locks.

Share Update
Synoniem voor Row Share. Wordt gebruikt in oudere Oracle versies en blijft voorlopig gehandhaafd voor compatibiliteit. Aardig om te weten is dat de schermtabellen in SQL*Forms default deze lock krijgen.

Share
Verhindert dat anderen gelijktijdig in dezelfde tabel kunnen wijzigen doordat ze geen Row Exclusive lock mogen plaatsen. Evenmin mogen ze een Exclusive of een Share Row Exclusive lock plaatsen. Ze mogen wel alvast de andere twee soorten locks plaatsen, maar zullen moeten wachten met wijzigen.

Row Exclusive
Lijkt veel op de Row Share lock, maar is iets strenger omdat men ook geen Row Share lock mag plaatsen. Deze lock wordt overigens impliciet geplaatst bij wijzigingen.

Share Row Exclusive
Is iets strenger dan de Share lock, omdat nu ook geen Share lock is toegestaan. Andere gebruikers mogen slechts raadplegen en een Row Share lock plaatsen.

Een extra optie in het LOCK TABLE statement is NOWAIT.
Sommige locks mogen niet geplaatst worden zolang er op de tabel nog bepaalde locks uitstaan. We moeten dan wachten tot de tabel weer vrijgegeven wordt. Er is dan sprake van een WAIT LOCK. We krijgen geen waarschuwing te zien en de wait lock kan in principe oneindig lang duren. Met NOWAIT kan er geen wait lock ontstaan. Als de tabel toevallig bezet is, wordt uw poging afgebroken en we zullen het statement later opnieuw moeten aanroepen. Bovendien wordt er een fout gegenereerd, welke we met een internal exception kunt afvangen.

De tweede manier om expliciet een table lock te plaatsen is met het gebruik van een cursor die gedefinieerd is met het SELECT ... FOR UPDATE ... statement.

Wat is nou het nut van het zelf plaatsen van locks, wanneer men bedenkt dat Oracle dit allemaal automatisch regelt?

- Met SELECT FOR UPDATE [OF] reserveren we op een selectieve manier een aantal rijen, waarna we gewoonlijk vervolgen met een DELETE of UPDATE WHERE CURRENT OF statement. Bij het openen van de cursor worden de rijen die door het SELECT statement geselecteerd worden alvast gelockt.
  Met de WHERE CURRENT OF clausule in een UPDATE of DELETE statement wordt de laatste rij die met FETCH is opgehaald gemuteerd. De gebruiker weet op deze manier zeker dat de betreffende rij niet door een andere gebruiker wordt veranderd.

- Een Exclusive lock komt van pas wanneer we niet wensen dat anderen tegelijkertijd in de tabel gaan wijzigen. Niet alleen omdat de tabelgegevens stabiel moeten blijven gedurende de hele transactie, maar ook omdat we elke rij willen kunnen benaderen: niemand kan rijen gaan locken.

- Een Share lock zouden we om dezelfde reden kunnen plaatsen. Het verschil met Exclusive is, dat anderen alvast locks kunnen plaatsen, namelijk Share en Row Share. Dit is beter voor de performance en verdient de voorkeur wanneer de transactie meer uit selecties dan uit wijzigingen bestaat.

- Een Share Row Exclusive lock verhindert alle locks behalve een Row Share lock. En kan gebruikt worden bij het raadplegen van de hele tabel om daarbij selectieve wijzigingen uit te gaan voeren.

- Een Row Share lock (ofwel Share Update lock) is de manier van locken die automatisch plaatsvindt op de schermtabellen in SQL*Forms applicaties.
  Deze lock lijkt veel op de impliciete Row Exclusive lock. De Row Share lock is wat minder streng, omdat anderen nog Share locks en Share Row Exclusive locks mogen plaatsen.

In een multi-user omgeving dienen we de hieronder genoemde regels na te streven. De reden daarvan is tweeërlei: het is beter voor de performance (update statements die op locks moeten wachten kosten immers tijd), en het voorkomt DEADLOCKS. Een deadlock is een situatie waarin twee locks op elkaar wachten, waardoor er theoretisch eeuwig gewacht kan worden. Oracle herkent deze situaties en draait één van de twee transactie terug (rollback).

Desalniettemin is het een onwenselijke situatie die zoveel mogelijk vermeden dient te worden.

- Gebruik alleen expliciete locks wanneer het echt nodig is.
- Probeer daarbij de minst strenge locks te plaatsen.
- Lock in verschillende programma's de tabellen in een vaste volgorde.
- Wacht niet te lang met een commit c.q. rollback. We moeten deze statements expliciet opgeven, want de beëindiging van een PL/SQL programma gaat niet automatisch gepaard met de afsluiting van een uitstaande transactie (of het moet zo zijn dat de optie AUTOCOMMIT is geactiveerd, dan wordt namelijk na elk statement of PL/SQL-blok een commit gegeven).

5.4.1 Lees-consistentie

Normaal gesproken past het Oracle RDBMS leesconsistentie toe op statementniveau. Dit houdt in dat zolang we aan het lezen zijn uit een tabel, de wijzigingen die tijdens de leesactie worden uitgevoerd niet worden getoond. De set records die we te zien krijgen is de set die aan het begin van de transactie in de tabel aanwezig was. Stel nu dat we in een programma meerdere tabellen raadplegen in verschillende statements, en we willen dat al deze statements als één transactie beschouwd worden, omdat de tabellen die geraadpleegd worden onderlinge relaties hebben. Oracle heeft hiervoor een voorziening in de vorm van het SET TRANSACTION READ ONLY statement. Het kan gebruikt worden in SQL*Plus en in PL/SQL.

We willen bijvoorbeeld een compleet rapport maken van alle ziekenhuizen, dus inclusief stafleden, afdelingen, bezetting en patiënten. Het is dan ongewenst dat gelijktijdige wijzigingen door andere gebruikers zichtbaar worden in het rapport. Denk maar aan het wijzigen van de ziekenhuisnummers in een tabel P_BEZETTING teneinde de patiënten te verhuizen, terwijl we net de betreffende patiënten hebben afgedrukt. Zonder leesconsistentie op transactieniveau zouden we de patiënten nog eens te zien krijgen! Door nu in het programma het voornoemde statement te gebruiken schermen we het programma af voor tussentijdse wijzigingen.

We zouden ook een Share lock kunnen plaatsen op de vijf tabellen. Maar daarmee vertragen we de gegevensverwerking. Gebruik liever dit statement:

```
set transaction read only ;
```

Op de tabellen wordt nu geen lock gelegd, maar Oracle zet de inhoud van de tabellen die benaderd worden binnen de komende transactie in het geheugen. Andere gebruikers kunnen de tabellen in de database dus gewoon wijzigen. Daarna beginnen we pas met de afzonderlijke queries. DML statements zijn niet toegestaan.

Pas na een commit of een rollback wordt de leesconsistente transactie afgesloten. Dat klinkt vreemd, omdat een query niets wijzigt. De commit c.q. rollback heeft dan ook geen enkele gevolgen voor de gegevens in de database, de tabellen die gebruikt zijn in deze transactie worden uit het geheugen gehaald. Maar formeel gesproken horen queries ook tot de categorie DML statements. Een transactie kan dus in principe louter uit queries bestaan. Het SET TRANSACTION statement moet het eerste SQL statement zijn binnen een transactie.

Tijdens de read only transactie worden leesbuffers gebruikt, de zgn. ROLLBACK SEGMENTEN. Deze kunnen vol raken, met name bij langdurige transacties, omdat die buffers tegelijk door anderen gebruikt worden. Het gevolg is dat er vanaf dat moment geen wijzigingen meer doorgevoerd kunnen worden, zonder dat men weet wat er aan de hand is. Neem dus toch een COMMIT of ROLLBACK statement op in uw programma wanneer we het SET TRANSACTION statement toepassen; PL/SQL doet dat niet impliciet na beëindiging van het programma.

6 Foutafhandeling

6.1 Inleiding

Een PL/SQL programma kent een aantal soorten fouten. Syntax fouten worden al tijdens de compilatie herkend, dit zijn bijvoorbeeld typfouten of het vergeten van een puntkomma. Semantische fouten, zijn fouten binnen een syntactisch correct programma, waardoor het programma niet doet wat we verwachten. We merken ze pas op wanneer het eindresultaat anders is dan we bedoeld hebben en er moet dan ergens een logische fout in het programma verscholen liggen.

De volgende cursor zal bijvoorbeeld nooit rijen ophalen omdat er in WHERE-clausule naam='DEN HAAG' staat terwijl Den Haag een plaats is.

```
cursor c_semantiek is
  select naam
  from   p_kantoren
  where  naam='DEN HAAG';
```

Deze foutsituaties bedoelen we nu niet; ze kunnen trouwens ook niet worden ondervangen. De programmeur moet de onjuistheden zelf herstellen. De fouten die dit hoofdstuk behandelt zijn geen echte fouten, maar eerder uitzonderingssituaties. In PL/SQL heeft men daar een passende term voor: Exceptions.

6.2 Exceptions

We kennen twee soorten exceptions: Predefined Exceptions en User Defined Exceptions.

Predefined Exceptions handelen allemaal over Oracle fouten. Ze worden soms 'Internal Exceptions' of 'Predefined Internal Exceptions' genoemd. Voorbeelden van Predefined Exceptions zijn:

- een geopende cursor wordt nog een keer geopend.
- de noemer in een breuk krijgt de waarde nul;
- een uniek geïndexeerde kolom krijgt een dubbele waarde .

User Defined Exceptions moeten we noemen in de declaratiesectie. Voorbeelden van User Defined Exceptions zijn:

- een zelf gedefinieerde limiet wordt overschreden;
- een waarde voldoet niet aan een proef;
- een naam staat niet in hoofdletters.

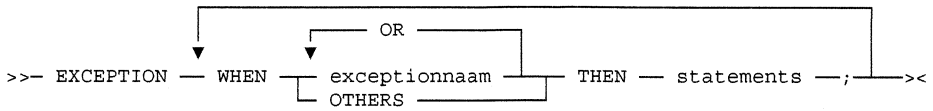
Syntax:

De declaratie van User Defined Exceptions heeft de volgende syntax:

```
>>— exceptionnaam —— EXCEPTION ——;——><
```

De exceptionnaam mogen we zelf bepalen. Het aanheffen van de exception vindt plaats in de programmasectie en de afhandeling in de foutafhandelingsectie.

De foutafhandelingsectie heeft de volgende syntax:



Predefined Exceptions hoeven we niet te declareren en worden door Oracle zelf gegenereerd. Hier volgt een lijst van alle Predefined Exceptions :

CURSOR_ALREADY_OPEN	De cursor was reeds open.
DUP_VAL_ON_INDEX	Een unieke index weigert een dubbele sleutel
INVALID_CURSOR	De cursor was niet gedefinieerd of geopend
INVALID_NUMBER	Data conversie van string naar getal ging fout
LOGIN_DENIED	Ongeldige gebruikersidentificatie
NO_DATA_FOUND	Een query gaf geen rijen
NOT_LOGGED_ON	Niet aangelogd tijdens aanroepen van een SQL statement
PROGRAM_ERROR	Interne PL/SQL fout
STORAGE_ERROR	Geheugentekort
TIMEOUT_ON_RESOURCE	De plaatsing van een lock wordt verhinderd
TOO_MANY_ROWS	Een SELECT INTO statement gaf meer dan één rij
TRANSACTION_BACKED_OUT	Een transactie wordt impliciet of expliciet gerollbackt.
VALUE_ERROR	Fout tijdens de verwerking van een waarde
ZERO_DIVIDE	Deling door nul

De WHEN OTHERS exception staat niet in dit rijtje, omdat het alle overige internal exceptions omvat die niet Predefined (die dus niet in bovenstaand rijtje staan) zijn of niet worden genoemd in de EXCEPTION sectie. De WHEN OTHERS clausule moet ofwel de enige clausule zijn ofwel helemaal achteraan komen in de foutafhandeling-sectie.

De WHEN clausule in de foutafhandelingsectie roept een foutenroutine aan, namelijk de statements achter het THEN keyword. Men noemt ze ook wel Exception handlers.

Voorbeeld:

```
declare
  cursor c_fout is
    select *
    from   p_werknemers;
  r_werknemer c_fout%rowtype;
begin
  open  c_fout;
  fetch c_fout into r_werknemer;
  for r_fout in c_fout loop
    insert into hulptabel
      values ( r_fout.persnr
             , r_fout.sal
             , r_fout.naam
             );
  end loop;
  close c_fout;
exception
  when cursor_already_open then
    insert into hulptabel
      values ( r_werknemer.persnr
             , null
             , 'De cursor was al geopend.'
             );
  when invalid_cursor then
    insert into hulptabel
      values ( null
             , null
             , 'Na de FOR-loop wordt de cursor al automatisch gesloten'
             );
end;
/
```

Dit programma plaatst een foutregel in de HULPTABEL, omdat de cursor c_fout bij het uitvoeren van het FOR commando nog open was.

Nadat een exception-routine is uitgevoerd eindigt het PL/SQL-blok en gaat de besturing verder met statements buiten dat blok. Bijvoorbeeld een eromheen liggend PL/SQL-blok of de rest van een host programma waarin het blok genest is.

Wanneer het PL/SQL-blok genest is in een ander PL/SQL-blok en er wordt een exception aangeheven terwijl het blok daar geen exception handler voor heeft, dan wordt gekeken naar de corresponderende handler in het omliggende blok.

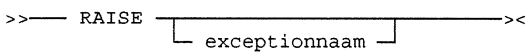
6.3 RAISE

Raise betekent "aanheffen". Met dit statement wordt expliciet een exception aangeheven. Zodra zich een exception voordoet (en wordt aangeheven) wordt de programmasectie afgebroken en wordt de Exception Handler (de code achter THEN) uitgevoerd.
Exceptions zijn dus storingslabels die in de foutafhandelingsectie worden aangegrepen door Exception Handlers. Bij het controleren op storingen moeten we onderscheid maken tussen SQL statements en procedurele statements.

Fouten in SQL statements gaan automatisch gepaard met een Internal Exception (of Predefined Exceptions) ofwel een Oracle error. We hebben zojuist gezien dat PL/SQL beschikt over een lijst van Predefined Exceptions. Deze lijst is erg kort in verhouding tot het totaal aan Oracle errors dat kan optreden, maar het bevat wel de meest voorkomende.

User Defined Exceptions handelen geen Oracle errors af. User Defined Exceptions moeten we zelf aanheffen met het RAISE statement, omdat PL/SQL dat niet automatisch doet. Predefined Exceptions kunnen we ook zelf met RAISE aanheffen.

Syntax:



We zien hieraan dat we de keuze hebben om één of geen exception mee te geven.
Het RAISE statement zonder exceptions mag alleen in een Exception Handler (in de foutafhandelingsectie dus) en heeft tot gevolg dat geprobeerd wordt de corresponderende exception in het omliggende blok aan te heffen.

Laten we het toepassen van exceptions met RAISE illustreren aan de hand van een voorbeeld: het controleren van banknummers.
Een banknummer is een getal van negen cijfers dat moet voldoen aan een speciale voorwaarde; het eerste cijfer wordt met negen vermenigvuldigd, het tweede met acht, het derde met zeven, enz. De som van deze producten moet deelbaar zijn door elf. We tonen een SQL-commandobestand waarin een PL/SQL-blok genest is.

Voorbeeld

```
accept a_bank number prompt 'geef het banknummer a.u.b. :'

declare v_subtotaal  number := 0;
        v_banknummer number ;
        e_niet_negen exception ;
        e_negatief   exception ;
        e_elfproef   exception ;

begin
  v_banknummer := &a_bank;
  if v_banknummer < 0 then
    raise e_negatief;                          --de exception e_negatief aanheffen
  elsif length(v_banknummer) <> 9 then
    raise e_niet_negen;                        --de exception e_niet_negen aanheffen
  end if;

  for t_nummer in 1..9 loop
    v_subtotaal := v_subtotaal + (10 - t_nummer) *
      (to_number(substr(to_char(v_banknummer),t_nummer,1)));
  end loop;

  if mod(v_subtotaal,11) <> 0 then
    raise e_elfproef;                          --de exception e_elfproef aanheffen
  else
    insert into hulptabel
      values( v_banknummer
            , null
            , 'banknummer geaccepteerd'
            );
  end if;

exception
  when e_niet_negen then
    insert into hulptabel
      values ( v_banknummer
             , null
             , 'banknummer <> 9 cijfers'
             );
  when e_negatief then
    insert into hulptabel
      values ( v_banknummer
             , null
             , 'banknummer negatief'
             );
  when e_elfproef then
    insert into hulptabel
      values ( v_banknummer
             , null
             , 'banknummer geweigerd'
             );
end;
/
```

Een korte toelichting bij de code:

- De eerste IF conditie kijkt of het een positief getal is en of het opgegeven nummer uit 9 cijfers bestaat.
- In de FOR-loop worden de afzonderlijke cijfers van het gegeven getal vermenigvuldigd met 10 minus de indexwaarde van de FOR-loop en bij elkaar opgeteld.
- De tweede IF conditie kijkt of we de som van de cijfers door 11 kunnen delen. We gebruiken hiervoor de restwaarde van de deling van de som van de cijfers en getal 11.
- Voldoet het nummer of de som van de afzonderlijke cijfers niet aan de voorwaarden, dan wordt met een RAISE een exception aangeroepen.

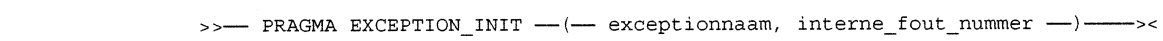
6.4 EXCEPTION_INIT

In de foutafhandeling-sectie kunnen we met de WHEN OTHERS clausule refereren aan alle interne exceptions welke niet met name worden aangehaald. Degene die we wel kunnen noemen, moeten voorkomen in de lijst van Predefined Exceptions.
Deze lijst is niet volledig: het bevat lang niet alle mogelijke foutsituaties die intern kunnen optreden. Maar de WHEN OTHERS clausule vangt ook die exceptions af. Dus alle mogelijke storingen worden daarmee op één grote hoop gegooid.

Misschien willen we wat genuanceerder te werk gaan - met andere woorden - we willen een interne exception die niet voor gedefinieerd is toch met name noemen in de foutafhandeling-sectie. Daartoe zullen we de reeks Predefined Exceptions moeten uitbreiden. We maken dus User Defined Internal Exceptions: storingslabels gebaseerd op Oracle fouten.

Het EXCEPTION_INIT statement geeft ons daartoe de mogelijkheid.

Syntax:



Termen:

PRAGMA	Een keyword dat verplicht is en de compiler vertelt dat het hier niet handelt om een normaal statement dat gerund moet worden, maar om een informatief statement voor de compiler. Zo'n statement wordt een PRAGMA genoemd.
exception_naam	De naam van uw zelf gedefinieerde interne exception. Deze exception moet daarvoor apart zijn gedeclareerd.
interne_fout_nummer	Een Oracle foutnummer. Oracle fouten herkennen we aan het formaat ORA-99999. We zijn ze vast al eens tegengekomen. Een voorbeeld: `ORA-00937 not a single-group group function` Gebruik alleen het nummer, maar zonder het voorvoegsel ORA. Voorloopnullen mogen we weglaten. Het nummer moet negatief zijn.

Het EXCEPTION_INIT statement moeten we opnemen in de declaratiesectie.
Een lijst van foutnummers vinden we in de bij het Oracle pakket meegeleverde Error Messages and Codes Manual of via de installeren Online Help bestanden van Oracle.

Voorbeeld:

```
declare
  cursor c_datum is
    select to_date('31-DOC-1999','DD-MON-YYYY')
    from dual;
  v_datum       date ;
  e_mijnfout    exception;
  pragma exception_init (e_mijnfout , -1843 ) ;
begin
  open c_datum;
  fetch c_datum into v_datum;
  close c_datum;
exception
  when e_mijnfout
    then insert into hulptabel
      values ( null
             , null
             , 'De naam van de maand is onjuist'
             );
  when others then
    insert into hulptabel
      values ( null
             , null
             , 'Oracle error'
             );
end;
/
```

We zien dat Oracle-foutmelding `ORA-01843: not a valid month` (omdat niet 'DEC', maar 'DOC' als maand is opgegeven) wordt vervangen door een insert in de hulptabel.

We mogen ook Predefined Exceptions herdefiniëren.
Onderstaande lijst toont de bijbehorende foutnummers.

Exception	Oracle foutnummer	Te gebruiken foutnummer
NOT_A_VALID_MONTH	-01843	-1843
CURSOR_ALREADY_OPEN	-06511	-6511
DUP_VAL_ON_INDEX	-00001	-1
INVALID_CURSOR	-01001	-1001
INVALID_NUMBER	-01722	-1722
LOGIN_DENIED	-01017	-1017
NO_DATA_FOUND	-01403	+100
NOT_LOGGED_ON	-01012	-1012
PROGRAM_ERROR	-06501	-6501
STORAGE_ERROR	-06500	-6500
TIMEOUT_ON_RESOURCE	-00051	-51
TRANSACTION_BACKED_OUT	-00061	-61
TOO_MANY_ROWS	-01422	-1422
VALUE_ERROR	-06502	-6502
ZERO_DIVIDE	-01476	-1476

De tabel laat een uitzondering zien bij NO_DATA_FOUND. Bij die Oracle fout mogen we niet -1403 gebruiken maar in plaats daarvan +100. Dit heeft een historische reden.
De keuze in foutnummers is niet geheel vrij: sommige fouten beschouwt PL/SQL als zgn. Fatal Errors. Deze mogen niet meedoen. Bijvoorbeeld:

```
ORA-00942: table or view does not exist
ORA-00904: invalid column name
```

We kunnen deze ook niet afvangen met de WHEN OTHERS clausule in de foutafhandeling-sectie. Maar dat geeft niet: PL/SQL herkent deze fouten reeds bij de compilatie; ze kunnen dus ook niet optreden.
Ook bestaande programma's zullen niet meer draaien wanneer later dergelijke fouten kunnen optreden (bijv. een tabel droppen die in een PL/SQL programma genoemd wordt).

Er is wel iets te zeggen over het soort Oracle fouten dat PL/SQL als Fatal beschouwt.
In de ranges

```
ORA-00900 t/m ORA-00999 , ORA-01700 t/m ORA-01799 ,
ORA-01900 t/m ORA-02099 , ORA-02140 t/m ORA-02499
```

hebben we te maken met de zgn. PARSE ERRORS. Dat zijn fouten die kunnen optreden zodra een SQL statement op juistheid wordt gecontroleerd. Denk daarbij vooral aan syntax, objectnamen en toegangsrechten. Doorgaans zijn dit soort fouten niet te ondervangen m.b.v. exceptions, uitzonderingen daargelaten.

Er zijn allerlei soorten Oracle fouten. EXECUTION ERRORS kunnen bijvoorbeeld pas optreden nadat het SQL statement eenmaal is geaccepteerd. Een typisch voorbeeld van zo'n fout is:

```
ORA-01403 : no data found.
```

In de Online Help van in het Startmenu vinden we een opsomming van de Oracle foutnummers.

6.5 SQLCODE en SQLERRM

Dit zijn geen statements maar functies, want ze voeren geen opdracht uit maar retourneren een uitkomst. De functies SQLCODE en SQLERRM bevatten respectievelijk het foutnummer en de daarmee corresponderende Oracle melding bij het huidige SQL statement.

De waarde van SQLCODE is 0 wanneer het statement succesvol verlopen is. Dat wil zeggen: er werd geen Internal Exception of een User Defined Exception aangeheven.
De waarde van SQLERRM is dan:

```
ORA-00000: normal, succesful completion
```

De waarde van SQLCODE is +1 wanneer het statement geen Internal Exception aanhief, dus ook geen User Defined Internal Exception, maar wel een gewone User Defined Exception.
De waarde van SQLERRM is dan:

```
User-Defined Exception
```

We kunnen de functies SQLCODE en SQLERRM niet rechtstreeks gebruiken in een SQL statement. Gebruik hulpvariabelen zoals in het volgende bijvoorbeeld:

Voorbeeld:

```
declare
  v_foutcode  number(4);
  v_boodschap varchar2(80);
begin
  v_foutcode  := sqlcode ;
  v_boodschap := sqlerrm ;
  insert into hulptabel
    values ( v_foutcode
           , null
           , v_boodschap
           );
end;
/
```

Die hulpvariabelen moeten we natuurlijk vooraf gedeclareerd hebben.
Let hierbij wel op de veldafmetingen:

- voor de code: numeriek, minimaal 4 posities
- voor de melding: alfanumeriek, minimaal 80 posities.

Hieronder volgen nog enkele voorbeelden.

Voorbeeld 1: succesvol SQL statement.

```
declare
  cursor c_goed is
    select 1
    from   dual;
  v_hulpveld number;
  v_melding  varchar2(80);
  v_code     number;

begin
  open c_goed;
  fetch c_goed into v_hulpveld;
  close c_goed;
  v_code    := sqlcode;
  v_melding := sqlerrm;
  insert into hulptabel
    values ( v_code
           , null
           , v_melding
           );
exception
  when others then
    v_code    := sqlcode;
    v_melding := sqlerrm;
    insert into hulptabel
      values ( v_code
             , null
             , v_melding
             );
end;
/
```

De foutafhandelingsectie is hier aanwezig, zodat als er iets fout gaat het programma toch wordt afgewerkt en de foutmelding in de hulptabel wordt gezet. Vooral bij procedures waarin meerdere PL/SQL-blokken zitten is dit handig. Anders wordt bij een foutsituatie de hele transactie teruggedraaid.

Voorbeeld 2: User Defined Exception.

```
declare
  cursor c_datum is
    select sysdate + 1
    from dual;
  v_datum   date;
  v_melding varchar2(80);
  v_code    number;
  e_te_laat exception;
begin
  open c_datum;
  fetch c_datum into v_datum;
  close c_datum;
  if v_datum > sysdate
    then raise e_te_laat;
  end if;
exception
  when e_te_laat then
    v_code    := sqlcode;
    v_melding := sqlerrm;
    insert into hulptabel
      values ( v_code
             , null
             , v_melding
             );
  when others then
    v_code    := sqlcode;
    v_melding := sqlerrm;
    insert into hulptabel
      values ( v_code
             , null
             , v_melding
             );
end;
/
```

Voorbeeld 3: Predefined (Internal) Exception.

```
declare
  cursor c_datum is
    select to_date('31-XXX-1999')
    from   dual;
  v_datum   date;
  v_melding varchar2(80);
  v_code    number;
begin
  open c_datum;
  fetch c_datum into v_datum;
  close c_datum;
exception
  when others then
    v_code    := sqlcode;
    v_melding := sqlerrm;
    insert into hulptabel
      values ( v_code
             , null
             , substr(v_melding,1,75)
             );
end;
/
```

N.B.: We nemen van V_MELDING enkel de eerste 75 karakters, omdat onze kolom in hulptabel maar 75 karakters breed is.

Bij de WHEN OTHERS EXCEPTION is het handig om de code en de melding van de fout op te nemen, omdat anders het vinden van de fout in het programma erg lastig is.

BIJLAGEN

Oracle Database:
PL/SQL voor ervaren programmeurs

A. Tabellen

tabel p_rekening

beschrijving

```
REK_NR          NUMBER(9)       NOT NULL
SAL             NUMBER(9,2)     NOT NULL
DATUM           DATE            NOT NULL
```

inhoud

```
    REK_NR        SAL DATUM
---------- ---------- ---------
         1       1000 31-DEC-93
         2       1000 31-DEC-93
         3       1000 31-DEC-93
         4       1000 31-DEC-93
         5       1000 31-DEC-93
```

tabel p_mutatie

beschrijving

```
VOLGNR          NUMBER(4)       NOT NULL
VAN_REK         NUMBER(9)       NOT NULL
NR_REK          NUMBER(9)       NOT NULL
BEDRAG          NUMBER(9,2)     NOT NULL
DATUM           DATE            NOT NULL
SOORT_TR        VARCHAR2(1)     NOT NULL
CODE            NUMBER(1)       NOT NULL
```

inhoud

```
    VOLGNR    VAN_REK     NR_REK     BEDRAG DATUM     S C
---------- ---------- ---------- ---------- --------- - -
         1          1          1        250 04-JAN-94 A 0
         2          2          2        500 05-JAN-94 A 0
         3          3          3        150 06-JAN-94 A 0
         4          4          5        250 07-JAN-94 O 0
         5          1          3        550 10-JAN-94 O 0
         6          4          4        350 11-JAN-94 B 0
         7          5          5        650 12-JAN-94 B 0
         8          5          1        250 13-JAN-94 O 0
         9          2          4         50 14-JAN-94 O 0
        10          6          1        750 17-JAN-94 O 0
        11          3          1       1750 18-JAN-94 O 0
        12          4          4         75 19-JAN-94 O 0
```

tabel p_kantoren

beschrijving

```
KANTNR          NUMBER(4)       NOT NULL
NAAM            VARCHAR2(25)
PLAATS          VARCHAR2(20)
```

inhoud

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

A. Tabellen

tabel p_werknemers

beschrijving

```
PERSNR          NUMBER(4)       NOT NULL
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_personen

beschrijving

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

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

A. Tabellen

tabel p_cursussen

beschrijving

```
NR              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_spelers

beschrijving

```
SPELNR          NUMBER(3)       NOT NULL
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

A. Tabellen

tabel p_wedstrijden

beschrijving

```
TEAMNR          NUMBER(2)       NOT NULL
SPELNR          NUMBER(3)       NOT NULL
GEWONNEN        NUMBER(3)       NOT NULL
VERLOREN        NUMBER(3)       NOT NULL
```

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

N.B. Het gaat om een tenniscompetitie, er kan dus niet gelijk gespeeld worden. Spelers mogen voor meerdere teams uitkomen. De kolommen gewonnen en verloren geven het totaal aantal gewonnen resp. verloren wedstrijden van de spelers voor dat team.

tabel p_teams

beschrijving

```
TEAMNR          NUMBER(2)       NOT NULL
SPELNR          NUMBER(3)       NOT NULL
DIVISIE         VARCHAR2(6)     NOT NULL
```

inhoud

TEAMNR	SPELNR	DIVISI
1	2	ere
2	27	tweede

tabel p_boetebedragen

beschrijving

```
BOETENR         NUMBER(2)       NOT NULL
SPELNR          NUMBER(3)       NOT NULL
DATUM           DATE
BEDRAG          NUMBER(7,2)
```

inhoud

BOETENR	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

A. Tabellen

tabel p_ziekenhuizen

beschrijving

```
ZIEKHNR         NUMBER(2)       NOT NULL
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

tabel p_afdelingen

beschrijving

```
ZIEKHNR         NUMBER(4)       NOT NULL
AFDNR           NUMBER(4)       NOT NULL
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

```
ZIEKHNR         NUMBER(2)       NOT NULL
AFDNR           NUMBER(2)       NOT NULL
PERSNR          NUMBER(4)       NOT NULL
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

A. Tabellen

tabel p_patienten

beschrijving

```
PATNR           NUMBER(5)       NOT NULL
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	11-12-66	M	3542764
12816	Schouten W.	Den Haag	23-04-73	V	7466384
19381	Elbers M.	Amsterdam	01-01-76	V	9753728
25218	Dekker B.	Utrecht	05-11-54	M	8466355
30940	Lammers T.	Arnhem	12-04-43	V	3452718
38911	Jong H.	Nijmegen	12-01-82	M	4656238
39410	Manders G.	Den Bosch	11-12-70	M	2794710
45630	Ravenhorst P.	Eindhoven	04-02-48	M	9872513
48220	Feenstra A.	Breda	27-02-77	V	3529976
50333	Horst E.	Utrecht	12-04-64	M	1232988

tabel p_bezetting

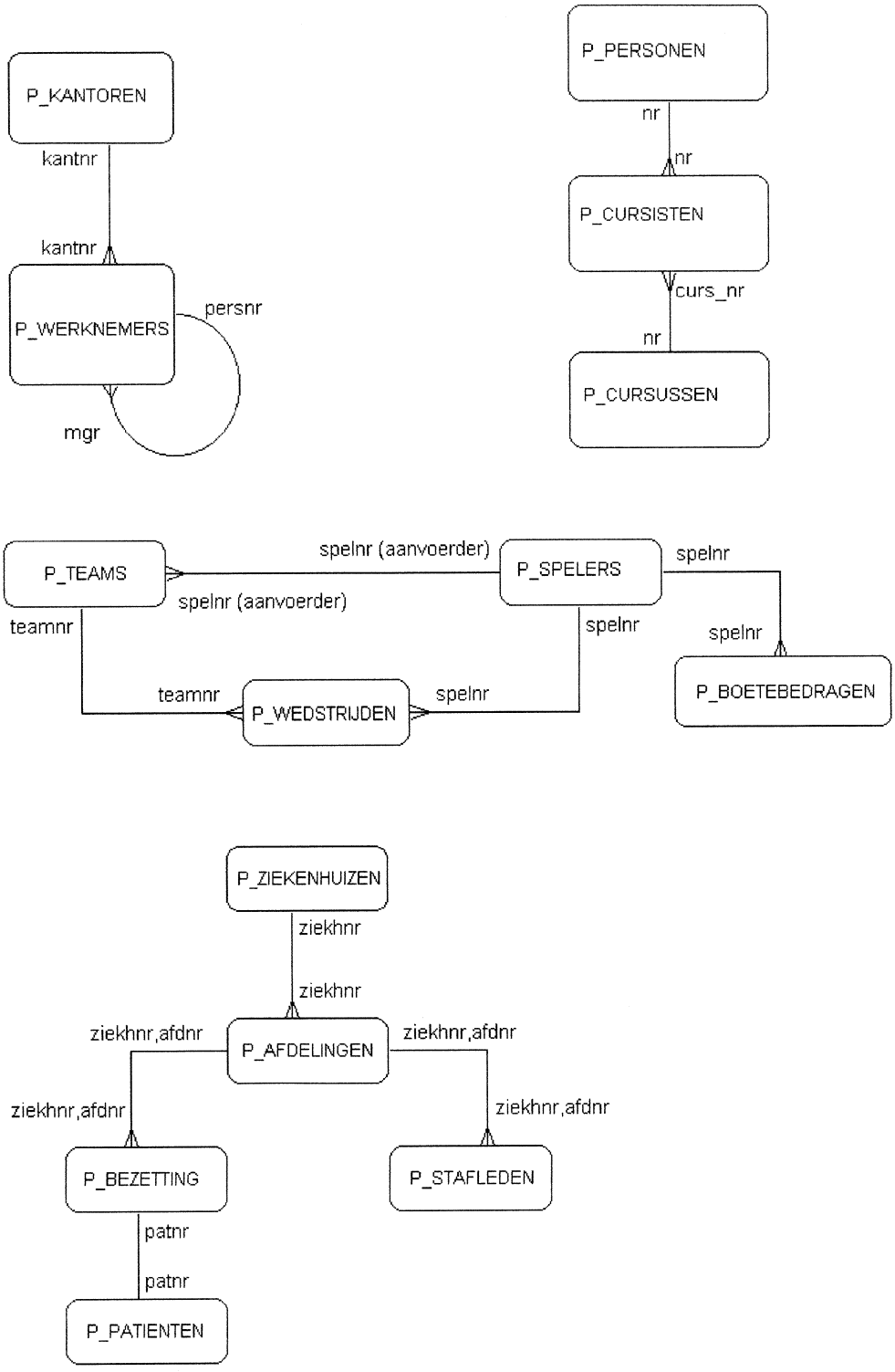
beschrijving

```
PATNR           NUMBER(5)       NOT NULL
ZIEKHNR         NUMBER(2)       NOT NULL
AFDNR           NUMBER(2)       NOT NULL
BEDNR           NUMBER(2)       NOT NULL
```

inhoud

PATNR	ZIEKHNR	AFDNR	BEDNR
11321	10	3	1
12816	10	3	2
19381	10	3	3
25218	15	4	1
30940	15	4	2
38911	20	6	1
39410	20	6	2
45630	20	6	3
48220	20	2	1
50333	25	4	1

B. Tabellen - relatiediagrammen



B. Tabellen - relatiediagrammen

C. Operatoren, functie en maskers

Rekenkundige operatoren

+	optellen
-	aftrekken of negatief getal aangeven
*	vermenigvuldigen
/	delen
(...)	zorgt voor een opgelegde volgorde van uitvoering

Syntax operatoren

&	geeft een substitutievariabele aan in een SQL statement
&&	geeft een substitutievariabele aan in een SQL statement en bewaart de ingevoerde waarde
(...)	geeft een subquery aan
'string'	geeft een alfanumerieke constante aan
"naam"	geeft een kolomnaam of alias aan waarin zich speciale tekens (bijvoorbeeld spaties) bevinden
@	voorvoegsel van een databaselink in de FROM clausule. Een databaselink is een verbinding naar een database. De apestaart kan tevens gebruikt worden als alias van het START commando in SQL*Plus.

Operatoren voor datumvelden

datum + n	n dagen bij datum optellen
datum - n	n dagen van datum aftrekken
dat2 - dat1	aantal tussenliggende dagen, de jongste datum het eerst vermelden

String operatoren

str1||str2	koppelt de strings str1 en str2 aan elkaar

Algemene operatoren

=	gelijk aan.
!= / < > / ^=	niet gelijk aan (afhankelijk van het type terminal).
>	groter dan
>=	groter dan of gelijk aan
<	kleiner dan
<=	kleiner dan of gelijk aan

C. Operatoren, functies en maskers

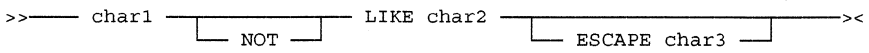
Operatoren voor woordpatronen

[NOT] LIKE met de volgende symbolen worden delen van een string aangegeven :
(ze worden ook wel 'wildcards' genoemd):

%	elke string van nul of meer willekeurige karakters
_ (underscore)	een willekeurig karakter

met de optie ESCAPE is het mogelijk naar een letterlijke % of _ te zoeken.

Syntax:



Voorbeeld:

```
select naam
from werknemers
where naam like 'i%%'.

NAAM
-------
ie%$aaa
i%$aaa
ie%$aa%

select naam
from werknemers
where naam like 'i%!%' escape '!'

NAAM
-------
ie%$aa%
```

Operatoren voor intervallen

`[NOT] BETWEEN waarde1 AND waarde2`
Voldoet (of voldoet niet) als een uitdrukking een waarde heeft die tussen waarde1 en waarde2 valt. Waarde1 en 2 tellen mee

`[NOT] IN (subselectie of lijst van waarden gescheiden door een komma)`
Voldoet (of voldoet niet) als de waarde van de uitdrukking in de lijst van waarde voorkomt. Mag niet in combinatie met een algemene operator worden gebruikt.

`IS [NOT] NULL`
Het '=' teken mag niet in combinatie met null worden gebruikt in een vergelijking

`<algemene operator> ANY of SOME (subselectie of lijst van waarden gescheiden door een komma)`
voldoet Als één van de waarden die achter any (of some) staat aan de algemene operator voldoet.

`<algemene operator> ALL (subselectie of lijst van waarden gescheiden door een komma)`
Voldoet als alle waarden die achter all staan aan de algemene operator voldoen

`[NOT] EXISTS (subselectie)`
Voldoet (of voldoet niet) als de subquery tenminste één rij als resultaat heeft

Opmerking
= ANY en = SOME zijn gelijkwaardig aan IN
!= ALL , <> ALL en ^= ALL zijn gelijkwaardig aan NOT IN

Condities kunnen met de operatoren AND en OR worden uitgebreid. U mag ronde haakjes gebruiken om de volgorde van afhandeling te beïnvloeden. AND geeft TRUE als linkerlid en rechterlid beide TRUE zijn. OR geeft TRUE als linkerlid of rechterlid TRUE is.

Groepsfuncties

AVG ([DISTINCT | ALL] uitdrukking)
(***AV**eraG**e***) Het gemiddelde van uitdrukking

COUNT ([DISTINCT | ALL] { * | uitdrukking })
Het aantal voorkomens van uitdrukking

MAX ([DISTINCT | ALL] uitdrukking)
De maximale waarde van uitdrukking

MIN ([DISTINCT | ALL] uitdrukking)
De minimale waarde van uitdrukking

STDDEV ([DISTINCT | ALL] uitdrukking)
(***STanDard DEViation***) De standaarddeviatie (= spreiding) van uitdrukking

SUM ([DISTINCT | ALL] uitdrukking)
De sommatie over uitdrukking

VAR[IANCE] ([DISTINCT | ALL] uitdrukking)
De variantie van uitdrukking

DISTINCT laat bij de berekening dubbele rijen weg.
Null values doen niet mee, behalve bij COUNT.

Rekenkundige functies

ABS (n)	(***ABS**olute value*) absolute waarde van n
ACOS (n)	De arccosinus van n ($-1 \leq n \leq 1$)
ASIN (n)	De arcsinus van n ($-1 \leq n \leq 1$)
ATAN (n)	De arctangens van n
CEIL (n)	Kleinste gehele getal groter dan of gelijk aan n
COS (n)	De cosinus van n (van een hoek in radialen)
COSH (n)	De cosinus hyperbolicus van n (van een hoek in radialen)
EXP (n)	(***EXP**onent*)e tot de n-de macht (e = 2.71828183...)
FLOOR (n)	Grootste gehele getal kleiner of gelijk aan n
LN (n)	De natuurlijke logaritme van n, n moet groter dan 0 zijn
LOG (m,n)	De logaritme van n met basis m. m kan elk positief getal zijn behalve 0 of 1, n kan elk positief getal zijn
MOD (m,n)	Rest van de deling m/n
POWER (m,n)	m tot de macht n. m en n kunnen elk willekeurig getal zijn. Echter als m negatief is dan moet n een integer zijn
ROUND (n[,m])	n afgerond tot m decimalen, indien m niet opgegeven wordt, wordt nul decimalen aangenomen. m mag negatief zijn om de cijfers links van de decimale komma af te ronden

`SIGN (n)`	-1 als n < 0 . 0 als n =0 . +1 als n > 0
`SIN (n)`	De sinus van n (van een hoek in radialen)
`SINH (n)`	De sinus hyperbolicus van n (van een hoek in radialen)
`SQRT (n)`	(***SQ**uare **RooT***) wortel van n. null indien n < 0
`TAN (n)`	De tangens van n (van een hoek in radialen)
`TANH (n)`	De tangens hyperbolicus van n (van een hoek in radialen)
`TRUNC (n[,m])`	(***TRUNC**ate)* als ROUND, ditmaal echter afgebroken op m decimalen

Functies met strings

`ASCII (string)`	ASCII of EBCDIC waarde van eerste karakter van string
`CHR (n)`	Het karakter met ASCII of EBCDIC waarde n
`CONCAT (str1,str2)`	(***CONCAT**enation*) De concatenatie van de strings str1 en str2 (vergelijk met ||).
`INITCAP (string)`	(***INIT**ial **CAP**ital*) de eerste letter van ieder woord in de string wordt omgezet in een hoofdletter, de overige letters worden kleine letters.
`INSTR (s1,s2 [,n | 1 [,m | 1 ]])`	(***IN STR**ing*) de positie van het m-de voorkomen van string s2 in string s1 startend op positie n.
`INSTRB (s1,s2 [,n[,m]])`	Identiek aan INSTR. Echter n en de returnwaarde worden weergegeven in bytes (dit maakt alleen verschil indien gebruik gemaakt wordt van een multi-byte karakterset)
`LENGTH (string)`	Lengte van de string
`LENGTHB (string)`	Identiek aan LENGTH. Echter de lengte van de string wordt in bytes weergegeven
`LOWER (string)`	Lowercase: string omzetten naar kleine letters
`LPAD (s1,n [,s2 | ' ' ] )`	(***Left PAD**ding*) string s1 links aangevuld tot lengte n met de karakters in string s2, indien s2 niet opgegeven is met spaties
`LTRIM (s1 [,s2 | ' ' ] )`	(***Left TRIM**ming*) string s1 waarbij de karakters die voorkomen in string s2 links verwijderd worden. default voor s2 is een spatie
`NLS_INITCAP (str [, 'nlsparams'] )`	(***National Language Support...***) Identiek aan INITCAP. Echter er kan een waarde gespecificeerd worden voor de NLS-parameter NLS_SORT. Deze parameter bepaalt onder andere de sorteer-volgorde van strings. De waarde van de parameter kan ofwel een taalafhankelijke sorteervolgorde zijn, ofwel het gereserveerde woord BINARY (in dit laatste geval wordt de sorteervolgorde bepaald a.d.v. van de bytes waardoor de verschillende karakters van de string voorgesteld worden).
`NLS_LOWER (str [, 'nlsparams'] )`	Identiek aan LOWER. Echter er kan een waarde gespecificeerd worden voor de NLS-parameter NLS_SORT.

`NLS_UPPER(str [, 'nlsparams'] )`	Identiek aan UPPER. Echter er kan een waarde gespecificeerd worden voor de NLS-parameter NLS_SORT.
`NLSSORT (srt [, 'nlsparams'])`	Retourneert de string in bytes die gebruikt wordt om de string te sorteren. Er kan een waarde gespecificeerd worden voor de NLS-parameter NLS_SORT.
`REPLACE (s1,s2 [,s3 | '' ])`	Vervangt in de string s1 elk voorkomen van patroon s2 door patroon s3. default voor s3 is een empty string.
`RPAD (s1,n [,s2 | ' ' ] )`	Als LPAD, maar nu rechts
`RTRIM (s1 [,s2 | ' ' ])`	Als LTRIM maar nu de karakters vanaf het einde
`SOUNDEX (string)`	Een fonetische code voor de (Amerikaanse) klank van de eerste lettergreep in de string
`SUBSTR (string,m [,n] )`	(***SUBSTR**ing*) gedeelte van string, beginnend op positie m en n karakters lang, of tot het einde van de string als n wordt weggelaten
`SUBSTRB (string,m [,n] )`	Identiek aan SUBSTR. Echter m en n worden in bytes weergegeven.
`TRANSLATE (s1,s2,s3)`	Vervangt in de string s1 elk karakter dat voorkomt in string s2 door het positioneel overeenkomstige karakter in string s3
`UPPER (string)`	Uppercase: string in hoofdletters

Functies met een datum

`ADD_MONTHS (d,n)`	Datum d plus n maanden, n mag ook negatief zijn
`LAST_DAY (d)`	Datum van de laatste dag van de maand die datum d bevat
`MONTHS_BETWEEN (d1,d2)`	Aantal maanden tussen d1 en d2. De jongste datum moet als eerste vermeld worden
`NEW_TIME (d,z1,z2)`	Datum en tijd d in tijdzone z1 omzetten naar de datum en tijd in tijdzone z2. Tijdzones: noteren binnen enkele quotes.

AST of ADT	Atlantic Standard Time resp. Daylight Time
BST of BDT	Bering Standard Time resp. Daylight Time
CST of CDT	Central Standard Time resp. Daylight Time
EST of EDT	Eastern Standard Time resp. Daylight Time
GMT	Greenwich Mean Time
HST of HDT	Alaska-Hawaii Standard Time resp. Daylight Time
MST of MDT	Mountain Standard Time resp. Daylight Time
NST	Newfoundland Standard Time
PST of PDT	Pacific Standard Time resp. Daylight Time
YST of YDT	Yukon Standard Time resp. Daylight Time

N.B. Daylight Time betekent zomertijd

`NEXT_DAY (d,dag)`	Geeft de datum van de eerstvolgende dag van de week aangegeven met dag (bijv. 'SUNDAY') na datum d
`ROUND (d)`	Datum d met de tijd afgerond naar de dichtsbijzijnde dag
`SYSDATE`	De systeemdatum (en -tijd)
`TRUNC (d)`	(***TRUNC**ate*) datum d met de tijd afgebroken op 's ochtends 0:00:00 uur

Veelzijdige functies

`DECODE ( uitdr, waarde1, code1 [,waarde2, code2] ..., default)`
Indien de waarde van uitdr gelijk is aan een van de opgegeven waarden dan wordt de corresponderende code gegeven, anders de default.

`DUMP (uitdr [,stelsel | 10 [,startpositie | 1 [,aantal bytes ]]])`
Geeft een dump van de interne representatie van de expressie in een op te geven talstelsel (8 = octaal, 10 = decimaal, 16 =hexadecimaal, 17 of meer = karakters). Aantal bytes is default tot het einde van de expressie.

`GREATEST (uitdr ,uitdr, ... )`
Geeft de grootste van een lijst van waarden

`LEAST (uitdr , uitdr,... )`
Als GREATEST maar nu de kleinste

`NVL ( uitdr1,uitdr2)`
(***N**ull **V**a**L**ue substitution*) indien uitdr1 IS NULL wordt uitdr2 gegeven.
indien uitdr1 IS NOT NULL wordt uitdr1 gegeven.

`UID`
(***U**ser **ID**entification*) geeft een integer die de huidige gebruiker uniek identificeert.

`USER`
Geeft de huidige gebruiker in het datatype VARCHAR2

`USERENV(argument)`
(***USER ENV**ironment*). Geeft informatie in het datatype VARCHAR2 omtrent de huidige sessie. Het argument kan zijn:

`'ENTRYID'`	geeft beschikbare auditing invoer identificatie.
`'LANGUAGE'`	geeft de taal en het land dat in de huidige sessie.
`'SESSIONID'`	geeft de auditing sessie identificatie.
`'TERMINAL'`	geeft de operating system identificatie.

`VSIZE`
(***V**alue **SIZE***) geeft de fysieke lengte van uitdr.

Conversie functies

`CHARTOROWID (string)`
Een string omzetten naar een rowid

`CONVERT (char, best_char [,bron_char])`
Converteert een karakterstring van de ene karakterset naar de andere.

`char`	te converteren string.
`best_char`	de karakterset waarnaar char geconverteerd wordt.
`bron_char`	de karakterset waarin char is opgeslagen in de database.

`HEXTORAW (string)`
Een hexadecimale string omzetten naar een binaire waarde

`RAWTOHEX (string)`
Een binaire string omzetten naar een hexadecimale waarde

`ROWIDTOCHAR (string)`
Een rowid omzetten naar een string van 18 karakters lang

`TO_CHAR (w [,weergavemasker[, nlsparams]] )`
Een numerieke waarde of datum omzetten in een string t.b.v. de opmaak. Accepteert ook de optionele parameters NLS_DATE_LANGUAGE (de taal voor dag en maand namen), NLS_NUMERIC_CHARACTERS (o.a. decimaal karakter), NLS_CURRENCY (lokaal munteenheidssymbool) en NLS_ISO_CURRENCY (ISO munteenheidssymbool) voor National Language Support.

`TO_DATE (string [,inleesmasker[nlsparams]] )`
Een string voorstellende een datum volgens het opgegeven masker omzetten naar een datumveld. Accepteert ook de optionele parameter NLS_DATE_LANGUAGE voor National Language Support.

`TO_MULTI_BYTE (char)`
Geeft van alle single-byte karakters het corresponderende multi-byte karakter

`TO_NUMBER (string[nlsparams])`
Een string van cijfers omzetten naar een numerieke waarde. Accepteert ook de parameters NLS_NUMERIC_CHARACTERS, NLS_CURRENCY en NLS_ISO_CURRENCY voor National Language Support.

`TO_SINGLE_BYTE (char)`
Geeft van alle multi-byte karakters het corresponderende single-byte karakter.

Weergavemaskers voor getallen

9999	Aantal negens geeft de lengte aan
9990	Idem, maar drukt ook losse cijfers 0 af
0999	Voorloopnullen afdrukken
$999	Een dollarteken als voorvoegsel
B999	Afdrukken van spaties i.p.v. voorloopnullen
99MI	Afdrukken van '-' achter een negatieve waarde
99PR	Negatieve waarde tussen < > afdrukken
99,99	Een komma in deze positie afdrukken
99V99	Vermenigvuldigen met 10 * het aantal negens na de V
99D99	Zet het decimale karakter in deze positie
9G999	Zet de scheiding tussen honderdtallen, duizendtallen etc. in deze positie
L999	Zet het lokale munteenheidsymbool in deze positie
C999	Zet het ISO munteenheidsymbool in deze positie
RN	Retourneert Romeinse cijfers in hoofd- of kleine letters. De waarde kan een integer zijn tussen 1 en 3999.
99.99	De decimale punt op deze positie uitlijnen
9.99EEEE	Wetenschappelijke notatie
DATE	De Juliaanse dag in het formaat MM/DD/YY

Maskers voor datumvelden -numeriek

CC of SCC	Waarde van de eeuw. S (=sign) zorgt voor een minteken bij een datum vóór Christus
SYYYY of YYYY	Volledig jaartal met of zonder teken
Y,YYY	Idem met een komma op deze positie
YYY	Laatste 3 posities van het jaar
YY	Laatste 2 posities van het jaar
Y	Laatste positie van het jaar
IYYY	Het jaar in 4 cijfers, gebaseerd op de ISO-standaard
IYY	Laatste 3 cijfers van een jaar, gebaseerd op de ISO-standaard
IY	Laatste 2 cijfers van een jaar, gebaseerd op de ISO-standaard
I	Laatste cijfer van een jaar, gebaseerd op de ISO-standaard
RR	Laatste 2 cijfers van een jaar. SQL statements zullen zowel voor als na de eeuwwisseling dezelfde waarden geven.
Q	Kwartaal
WW	Week van het jaar
IW	Week van het jaar, gebaseerd op de ISO-standaard
W	Week van de maand
MM	Maand
RM	Maand uitgedrukt in Romeinse cijfers, waarbij JAN = I etc.
DDD	Dag van het jaar
DD	Dag van de maand
D	Dag van de week
J	Julian day. het aantal dagen sinds 31/12 -4712
HH of HH12	Uur van de dag (1-12)
HH24	Uur van de dag (0-23)
MI	Minuten
SS	Seconden
SSSSS	Seconden na middernacht

Maskers voor datumvelden -strings

SYEAR of YEAR	Jaartal (b.v. NINETEEN-EIGHTY-NINE)
MONTH	Naam van de maand
MON	Eerste drie letters van de maand
DAY	Naam van de dag
DY	Eerste drie letters van de dag
AM of PM	Indicator (AM - vóór 12 uur, PM - ná 12 uur)
BC of AD	Indicator van jaartal (vóór Chr of na Chr)

Toevoegingen aan datumvelden

THST,ND,RD,TH	Rangtelwoord als achtervoegsel bij nummer (DDTH voor 4TH)
SP	Spelling van het nummer (DDSP voor four)
SPTH of THSP	Spelling van het rangtelwoord (DDSPTH voor fourth)
. , /	Tekens bij punctuatie (punt of komma of slash)
"..."	Willekeurige string
fm	"FILL MODE": voor- of achtervoegsel voor een masker teneinde het aanvullen met spaties te onderdrukken

De maskers zijn CASE SENSITIVE.
Bijvoorbeeld: MONTH => AUGUST, Month => August, month => august.

C. Operatoren, functies en maskers

Oracle ondersteunt het gebruik van verschillende talen. Deze ondersteuning wordt National Language Support (NLS) genoemd. Oracle kan single-byte en multi-byte karakters verwerken en converteren van de ene naar de andere. Single-byte wil zeggen dat ieder karakter voorgesteld kan worden door één byte. De meeste talen worden ondersteund door de single-byte karakterset. Alleen de Aziatische talen, zoals Japans en Chinees, worden ondersteund door een multi-byte karakterset, aangezien deze talen duizenden karakters hebben.
Met NLS passen datum- en nummermaskers zich automatisch aan aan de maskers zoals die in het gekozen land gebruikt worden. NLS stelt gebruikers derhalve in staat met Oracle te communiceren in de eigen taal.
De belangrijkste parameter is NLS_LANGUAGE, door deze parameter wordt onder andere bepaald in welke taal de verschillende Oracle meldingen gegeven worden. Andere NLS parameters zijn (bij het ALTER SESSION statements kunt u de uitleg bij deze parameters vinden):

```
NLS_TERRITORY
NLS_DATE_FORMAT
NLS_DATE_LANGUAGE
NLS_NUMERIC_CHARACTERS
NLS_ISO_CURRENCY
NLS_CURRENCY
NLS_SORT
```

Voorbeeld:

```
select to_char(sum(sal),'L99G999D99','nls_numeric_characters = ,.')
from werknemers;
```

C. Operatoren, functies en maskers

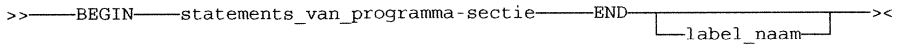
D. Statements

BEGIN — Procedureel

doel

Het afbakenen van de programma-sectie. De programma-sectie is de enige verplichte sectie in een PL/SQL-blok.

syntax



termen

label_naam	Wanneer het PL/SQL-blok bij aanvang is gelabeld dan moet u deze naam herhalen na het END-keyword.

CASE-statement (Oracle9i) — Procedureel

doel

Met behulp van het CASE statement kunt u regelen dat een aantal acties onder een bepaalde voorwaarde worden opgestart. Het CASE statement kan vertakt worden met één of meerdere WHEN clauses en eventueel een ELSE clause.
Mogelijk zijn een simple en een searched CASE statement.

syntax

simple:

```
CASE selector
  WHEN expressie1 THEN statement;
  WHEN expressie2 THEN statement;
  ...
  WHEN expressieN THEN statement;
  [ELSE statement;]
END CASE;
```

voorbeeld:

```
case v_functie
  when 'Klerk' then v_verhoging := 1.02;
  when 'Verkoper' then v_verhoging := 1.06;
  else v_verhoging := 1.08;
end case;
```

searched:

```
CASE
  WHEN conditie1 THEN statement;
  WHEN conditie2 THEN statement;
  ...
  WHEN conditieN THEN statement;
  [ELSE statement;]
END CASE;
```

voorbeeld:

```
case
  when to_char(sysdate,'hh24') < 6 then
    v_begroeting := 'Het is nacht';
  when to_char(sysdate,'hh24') < 12 then
    v_begroeting := 'Goedemorgen';
  when to_char(sysdate,'hh24') < 18 then
    v_begroeting := 'Goedemiddag';
  else
    v_begroeting := 'Goedenavond';
end case;
```

termen

selector	Een waarde of uitdrukking die vergeleken wordt met de expressie.
expressie	De expressie waarmee de selector wordt vergeleken om te bepalen of het statement na de bijbehorende WHEN moet worden uitgevoerd.
conditie	De conditie die bepaalt of het statement na de bijbehorende WHEN wordt uitgevoerd.
statement	Het uit te voeren PL/SQL wanneer de expressie of de conditie voldoet.

D. Statements

CASE-expressie (Oracle9i) **Procedureel**

doel

Bepalen van een waarde zonder gebruik te hoeven maken van een IF-TEN-ELSE constructie of een CASE-statement.
Mogelijk zijn een simple en een searched CASE expressie.

syntax

simple:

```
CASE selector
  WHEN expressie1 THEN resultaat
  WHEN expressie2 THEN resultaat
  ...
  WHEN expressieN THEN resultaat
  [ELSE resultaat]
END;
```

voorbeeld:

```
v_verhoging := case v_functie
                 when 'Klerk' then 1.02
                 when 'Verkoper' then 1.06
                 else 1.08
               end;
```

searched:

```
CASE
  WHEN conditie1 THEN resultaat
  WHEN conditie2 THEN resultaat
  ...
  WHEN conditieN THEN resultaat
  [ELSE resultaat]
END;
```

voorbeeld:

```
v_indicatie := case
                 when v_prijs < 100 then 'laag'
                 when v_prijs < 500 then 'gemiddeld
                 when v_prijs < 1000 then 'hoog'
                 else 'zeer hoog'
               end;
```

termen

selector	Een waarde of uitdrukking die vergeleken wordt met de expressie.
expressie	De expressie waarmee de selector wordt vergeleken om te bepalen of het statement na de bijbehorende WHEN moet worden uitgevoerd.
conditie	De conditie die bepaalt of het statement na de bijbehorende WHEN wordt uitgevoerd.
resultaat	De waarde die de CASE expressie oplevert.

CLOSE **Embedded SQL**

doel

Een openstaande cursor sluiten. Zie het DECLARE CURSOR statement voor een verklaring van het begrip cursor.

syntax

```
>>-----CLOSE-----cursor_naam-----><
```

termen

cursor_naam	Een al gedeclareerde én geopende cursor. Anders zal er de Oracle-fout ORA-01001 optreden, ofwel de Internal Exception INVALID_CURSOR.

D. Statements

Commentaar **Procedureel**

doel
U kunt commentaar opnemen ter verduidelijking van het programma.
Dit kan op twee manieren gebeuren: met een regel tegelijk of uitgesmeerd over meerdere regels.

syntax 1
```
-- commentaar_tekst
```

Dit formaat commentaar moet doorlopen tot het einde van de regel;
er mogen geen keywords of wat dan ook erop volgen. Het moet dus op een schone regel komen of ná de tekst.

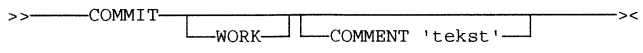
syntax 2
```
/* commentaar_tekst */
```

Dit formaat geeft meer vrijheden: u mag het overal plaatsen waar maar een spatie kan staan. Ook mag de tekst over meerdere regels verspreid liggen. De commentaartekst wordt aan de linkerkant begonnen met een /* en aan de rechterkant afgesloten met */. Alles wat tussen deze delimiters ligt wordt als commentaar opgevat.

COMMIT **SQL**

doel
Maakt een uitstaande transactie van de gebruiker definitief. Onder een transactie wordt verstaan alle wijzigingen op de gegevens in de database die de gebruiker heeft teweeggebracht met een of meer SQL-statements van het type DML (data manipulation language), zoals INSERT, UPDATE en DELETE. Oracle werkt niet met statements tegelijk maar met transacties.
Het COMMIT statement geeft aan Oracle het sein om de transactie te voltooien. Dit sein noemt men een COMMIT. Vanaf dat moment zijn de wijzigingen niet alleen voor de gebruiker zelf maar voor iedereen zichtbaar.
Uitstaande locks en savepoints m.b.t. deze transactie worden opgeheven. Zolang de commit nog niet gedaan is kan de hele transactie worden teruggedraaid.

syntax



termen

WORK	heeft geen effect en is er voor ANSI compatibiliteit.
COMMENT	eventueel commentaar bij de commit.

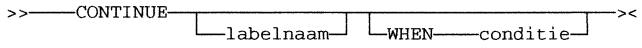
opmerking
Wees voorzichtig met het geven van een commit terwijl er nog een cursor openstaat met een FOR UPDATE clause. Elke volgende FETCH zal dan fout gaan; u moet zulke cursors eerst sluiten.

D. Statements

CONTINUE **Procedureel**

doel
Om een iteratie voortijdig te kunnen stoppen staan u een viertal statements ter beschikking: EXIT, GOTO, CONTINUE en RAISE. Het CONTINUE statement noemt het label van de programmalus.

syntax



opmerking
Het CONTINUE statement mag uitsluitend binnen lussen worden gebruikt. De conditie mag refereren aan de evt. gebruikte teller in de lus. Gelijknamige tellers kunnen worden onderscheiden door hun label.
U kunt de conditie ook opnemen in een IF statement.

DDL en DML statements **SQL Statements**

DDL is een afkorting voor Data Definition Language. De meeste DDL-statements hebben rechtstreeks effect op de data dictionary. Het betreft dan het invoeren, veranderen of verwijderen van definities van database objecten zoals tabellen, views, grants etc. We kunnen ze niet opnemen in PL/SQL, behalve wanneer gebruik wordt gemaakt van de mogelijkheden van Dynamic SQL met behulp van de package DBMS_SQL.
De volgende statements vallen binnen deze categorie:
ALTER, CREATE, DROP, RENAME,
AUDIT, NOAUDIT,
COMMENT,
GRANT, REVOKE,
SET TRANSACTION,
VALIDATE INDEX.

DML is een afkorting van Data Manipulation Language. Het gaat hierbij om transacties: het raadplegen of muteren van gegevens, tabelvergrendeling toepassen en transacties afsluiten.
De volgende statements vallen binnen deze categorie:
DELETE, INSERT, SELECT, UPDATE,
COMMIT, ROLLBACK,
LOCK TABLE,
SAVEPOINT.

De DML-statements INSERT, UPDATE en DELETE hebben in PL/SQL exact dezelfde syntax als in het standaard SQL. De enige uitzondering vormt de mogelijkheid van een WHERE CURRENT OF clause. Al deze statements maken automatisch gebruik van een impliciete cursor. Die hoeft u dus zelf niet te definiëren. De cursor is nodig omdat DML-statements evenals queries rijen moeten opvragen.
Queries onderscheiden zich doordat ze de rijen na afloop presenteren.

Met de impliciete cursor kunnen we het proces volgen. Het gaat om het huidige DML-statement. De naam van de cursor is "SQL". De vier attributen heten :

SQL%NOTFOUND
SQL%FOUND
SQL%ROWCOUNT
SQL%ISOPEN

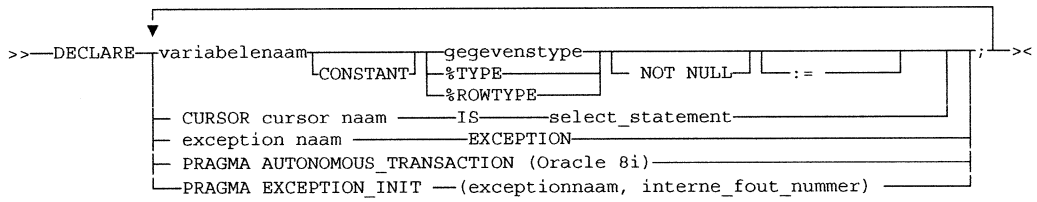
Het SQL%ISOPEN attribuut is eigenlijk overbodig, omdat impliciete cursors automatisch worden gesloten na uitvoering van het SQL-statement.

Het SELECT INTO statement krijgt ook een impliciete cursor toegewezen, en de naam is eveneens SQL.

DECLARE **Procedureel**

doel
In de declaratie-sectie maakt u bijvoorbeeld hulpvariabelen bekend: naam en gegevenstype. Zonodig kunt u meteen een waarde toekennen. De declaratie beperkt zich niet alleen tot variabelen en constanten. U kunt er ook EXCEPTIONS en CURSORS bekendmaken.

syntax



Voor de namen van objecten gelden dezelfde regels als in SQL. Dubbele namen mogen niet voorkomen.

termen

CONSTANT	Aangeven dat de variabele een constante is: de eenmaal toegekende waarde kan niet meer veranderd worden. Het toekenningsstatement is verplicht.
:=	Een waardetoekenning. De gegevenstypen van het resultaat van de expressie en de variabele moeten gelijk zijn. Gedeclareerde variabelen zonder assignment krijgen initieel de waarde NULL.
DEFAULT	In plaats van de toekenningsoperator kent PL/SQL nu ook dit gereserveerde woord om constanten, variabelen, parameters of velden een waarde toe te kennen.
EXCEPTION_INIT	Zie EXCEPTION_INIT
NOT NULL	Aangeven dat de variabele niet leeg mag worden.
AUTONOMOUS_TRANSACTION	(Oracle 8i). In de declaratiesectie kan met PRAGMA AUTONOMOUS_TRANSACTION worden aangegeven dat het volgende PL/SQL-blok als een apparte transactie moet worden uitgevoerd.

Voor uitleg bij de verschillende gegevenstypen verwijzen we naar de kop 'Gegevenstypen'.

D. Statements

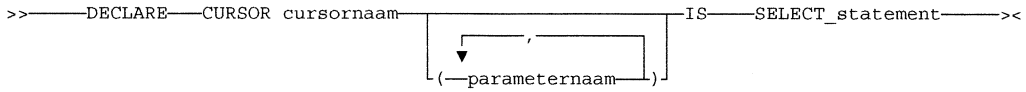
DECLARE CURSOR — Procedureel

doel

In PL/SQL moet u voor elke query waarvan het aantal rijen niet van te voren bekend is expliciet een cursor declareren. Voor DELETE, UPDATE en INSERT statements houdt PL/SQL zelf een cursor bij. Een cursor is een kleine work area van enkele kilobytes die de taak heeft om een SQL-statement te besturen. Het leest het statement in en controleert op syntax en bevoegdheden. Daarna wordt rij voor rij afgewerkt. De cursor fungeert als een wijzer die onder andere het huidige rijnummer bijhoudt.

Bij het declareren van een cursor kunt u parameters meegegeven.

syntax



termen

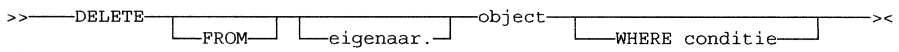
parameter_veld	Het parameterveld bestaat uit de naam plus een van de volgende gegevenstypes: VARCHAR2, NUMBER, DATE, BOOLEAN. U moet géén lengte of precisie meegeven.
select-statement	Een query zoals geformuleerd kan worden in SQL*Plus. Alle clauses behalve INTO zijn toegestaan (zie SELECT). Aparte aandacht verdient het SELECT ... FOR UPDATE statement. Het bijzondere van zulke queries is dat ze geen rijen ophalen om te tonen, maar om er expliciet locks op te plaatsen. Later kunnen ze dan m.b.v. UPDATE of DELETE statements verwerkt worden.

DELETE — SQL

doel

Het verwijderen van rijen uit een tabel of een view.

syntax



termen

object	Een tabel of view. Geef de naam van de eigenaar op wanneer het niet uw eigen object betreft; voorwaarde is wel dat u over de nodige bevoegdheid beschikt. Het verwijderen van rijen uit een view betekent in feite het verwijderen van rijen uit de onderliggende tabel.
WHERE	Alleen de rijen die aan de conditie voldoen worden verwijderd. Zonder WHERE clause worden alle rijen verwijderd.

EXCEPTION — Procedureel

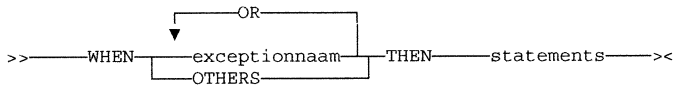
doel
Een foutafhandeling-sectie markeren. Hierbinnen kan door referentie aan bepaalde storingslabels de diverse fouten afgehandeld worden.
Een PL/SQL programma kent een aantal soorten fouten. We bedoelen hier niet de syntaxfouten en semantische fouten. We bedoelen eigenlijk geen echte fouten, maar eerder uitzonderingssituaties. In PL/SQL noemt men ze exceptions. We kennen Predefined Exceptions (of Internal Exceptions of Predefined Internal Exceptions) en User Defined Exceptions.

syntax

In de declaratie:

```
>>-----exceptionnaam-----EXCEPTION-----;-----><
```

In de exception handler:



opmerking
Er bestaan een aantal Predefined Exceptions. Ze worden ook Internal Exceptions genoemd, omdat het om Oracle fouten gaat. bijvoorbeeld:

CURSOR_ALREADY_OPEN	De cursor was reeds open.
INVALID_NUMBER	Data conversie van string naar getal ging fout

De WHEN OTHERS clausule omvat alle overige Internal Exceptions die niet Predefined zijn of niet worden genoemd in de EXCEPTION sectie. De WHEN OTHERS clause moet ofwel de enige clause zijn ofwel helemaal achteraan komen. De WHEN clause in de foutafhandeling-sectie roept een fouten-routine aan, namelijk de statements achter het THEN keyword. Men noemt ze ook wel EXCEPTION HANDLERS.

EXCEPTION_INIT — Procedureel

doel
Zelf Internal Exceptions gaan voordefiniëren.
De WHEN OTHERS clause vangt die exceptions af die niet in de lijst van Predefined Exceptions staan. Dus alle mogelijke storingen worden daarmee op één grote hoop gegooid. Misschien wilt u een interne exception die niet voorgedefinieerd is toch met name noemen in de foutafhandeling-sectie. Daartoe zult u de reeks Predefined Internal Exceptions moeten uitbreiden. U maakt dus User Defined Internal Exceptions.

syntax

```
>>-----PRAGMA EXCEPTION_INIT-----(-----exceptionnaam, interne_fout_nummer-----)-----><
```

termen

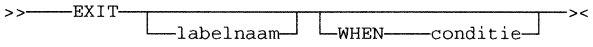
PRAGMA	Een keyword dat verplicht is en de compiler vertelt dat het hier niet handelt om een normaal statement dat gerund moet worden, maar om een informatief statement voor de compiler. Zulk soort statement wordt een PRAGMA genoemd.
exception_naam	De naam van uw zelf gedefinieerde interne exception. Deze exception moet daarvóór apart zijn gedeclareerd.
interne_fout_nummer	Een Oracle foutnummer. Oracle fouten herkent u aan het formaat ORA-99999. Bijvoorbeeld ORA-00942 met de melding: 'table or view does not exist'.

EXIT — Procedureel

doel

Om een iteratie voortijdig te kunnen stoppen staan u een viertal statements ter beschikking: EXIT, GOTO, CONTINUE en RAISE. Het EXIT statement noemt het label van de programmalus.

syntax



opmerking

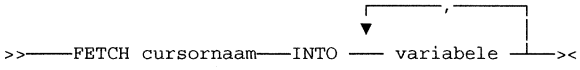
Het EXIT statement mag uitsluitend binnen lussen worden gebruikt. De conditie mag refereren aan de evt. gebruikte teller in de lus. Gelijknamige tellers kunnen worden onderscheiden door hun label.
U kunt de conditie ook opnemen in een IF statement.

FETCH — Embedded SQL

doel

Het ophalen van een afzonderlijke rij uit de database binnen een query-cursor. De cursor moet eerst gedeclareerd en geopend zijn.

syntax



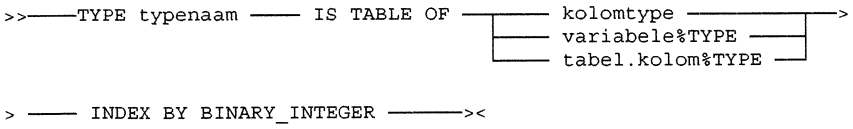
termen

variabele	Een scalair, dus geen array-variabele. De variabele moet gedeclareerd zijn. Voor elke kolom of expressie die door de query-cursor wordt geselecteerd moet er een overeenkomstige variabele zijn aangewezen. De gegevenstypen moeten overeenstemmen of converteerbaar zijn.
rij_variabele	Een zgn. STRUCTURE : een agglomeraat van velden met uiteenlopende gegevenstypen. (zie DECLARE).

Gegevenstypen

CHAR(n)	om karakterstrings met een vaste lengte op te slaan. Zonder n wordt een default lengte van 1 aangenomen. De maximale lengte is 32767 bytes (in tegenstelling tot de maximale lengte 2000 van een CHAR kolom in een tabel - vanaf Oracle8);
VARCHAR2(n)	om karakterstrings van variabele lengte op te slaan. De maximale lengte is 32767 bytes;
VARCHAR(n)	om karakterstrings van variabele lengte op te slaan. Is nu nog identiek aan VARCHAR2. In latere versies echter kan dat veranderen, zodat aanbevolen wordt VARCHAR2 te gebruiken.
NUMBER(n [, m])	voor numerieke velden, evt. gespecificeerd met grootte en decimalen. Voor de grootte wordt defaultwaarde 38 genomen, wanneer we de grootte weglaten.
DATE	voor datumvelden;
BOOLEAN	voor Booleaanse velden: kan de waarde TRUE, FALSE of NULL krijgen. Het zijn afwijkende variabelen, we kunnen ze niet vullen vanuit de database en evenmin wegschrijven naar de database;
BINARY INTEGER	om integers op te slaan, maakt niet uit van welke grootte;
LONG	om karakterstrings van variabele lengte op te slaan. Is identiek aan VARCHAR2 met het verschil dat de maximale lengte van een LONG variabele 32760 bytes is. Let op dat data in een database-kolom van type LONG een lengte van maximaal 2 Gb kan hebben en dus niet zomaar in PL/SQL kan worden ingelezen;
RAW	om binaire data of bytestrings op te slaan. Maximale lengte is 32767 bytes. Dit datatype wordt gebruikt voor plaatjes. RAW data kunnen door PL/SQL niet geïnterpreteerd worden;
LONG RAW	om binaire data of bytestrings op te slaan. Maximale lengte is 32760 bytes. LONG RAW data kunnen door PL/SQL niet geïnterpreteerd worden;
ROWID	analoog aan de pseudokolom ROWID, die elke tabel heeft, heeft PL/SQL de variabele ROWID. Deze is te manipuleren met de functies CHARTOROWID en ROWIDTOCHAR;
TABLE	objecten van het type TABLE worden PL/SQL tabellen genoemd. Ze zijn opgebouwd als database tabellen, maar ze zijn niet gelijk. De grootte van een PL/SQL tabel is onbeperkt en ze kan slechts één kolom hebben en een primaire sleutel. PL/SQL tabellen moeten gedeclareerd worden in twee stappen. Eerst de definitie van een TABLE type (1), dan de declaratie van tabellen van dat type (2).

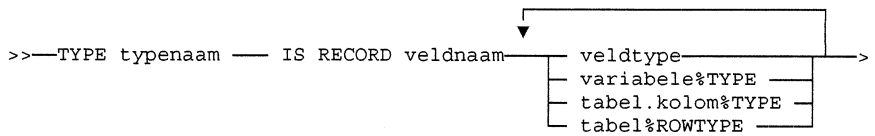
Syntax:



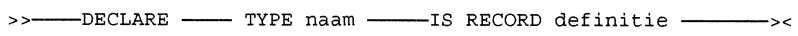
RECORD	Objecten van het type RECORD worden records genoemd. Records hebben velden met unieke namen, die van elk datatype kunnen zijn. Bovendien kunnen records gedeclareerd worden als de formele parameters van procedures en functies. PL/SQL records moeten in twee stappen gedeclareerd worden. Eerst de definitie van een RECORD type (1), dan de declaratie van records van dat type (2).

Syntax:

(1):



(2):



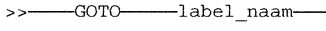
%TYPE	neemt het gegevenstype over van een eerder gedeclareerd veld of een kolom uit de database. Plaats de veldnaam c.q. kolomnaam als voorvoegsel.
%ROWTYPE	voor structuren: neemt de gegevenstypen over van de kolommen uit de database die samen een record vormen. Plaats de tabel- of viewnaam als voorvoegsel. Een tevoren gedeclareerde cursornaam mag ook, dit zien we later nog terug. Deze variabelen mogen niet beschouwd worden als één veld; het zijn samengestelde velden ofwel STRUCTURES. We kunnen wel de afzonderlijke velden eruit lichten, namelijk door de veldnaam als extensie mee te geven.
CLOB	Een groot object dat alleen karakters (CHAR) bevat. In Oracle 9i kan een CLOB maximaal 4 gigabyte worden. Vanaf Oracle 10g kan dit maximaal 128 terabyte worden.
BLOB	Een groot binair object zoals een afbeelding of een Word-document. Voor grootte zie CLOB.
BFILE	Bevat een verwijzing naar een BLOB. Deze BLOB is dan niet in de database opgeslagen, maar door de verwijzing is het wel mogelijk uit de file te lezen en ernaar te schrijven.
BINARY_FLOAT	Nieuw datatype in Oracle10g. Gebroken getallen worden 32-bits opgeslagen met binaire precisie (de cijfers 0 en 1). Daardoor kunnen rekenkundige taken veel sneller uitgevoerd worden in vergelijking met het gebruik van NUMBER. BINARY_FLOAT ondersteunt ondermeer NaN (not a number) en oneindig.
BINARY_DOUBLE	Nieuw datatype in Oracle10g. Vergelijkbaar met BINARY_FLOAT, behalve dat dit datatype 64-bits is.

GOTO **Procedureel**

doel

Met behulp van het GOTO statement zijn onvoorwaardelijke sprongen mogelijk. U moet daarbij gebruikmaken van labels.

syntax



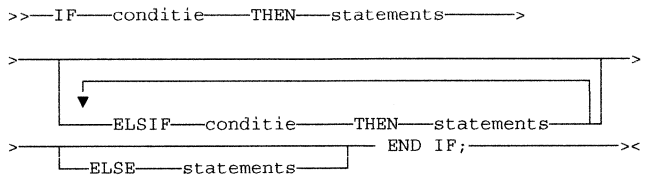
D. Statements

IF — Procedureel

doel

Met behulp van het IF statement kunt u regelen dat een aantal acties onder een bepaalde voorwaarde worden opgestart zodra de voorwaarde TRUE oplevert. Het IF statement kan vertakt worden met de clauses ELSIF en ELSE.

syntax

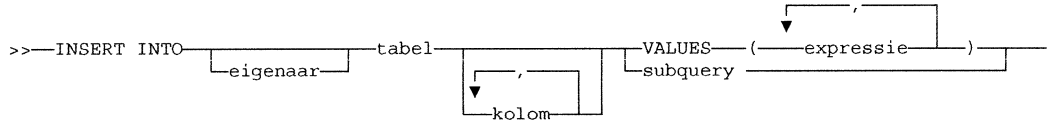


INSERT — SQL

doel

Nieuwe rijen toevoegen aan een tabel of view.

syntax



termen

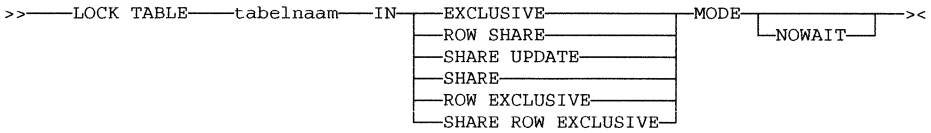
tabel	De tabel of view waaraan rijen toegevoegd worden. Wanneer het andermans object is dient u zijn naam als voorvoegsel te gebruiken; voorwaarde is dat u over voldoende bevoegdheid beschikt.
kolom	Naam van de kolom in de tabel of view. U hoeft geen kolomnamen op te geven wanneer het alle kolommen betreft, mits hun volgorde en aantal overeenstemt met de waarden in de VALUES clause resp. de kolommen in de query.
VALUES	Hier mogen alleen waarden staan voor de achter tabel genoemde kolommen (de andere kolommen worden automatisch op null gezet en mogen in dat geval dan ook niet als not null gedefinieerd zijn). Indien geen van de kolommen gespecificeerd is moet voor elk veld van de toe te voegen rij een waarde opgegeven worden (null als het veld de null waarde krijgt). De gegevenstypen moeten overeenstemmen. Plaats alfanumerieke waarden tussen enkele quotes. N.B. u kunt op deze manier maar één rij tegelijk toevoegen.
query	Om alle door de query geselecteerde kolommen en rijen tegelijkertijd aan de tabel toe te voegen.

LOCK TABLE — SQL

doel

Het expliciet aanbrengen van locks. Locks zijn voorzieningen die ervoor moeten zorgen dat gedurende gelijktijdige transacties op een database object de definities en de rijen van dat object beveiligd worden tegen inconsistentie. Oracle zorgt automatisch voor locking. Meestal wordt het LOCK TABLE statement toegepast om strengere locks te plaatsen.

syntax



termen

tabelnaam — Gebruik de naam van de eigenaar als voorvoegsel wanneer het om andermans tabel gaat. Toegestaan wanneer u over voldoende bevoegdheden beschikt.

De afzonderlijke modes worden in de cursustheorie besproken.
Onderstaand schema toont de acties die toegestaan zijn voor andere gebruikers tijdens het uitstaan van een bepaalde lock.

ACTIE → / LOCK MODE ▼	Raad-plegen	Wijzi-gen	Plaats X lock	Plaats RS lock	Plaats S lock	Plaats RX lock	Plaats SRX lock	Gebruikte statement om de lock te bewerkstelligen
Exclusive (X)	JA	--	--	--	--	--	--	LOCK TABLE IN EXCLUSIVE MODE of een DDL STATEMENT
Row Share (RS)	JA	JA	--	JA	JA	JA	JA	LOCK TABLE IN ROW SHARE MODE of SELECT FOR UPDATE OF
Share (S)	JA	--	--	JA	JA	--	--	LOCK TABLE IN SHARE MODE
Row Excl. (RX)	JA	JA	--	JA	--	JA	--	LOCK TABLE IN ROW EXCLUSIVE MODE of INSERT /UPDATE /DELETE
Share Row Excl.(SRX)	JA	--	--	JA	--	--	--	LOCK TABLE IN SHARE ROW EXCLUSIVE MODE

LOOP — Procedureel

doel

Dit statement creeert een programmalus ofwel iteratie en is bedoeld om een reeks van statements een aantal malen te herhalen.

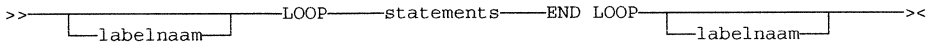
D. Statements

Mogelijkheden:

1) Ongelimiteerde lussen

Dit soort lussen kennen geen impliciete stop. De reeks van statements tussen de keywords LOOP en END LOOP worden in principe oneindig herhaald. U zult statements als EXIT, GOTO of RAISE binnen de lus moeten opnemen.

syntax

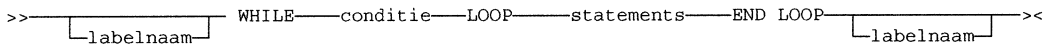


Het label is optioneel en geeft de lus een naam. Gebruik dezelfde naam wanneer u beide labels opneemt in het statement. Programmalussen kunnen genest worden. Een label kan gebruikt worden in het EXIT statement om bijvoorbeeld uit een outer loop te springen.

2) WHILE lussen

In dit soort lussen wordt vooraf getest op een conditie en wanneer het resultaat TRUE is wordt de reeks van statements uitgevoerd. Na afloop wordt de conditie opnieuw getest.

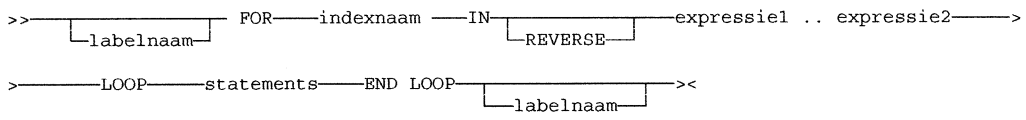
syntax



3) Numeriek geïndexeerde lussen

Deze worden gestuurd door een teller. U geeft een domein op. Het aantal iteraties is gelijk aan de eindwaarde minus de startwaarde plus 1 .

syntax



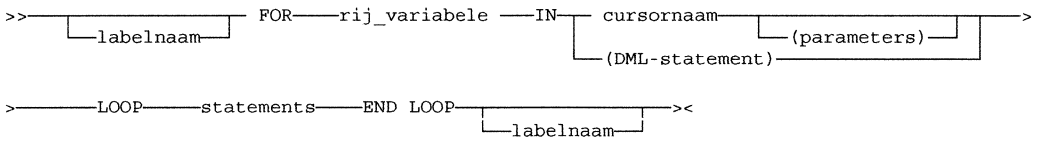
termen

index_naam	Een naam voor de teller. Het gegevenstype is automatisch numeriek.
expressie	Een formule die een geheel getal oplevert. Gewoonlijk geeft men eenvoudig het getal zelf op. De eerste expressie mag geen groter getal opleveren dat de tweede expressie. Alleen bij de eerste cyclus worden de expressies uitgerekend. Na elke cyclus wordt de teller met de waarde 1 opgehoogd. Het is niet mogelijk om een andere ophoogwaarde te kiezen.
REVERSE	Aangeven dat de teller bij aanvang de waarde van de tweede expressie krijgt en na elke cyclus met de waarde 1 wordt verminderd.

4) Cursor-gestuurde lussen

De bedoeling van deze iteratie is om voor elke rij die door de query wordt opgehaald eenzelfde reeks statements uit te voeren.

syntax



termen

rij_variabele	Een variabele waarin de afzonderlijke kolommen qua naam en gegevenstype vervat zijn. Een agglomeraat van meerdere velden dus. .
cursor_naam	De expliciete cursor die geassocieerd wordt met de query.
parameter	Een parameter kunnen we vergelijken met het meegeven van substitutie-variabelen aan een query. Gewoonlijk worden de parameters gebruikt in de WHERE of HAVING clause om de selectie toe te spitsen.
DML-statement	Hiermee is een impliciete cursor geassocieerd. In plaats van de naam van die cursor (in werkelijkheid: "SQL") noemt u het statement voluit, omsloten met ronde haken.

NULL **Procedureel**

doel
Het NULL statement is een dummy statement: het doet niets. Het heeft overigens niets te maken met de waarde NULL. Gewoonlijk wordt het NULL statement gebruikt ten behoeve van de leesbaarheid. In het programma wordt dan duidelijk dat de programmeur expliciet de actie "niets doen" bedoelt.

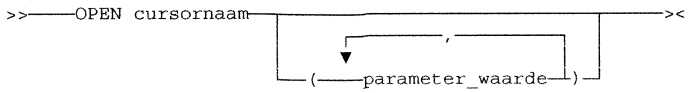
syntax

opmerking
Bij het GOTO statement wordt gesprongen naar het eerstvolgende statement onder het label. Soms is er geen statement van toepassing, maar wel nodig i.v.m. de syntax. Gebruik dan NULL.

OPEN **Embedded SQL**

doel
Het openen van een gedeclareerde cursor. Eventuele cursor-parameters krijgen een waarde toegekend. Een geopende cursor is gereed voor uitvoering van de query.

syntax



Voor de parameter_waarde vult u een constante in of een programma-variabele of een expressie. U moet net zoveel parameters meegeven als in het DECLARE CURSOR statement. Houd ook dezelfde volgorde aan, omdat de parameters positioneel zijn. Tenzij u expliciet de naam opgeeft:

```
open cursornaam(parameternaam => parameterwaarde, ...);
```

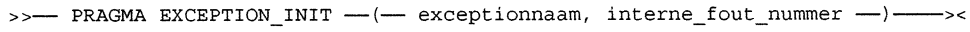
Wanneer u beide soorten referenties door elkaar gebruikt, moet u beginnen met de positionele parameters.

opmerking
In de bijbehorende statements FETCH en CLOSE worden geen parameters meegegeven.

PRAGMA EXCEPTION_INIT — Procedureel

doel
Mogelijkheid om User Defined Internal Exceptions te maken. Dit zijn storingslabels gebaseerd op Oracle fouten.

syntax



Termen:

PRAGMA	Een keyword dat verplicht is en de compiler vertelt dat het hier niet handelt om een normaal statement dat gerund moet worden, maar om een informatief statement voor de compiler. Zo'n statement wordt een PRAGMA genoemd.
exception_naam	De naam van uw zelf gedefinieerde interne exception. Deze exception moet daarvoor apart zijn gedeclareerd.
interne_fout_nummer	Een Oracle foutnummer. Oracle fouten herkennen we aan het formaat ORA-99999. We zijn ze vast al eens tegengekomen. Een voorbeeld: `ORA-00937 not a single-group group function` Gebruik alleen het nummer, maar zonder het voorvoegsel ORA. Voorloopnullen mogen we weglaten. Het nummer moet negatief zijn.

D. Statements

PRINT **SQL*Plus**

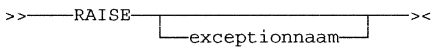
doel
Met dit commando kan de waarde van een bepaalde variabele getoond worden (zie ook VARIABLE).

syntax

RAISE **Procedureel**

doel
Raise betekent "aanheffen". Met dit statement wordt expliciet een exception aangeheven. Zodra zich een exception voordoet (en wordt aangeheven) wordt de programma-sectie afgebroken. Exceptions zijn storingslabels die in de foutafhandeling-sectie afgewerkt worden door de overeenkomstige Exception Handlers.

syntax



termen

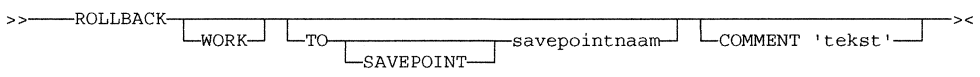
exception_naam	Een gedeclareerde exception. U ziet aan de syntax dat u één of geen exception kunt meegegeven. Het RAISE statement zonder exceptions mag alleen in een Exception Handler (in de foutafhandeling-sectie dus) en heeft tot gevolg dat de huidige exception nogmaals aangeheven wordt.

opmerking
U kunt een User Defined Exception hangen onder zowel een SQL-statement als een procedureel statement.

ROLLBACK **SQL**

doel
Het ongedaan maken (terugdraaien) van een uitstaande transactie. De database-gegevens herkrijgen hun status van voor de transactie. Tevens worden de bijbehorende locks vrijgegeven. Wanneer een transactie onderbroken wordt door stroomuitval of door een fout tijdens het uitvoeren van een statement volgt er automatisch een rollback.

syntax



termen

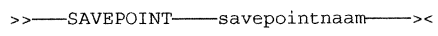
WORK	Heeft geen effect. Voor ANSI compatibiliteit.
TO SAVEPOINT	Om de transactie gedeeltelijk terug te kunnen draaien, namelijk tot aan een bepaalde vlag die binnen de transactie was gedefinieerd.

COMMENT Eventueel commentaar bij de rollback.

SAVEPOINT SQL

doel
Savepoints zijn markeringen binnen transacties en worden gebruikt om grote transacties onder te verdelen in kleinere delen. Dit stelt de programmeur in staat om een transactie gedeeltelijk terug te draaien.

syntax

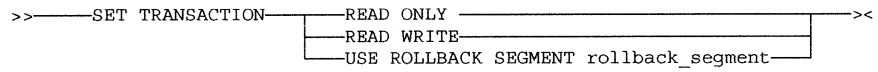


opmerking
Voor de naam gelden dezelfde afspraken als bij andere database-objecten zoals tabellen. Binnen de transactie moeten de savepoints een unieke naam hebben. Mocht u een savepoint definiëren dat al eerder bestond, dan geldt de laatste.
U refereert naar een savepoint in het ROLLBACK TO SAVEPOINT statement.

SET TRANSACTION SQL

doel
Lees-consistentie niet beperken tot afzonderlijke queries, maar te handhaven gedurende een transactie die uit meerdere queries bestaat.

syntax



opmerking
Het SET TRANSACTION statement moet het eerste statement zijn in een transactie.

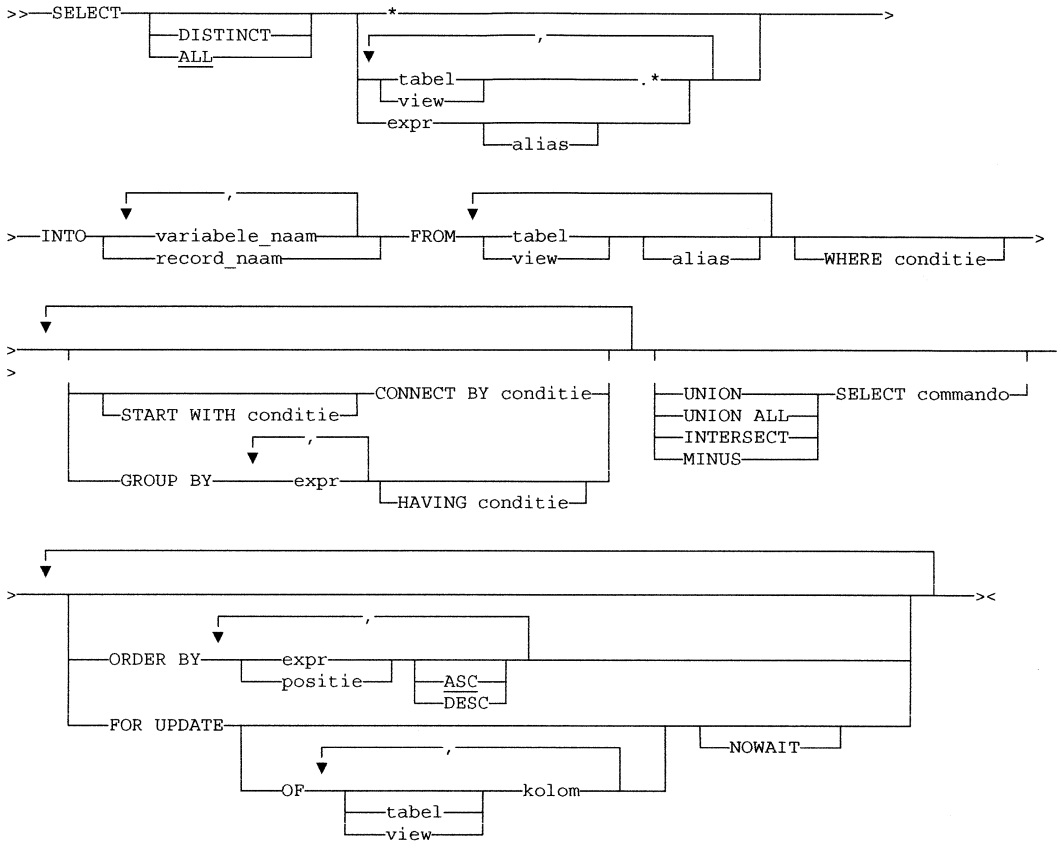
termen

READ ONLY	Maakt de huidige transactie read only.
READ WRITE	Maakt de huidige transactie read write.
USE ROLLBACK SEGMENT segment_naam	Wijst de huidige transactie toe aan het gespecificeerde rollback segment.

SELECT SQL

doel
Het raadplegen van één enkele rij uit een tabel of view.

syntax



Toelichting :
Op de INTO clause na is de syntax volkomen gelijk aan de standaard syntax.

termen

ALL / DISTINCT — Zorgt ervoor dat in beginsel alle rijen worden opgehaald. DISTINCT onderdrukt dubbele occurrences.

expr — Een expressie. Hieronder wordt verstaan: kolom; pseudo-kolom (rownum, level, user, uid, sysdate, null); constante of substitutie-variabele; functie van bovengenoemde element(en) van willekeurige complexiteit; een door de gebruiker gedefinieerde functie die opgeslagen is in de database.
Bijzonderheden:
U mag een * (asterisk) gebruiken om aan te geven dat u van een tabel of view alle kolommen wil selecteren; U mag de betreffende tabelnaam of viewnaam of een alias gebruiken als voorvoegsel voor de expressie; De expressie kunt u ook een alias geven door achter de expressie de alias mee te geven. Deze moet omgeven zijn met dubbele quotes wanneer er spaties of punctuatie in staat.

INTO

Aangeven waar de geselecteerde expressies in bewaard moeten worden. U kunt kiezen uit losse variabelen of een rij-variabele.

variabele — Een scalair, dus geen rijvariabele. De variabele moet gedeclareerd zijn. Voor elke kolom of expressie die door de query-cursor wordt geselecteerd moet er een overeenkomstige variabele zijn aangewezen. De gegevenstypen moeten overeenstemmen of converteerbaar zijn.

record — Een zgn. STRUCTURE : een agglomeraat van velden met uiteenlopende gegevenstypen. (zie DECLARE).

FROM

Specificeert de tabel, view of synoniem welke geraadpleegd moet worden. Meerdere tabellen geven een join aan. Een alias kan worden opgegeven na de objectnaam.
De FROM clause is verplicht, dus ook wanneer u geen kolommen maar andere expressies selecteert en daar in principe geen tabel voor nodig hebt. Gebruik dan een dummy tabel, bijv. DUAL.

WHERE

Geeft een conditie aan voor het filteren van rijen en/of het samenvoegen (join) van tabellen. De WHERE clause kan uit meerdere predicaten bestaan door gebruikmaking van AND en/of OR. Het linkerlid van een predicaat moet een expressie zijn of [NOT] EXISTS. Het rechterlid moet een expressie of een subquery zijn. De subquery moet omsloten worden door kromme haakjes.

CONNECT BY

Zorgt ervoor dat de rijen in een boomstructuur worden geselecteerd en geeft de relatie tussen de rijen aan. De voorwaarde kan zijn:
PRIOR uitdr <operator> uitdr ;
uitdr <operator> PRIOR uitdr.
PRIOR geeft de parent aan in de parent-child verbinding. Met de CONNECT BY clause is het mogelijk om tevens de pseudo-kolom LEVEL te selecteren. Deze toont het niveau binnen de boomstructuur. CONNECT BY mag niet voorkomen in subqueries of joins.

START WITH

Alleen in combinatie met CONNECT BY. Identificeert de rijen, die als wortels van de boom worden gebruikt, door middel van een uitdrukking waaraan deze rijen moeten voldoen.

GROUP BY

Om rijen die dezelfde waarden in één of meer velden hebben te groeperen tot enkelvoudige rijen. Er vindt hierbij impliciete sortering plaats.

HAVING

Filtert de rijen die na de GROUP BY overblijven.
Mag alleen in combinatie met GROUP BY. Voor de formulering van de voorwaarde gelden dezelfde regels als bij WHERE.

SET operator

Verzamelt de rijen van twee queries en sorteert ze, waarna er een operatie op losgelaten wordt. Op UNION ALL na verwijderen ze ook de dubbele rijen. Oracle kent vier SET operatoren:

UNION	Verenigt de rijen.
UNION ALL	Verenigt de rijen en geeft ook de dubbele rijen.
INTERSECT	Geeft de rijen die door beide queries worden opgehaald.
MINUS	Geeft de rijen van de eerste query behalve de rijen van de tweede query.

De kolommen in de queries moeten qua aantal en gegevenstype overeenstemmen. Alleen de laatste query mag een ORDER BY bevatten.

ORDER BY

Sorteert de rijen expliciet op de waarde in de expressie, oplopend (ASC) of aflopend (DESC). In plaats van de expressie kunt u ook het rangnummer van die expressie in de SELECT regel noemen.

FOR UPDATE [OF]

Plaatst een lock op de geselecteerde rijen; deze kunnen pas weer door andere gebruikers worden benaderd na een commit of rollback. De wijzigingen zelf kunnen vervolgens met INSERT, UPDATE of DELETE statements gedaan worden. In geval van een query op meerdere tabellen kan met het optionele OF tabelnaam.kolomnaam gespecificeerd worden welke tabel(len) gelocked moeten worden. Alleen FOR UPDATE locked alle tabellen uit de FROM clause.
In dit SELECT statement mag niet gebruikt worden: Distinct, Group By, set operatoren (Union , Minus , Intersect), groepfuncties (Count, Max, etc.). Wat in de SELECT regel staat is niet zo belangrijk: het gaat om de rijen en niet om de kolommen. Dat geldt ook voor de FOR UPDATE clause, ofschoon men daarin afzonderlijke kolommen kan specificeren. Oracle lockt minimaal rijen en geen kolommen. Op de tabel wordt een Row Share lock geplaatst en op de rijen een (zoals altijd) Exclusive lock.

NOWAIT

Breekt de query af wanneer rijen die FOR UPDATE worden geselecteerd niet benaderd kunnen worden, bijvoorbeeld wanneer iemand anders al een lock op deze rijen heeft uitstaan.

Opmerking

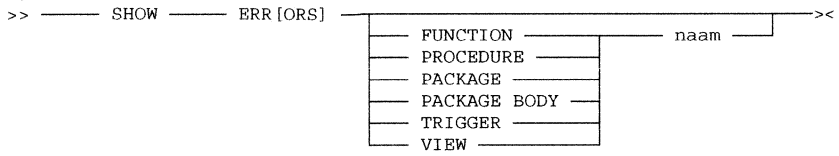
De volgorde van de clauses van het volledige SELECT-statement is als volgt

1	SELECT
2	INTO
3	FROM
4	WHERE
5/7	CONNECT BY
6/8	START WITH
5/7	GROUP BY
6/8	HAVING
9	SET operator
10/11	ORDER BY
10/11	FOR UPDATE (OF)

SHOW ERRORS — SQL*Plus

doel
Met het commando SHOW ERR[ORS] kan een specificatie van de gemaakte compilatiefouten bekeken worden.

syntax

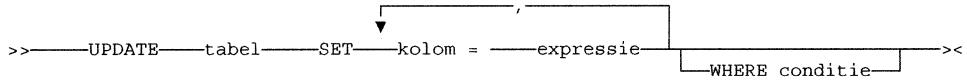


opmerking
Indien er geen argumenten worden meegegeven aan het commando dan toont SHOW ERR[ORS] het meest recent gecompileerde object.

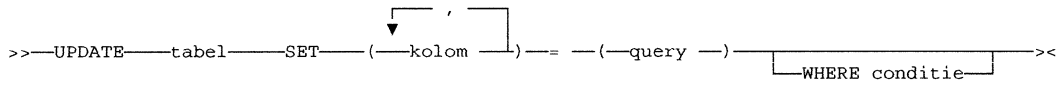
UPDATE — SQL

doel
Het veranderen van kolomwaarden in een tabel of view.

syntax 1



syntax 2



termen

tabel	Een tabel, synoniem of view. Wanneer die niet van uzelf is moet u de naam van de eigenaar als voorvoegsel meegeven. U moet dan wel over voldoende rechten beschikken.
alias	U kunt een alias meegeven waaraan u kunt refereren in andere clauses van het statement.
SET	Geeft aan welke kolommen binnen een rij gewijzigd worden en welke waarden deze kolommen krijgen.
WHERE	Alleen de rijen die aan de conditie voldoen worden gewijzigd. Zonder WHERE clause worden alle rijen gewijzigd. Een bijzondere conditie is : WHERE CURRENT OF cursor_naam. Hierbij wordt gerefereerd aan de huidige rij die door een FETCH statement is opgehaald. De cursor betreft een query van het formaat SELECT ... FOR UPDATE.

query — Geeft de velden in de genoemde kolommen de overeenkomstige waarden welke de query ophaalt. De query mag geen ORDER BY of FOR UPDATE clause bevatten en moet precies één rij ophalen voor elke rij die gewijzigd moet worden en evenveel expressies selecteren als de links van het = teken aantal genoemde kolommen. Indien er geen rij wordt opgehaald worden NULL waarden ingevuld.

VARIABLE **SQL*Plus**

doel
Met dit commando kunnen variabelen gedeclareerd worden. (Zie ook het commando PRINT).

syntax

```
>>———VARIABLE naam type————>
```

WERKBOEK

Opdrachten en uitwerkingen

Oracle Database:
PL/SQL voor ervaren programmeurs

OPDRACHTEN

Oracle Database:
PL/SQL voor ervaren programmeurs

Opdrachten hoofdstuk 3

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 PL31.SQL, bij opdracht 2 van hoofdstuk 3 hoort het script met de naam PL32.SQL, enz.).

1. Maak een PL/SQL-blok dat de tabel HULPTABEL vult met alle even getallen tussen 0 en 50, behalve de getallen van 20 tot en met 30. Gebruik in dit programma een FOR-loop met index.

   Hint: gebruik de SQL-functie MOD om te bepalen of een getal even is.

2. Maak een PL/SQL-blok met een FOR loop dat de som van alle getallen van 1 tot en met een opgegeven getal bij elkaar optelt. De verkregen som moet in de HULPTABEL komen te staan. Wanneer het opgegeven getal bijvoorbeeld de waarde 5 heeft, dan moet de som gelijk worden aan 15 (1+2+3+4+5).

   Vóór de optelling begint, moet worden getest of het opgegeven getal wel groter of gelijk is aan 1. Anders moet in de tabel HULPTABEL de foutmelding 'Getal < 1' komen te staan.

   Het op te geven getal dient ingelezen te worden middels het ACCEPT commando, dat aan het PL/SQL-blok voorafgaat (zie ook voorbeelden in de cursusmap).

   Voorbeeld:

```
accept a_nummer number prompt 'Geef een getal : '

declare
  v_getal   number;

begin
  v_getal:= &a_nummer;
```

3. Maak een PL/SQL-blok met een FOR loop dat de nummers van bankbiljetten controleert op hun geldigheid. Bij een geldig nummer moet de som der afzonderlijke cijfers gedeeld door 9 een geheel getal opleveren. De resultaten kunt u in HULPTABEL zetten.

   bankbiljetten bevatten altijd nummers van 10 cijfers lang; pas zonodig de breedte van numerieke kolommen aan met behulp van het SQL*Plus commando **set numwidth** 10.

   Hint: gebruik de SQL-functie SUBSTR om de afzonderlijke cijfers uit het bankbiljetnummer te halen.

Op de volgende pagina staan nog een aantal opdrachten.

Opdrachten hoofdstuk 3

Optioneel:

4. Maak een programma dat vraagt om een naam. Schrijf vervolgens in de hulptabel een rij waarin in de tweede kolom de opgegeven naam en in de derde kolom een opmerking over die naam.
   Deze opmerking is afhankelijk van de naam en wordt als volgt bepaald (gebruik hiervoor CASE):
   - Is de naam langer dan 10 karakters, dan is de opmerking "Naam te lang".
   - Bevat de naam een punt, dan is de opmerking "Bevat een punt".
   - Is de eerste letter geen hoofdletter, dan "Begin geen hoofdletter".
   In alle andere gevallen is de opmerking "Geen bijzonderheden".

5. Maak een programma waarmee een ingevoerde naam omgedraaid in de hulptabel terecht komt. Bijvoorbeeld: Marcel wordt Lecram (het eerste karakter wordt hierbij dus een hoofdletter, de rest kleine letters).

   De op te geven naam dient ingelezen te worden middels het ACCEPT commando, dat aan het PL/SQL-blok voorafgaat (zie ook voorbeelden uit het boek).

   Voorbeeld:

```
accept a_naam char prompt 'Geef een naam a.u.b. :'

declare
  v_naam          varchar2(100);

begin
  v_naam := '&a_naam';
  ...
```

   Hint: maak voor deze opdracht onder andere gebruik van de SQL-functie SUBSTR.

6. Maak een PL/SQL-blok dat van een zelf op te geven getal n n! (n-faculteit) berekent. (Faculteit wil zeggen: n * (n-1) * (n-2) *...* 2 * 1, bijvoorbeeld 8! is 8*7*6*5*4*3*2*1).
   Het getal dient weer ingelezen te worden middels het ACCEPT commando, dat aan het PL/SQL-blok voorafgaat.

7. Maak een programma dat een string met namen inleest. De namen worden binnen de string gescheiden door komma's. Elke naam wordt als nieuwe rij in kolom2 van de hulptabel weggeschreven. In de rijen wordt in het eerste veld (kolom1) ook steeds een volgnummer weggeschreven, dat bij een nieuw persoon met 1 wordt opgehoogd.

   Hint: maak voor deze opdracht gebruik van de SQL-functies INSTR en SUBSTR.

Opdrachten hoofdstuk 4

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 PL41.SQL, bij opdracht 2 van hoofdstuk 4 hoort het script met de naam PL42.SQL, enz.).

1. Directeur Kraay overweegt om de salarissen van zijn werknemers (tabel P_WERKNEMERS) te gaan verhogen. Hij wil een overzicht van de salarissen wanneer hij de volgende salarisverhogingen zou doorvoeren:
   * 1% salarisverhoging voor alle klerken en analisten
   * 2% salarisverhoging voor alle verkopers
   * geen salarisverhoging voor de werknemers met overige functies

   Maak een PL/SQL-blok dat de nieuwe salarissen berekent en het oude salaris, het nieuwe salaris en de werknemersnaam in de HULPTABEL zet. Gebruik voor het ophalen van de werknemersgegevens een FOR-LOOP met cursor.

2. Maak een PL/SQL-blok dat van een opgegeven kantoornummer alle namen van de werknemers met hun salaris in de tabel HULPTABEL zet.

   Vraag het kantoornummer op middels het ACCEPT-commando en gebruik voor het ophalen van de werknemersgegevens een FOR-LOOP met cursor.

   Er hoeft in dit programma nog niet gecontroleerd te worden of het opgegeven kantoornummer een bestaand nummer is. Probeer het programma met verschillende kantoornummers uit.

3. Maak een PL/SQL-blok dat van een opgegeven ziekenhuisnaam controleert of deze bestaat in de tabel P_ZIEKENHUIZEN. Als deze naam bestaat zet deze dan samen met het aantal bedden in de tabel HULPTABEL met daarbij de tekst 'Dit ziekenhuis is bekend', en anders de opgegeven naam met de tekst 'Dit ziekenhuis is niet bekend'.

   Maak gebruik van het accept-commando en probeer het programma met verschillende namen uit.

Op de volgende pagina staan nog een aantal opdrachten.

Opdrachten hoofdstuk 4

4. Maak een PL/SQL-blok dat het volgende uitvoert: uit de tabel P_WERKNEMERS wordt het gemiddelde salaris per functie berekend. Normaal zou u dat oplossen m.b.v. de query:

```
select functie, avg(sal)
from   p_werknemers
group by functie;
```

U gaat nu het gemiddelde berekenen met een PL/SQL-blok, maar dan **zonder** GROUP BY of SQL-functies (AVG, SUM, etc..).

U laat het programma met behulp van een loop steeds een functie uit de tabel ophalen. Per functie wordt het gemiddelde berekend (zonder dus gebruik te maken van de SQL-functie SUM).

Verzamel per functie de volgende gegevens in de hulptabel: het aantal mensen dat de functie uitoefent, de functienaam en het gemiddelde salaris van de functie.

Optioneel:

5. Maak een PL/SQL-blok dat van een opgegeven ziekenhuisnummer en afdelingsnaam de stafleden die daar werken ophaalt.
Controleer eerst in tabel P_AFDELINGEN of de opgegeven afdelingsnaam en het ziekenhuisnummer bestaan. Als de afdeling in het ziekenhuis niet bestaat zet dan in de HULPTABEL het opgegeven ziekenhuisnummer en de afdelingsnaam met de tekst 'Deze afdeling is niet bekend'.
Controleer, wanneer de afdeling wel in het ziekenhuis bestaat, in de tabel P_STAFLEDEN of er stafleden op de afdeling werken. Zet, als er geen stafleden werken, in de HULPTABEL het opgegeven ziekenhuisnummer en de afdelingsnaam met de tekst 'Geen stafleden aanwezig'. Zet anders het ziekenhuisnummer, de afdelingsnaam en de namen van de stafleden die daar werken in de HULPTABEL.

Probeer het programma met verschillende afdelingsnamen en ziekenhuisnummers uit.

1. **Hint :** Gebruik hiervoor twee cursoren en een while loop.

Opdrachten hoofdstuk 5

1. Maak één PL/SQL programma dat in de tabel P_WERKNEMERS de onderstaande veranderingen voor de juiste rijen doorvoert.
Geef bij de opdracht het commandoscriptje de naam PL51.SQL.

In verband met een reorganisatie worden de werknemers van de kantoren Boekhouding en Verkoop verwisseld. Daarom moet de tabel P_WERKNEMERS worden aangepast; de medewerkers van kantoor 10 wisselen met de medewerkers van kantoor 30.

Verder krijgt iedere medewerker een salarisverhoging.

- de salarissen tot en met 2000 met 8%;
- de salarissen van 2000 tot en met 2500 met 6%;
- de salarissen van 2500 tot en met 3500 met 4%;
- de salarissen boven 3500 met 2%.

Omdat het bedrijf gaat internationaliseren moeten daarnaast alle salarissen, nu in euro's, worden omgezet naar dollars (houd een koers aan van € 1,05 voor 1 dollar).

Maak gebruik van FOR UPDATE... en WHERE CURRENT OF...

LET OP! Draai na het maken van de opdrachten de gemaakte wijzigingen terug via een ROLLBACK;

Optioneel:

2. Maak één PL/SQL programma dat in de tabellen P_KANTOREN en P_WERKNEMERS de onderstaande veranderingen voor de juiste rijen doorvoert:
   - De gebruiker moet een kantoornaam kunnen opgeven.
   - Wanneer de opgegeven kantoornaam in de tabel P_KANTOREN voorkomt, moet deze kantoornaam in de tabel P_KANTOREN gewijzigd worden in kleine letters.
   - Voor de werknemers die in het opgegeven kantoor werken en als functie Manager of Directeur hebben, moet de toeslag in 500 euro worden gewijzigd.
   - Voor alle overige werknemers in het opgegeven kantoor moet de toeslag in 1000 euro worden gewijzigd.

   Geef bij de opdracht het commandoscriptje de naam PL52.SQL.

   Maak gebruik van FOR UPDATE... en WHERE CURRENT OF...

   LET OP! Draai na het maken van de opdracht de gemaakte wijzigingen terug via een ROLLBACK;

Opdrachten hoofdstuk 5

Opdrachten hoofdstuk 6

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 PL61.SQL, bij opdracht 2 van hoofdstuk 6 hoort het script met de naam PL62.SQL, enz.).

1. Maak opgave 3 van hoofdstuk 3 nogmaals, maar dan gebruik makend van minimaal drie exceptions (bv: geen 10 cijfers, geldig nummer, ongeldig nummer).

2. Maak een programma dat van een opgegeven naam van een werknemer de naam van de manager ophaalt. De werknemer en zijn manager worden opgeslagen in de HULPTABEL.

   Maak in het programma weer gebruik van het ACCEPT-commando om de naam van de werknemer in te lezen. Gebruik in ieder geval de volgende twee exceptions :
   - e_geen_werkn (als de werknemer niet in de tabel P_WERKNEMERS voorkomt),
   - e_geen_manager (als de werknemer geen manager heeft).

   Hint: het persnr van de manager is het mgr-nummer van de werknemer die onder de manager werkt.

3. Overeenkomstig het voorbeeld met de negenproef gaat U een PL/SQL programma opnemen in een SQL-commandobestand. De gebruiker wordt gevraagd om een getal op te geven tussen 1 en 1000. Het programma moet testen of het getal een priemgetal is. Een priemgetal is een getal dat alleen deelbaar is door 1 en door zichzelf, zoals bijvoorbeeld 2, 11, 17 en 31.

   Schrijf de analyse weg in een hulptabel, gebruik makend van de volgende exceptions:
   het getal is 0 ;
   - het getal is groter dan 1000 ;
   - het getal is negatief ;
   - het getal is niet geheel ;
   - het getal is geen priemgetal. Noem ook de eerste deler.

   Verder:
   - behandel het getal 1 en 2 apart; dit zijn priemgetallen.
   - De overige getallen deelt u eerst door 2 om te kijken of ze even zijn; dit zijn geen priemgetallen
   - De oneven getallen deelt u eerst door 3, vervolgens door 5, 7, 9, 11 etc... zolang de deler kleiner of gelijk is aan de wortel van het getal.

i *Wanneer u gebruik maakt van een ampersand (&) variabele, dan kunt u in SQL Developer in het Enter Value dialoogvenster een lege waarde opgeven door letterlijk NULL in te typen.*

Optioneel:

4. Maak een programma dat controleert of een ingevoerde string een postcode kan zijn. Neem de volgende exceptions op:
   - De eerste 4 karakters zijn geen cijfers.
   - De laatste twee karakters zijn geen tekst.
   - De laatste twee karakters zijn geen hoofdletters.
   - Tussen de cijfers en de twee letters bevindt zich geen spatie.

Opdrachten hoofdstuk 7 - Casus 'Banktransacties'

Maak een commandobestand met de naam PL7.SQL. Het programma dat u gaat maken, houdt van een tabel met rekeningnummers het saldo bij aan de hand van een mutatietabel. De code verwerkt de gegevens van de mutatietabel P_MUTATIE in de rekeningtabel P_REKENING.

De kolom SOORT_TR in de tabel P_MUTATIE geeft aan om wat voor transactie het gaat. De mogelijke soorten transacties in de tabel P_MUTATIE zijn:

- A opname (AF)
- B storting (BIJ)
- O overschrijving

In de tabel P_MUTATIE geven de kolommen VAN_REK en NR_REK aan waar de bedragen in tabel P_REKENING moeten worden verwerkt. De kolom VAN_REK is het rekeningnummer waarvan het saldo met het bedrag wordt *verminderd* en de kolom NR_REK is het rekeningnummer waarvan het saldo met het bedrag wordt *vermeerderd*. De kolom SOORT_TR bepaalt of er saldi worden vermeerderd en/of verminderd.

Na het verwerken van een rij uit de mutatietabel moet met een code in de mutatietabel worden aangegeven dat een transactie is uitgevoerd. Er kan daarbij ook een foutsituatie zijn ontstaan. De kolom CODE in de tabel P_MUTATIE kan daarom de volgende waarden bevatten:

- 0 nog niet uitgevoerd
- 1 uitgevoerd
- 2 saldo ontoereikend
- 3 rekeningnummer bestaat niet
- 4 overschrijving naar eigen rekening niet mogelijk
- 5 anders

Een beschrijving en de inhoud van de tabellen P_REKENING en P_MUTATIE vindt u overigens in de appendix..

Tips:
Bouw het programma in stukjes op, begin bijvoorbeeld met de transactie storting (B) en zorg ervoor dat de tabellen P_MUTATIE en P_REKENING op de juiste manier worden bijgewerkt. Het juiste saldo en de datum van P_REKENING moeten worden gewijzigd en de code uit P_MUTATIE moet worden aangepast. Als dit deel werkt, ga dan verder met het deel dat de opnamen verwerkt en maak tenslotte de code voor de overschrijvingen.

Maak geen gebruik van exceptions. Bij het aanroepen van een exception wordt na het uitvoeren van de exception het programma namelijk verlaten; het programma stopt en er worden geen volgende rijen meer uit tabel P_MUTATIE opgehaald.

Met het commando `@herstel` (het bestand herstel.sql bevindt zich in uw homedirectory) is het mogelijk om de tabellen in de oude staat terug te brengen.

Om te controleren of deze opdracht gelukt is geven we op de volgende pagina het eindresultaat.

Opdrachten hoofdstuk 7 - Casus 'Banktransacties'

Het eindresultaat van deze opdracht:

```
@herstel

1 row created.
...
...
1 row created.

Commit complete.


@pl7

Anonymous block completed


select *
from   p_rekening;

   REK_NR        SAL DATUM
--------- ---------- ---------
        1        450 13-JAN-94
        2        450 14-JAN-94
        3       1400 10-JAN-94
        4       1150 14-JAN-94
        5       1650 13-JAN-94


select *
from   p_mutatie;

   VOLGNR   VAN_REK    NR_REK    BEDRAG DATUM     S       CODE
--------- --------- --------- --------- --------- - ---------
        1         1         1       250 04-JAN-94 A         1
        2         2         2       500 05-JAN-94 A         1
        3         3         3       150 06-JAN-94 A         1
        4         4         5       250 07-JAN-94 O         1
        5         1         3       550 10-JAN-94 O         1
        6         4         4       350 11-JAN-94 B         1
        7         5         5       650 12-JAN-94 B         1
        8         5         1       250 13-JAN-94 O         1
        9         2         4        50 14-JAN-94 O         1
       10         6         1       750 17-JAN-94 O         3
       11         3         1      1750 18-JAN-94 O         2
       12         4         4        75 19-JAN-94 O         4
```

UITWERKINGEN

Oracle Database:
PL/SQL voor ervaren programmeurs

De uitwerkingen ontvangt u van de docent,
na het afronden van deze cursus.

INDEX

OPM: Pagina's inclusief hoofdstuk-aanduiding zijn te vinden in het theoriegedeelte in het begin van de map. Overige pagina's verwijzen naar de bijlagen.

%FOUND. *Zie* cursorattributen
%NOTFOUND. *Zie* cursorattributen
%ROWTYPE, 28; 32
%TYPE, 28
@ (START), 9
@ (verbinding met database), 9
ABS, 11
ACOS, 11
ADD_MONTHS, 13
Algemene operatoren
  syntax, 9
ALL, 10
ANY, 10
ASCII, 12
ASIN, 11
ATAN, 11
Autonome transacties, 5-6
AVG, 11
BEGIN, 19
BETWEEN, 10
BFILE, 28
BLOB, 28
CASE-expressie, 20
CASE-statement, 19
CEIL, 11
CHARTOROWID, 14
CHR, 12
CLOB, 28
CLOSE, 20. *Zie* OPEN, FETCH en CLOSE
Commentaar, 21
COMMIT, 21; 34
CONCAT, 12
controles, 4-11
controles met een lus, 4-15
CONVERT, 14
COS, 11
COSH, 11
COUNT, 11
cursor, 4-3
Cursor, 24
Cursor attribuut
  FOUND, 23
  ISOPEN, 23
  NOTFOUND, 23
  ROWCOUNT, 23
cursorattributen, 4-13
cursorgestuurde lus, 4-6
DDL statements, 22
  ALTER, 22
  AUDIT, 22
  COMMENT, 22
  CREATE, 22
  DROP, 22
  GRANT, 22
  NOAUDIT, 22
  RENAME, 22
  REVOKE, 22
  SET TRANSACTION, 22
  VALIDATE INDEX, 22
DECLARE, 23
DECLARE CURSOR, 24
DECODE, 14
DEFAULT, 23
DELETE, 21; 24
DML statements, 22
  COMMIT, 22
  DELETE, 22
  INSERT, 22
  LOCK TABLE, 22
  ROLLBACK, 22
  SAVEPOINT, 22
  SELECT, 22
  UPDATE, 22
DUMP, 14
Exception, 6-1
EXCEPTION, 25; 34
  INTERNAL, 25
  PREDEFINED, 25
  USER DEFINED, 25
EXCEPTION_INIT, 6-5; 25
EXISTS, 10
EXIT, 22; 26
EXP, 11
FETCH, 26. *Zie* OPEN, FETCH en CLOSE
FLOOR, 11
FOR, 4-5
FOR UPDATE, 5-1
Functies
  Conversie - syntax, 14
  Datum - syntax, 13
  groepsfuncties - syntax, 11
  Rekenkundige - syntax, 11
  String - syntax, 12
  Veelzijdige functies - syntax, 14
Gemiddelde, 11
generatietaal, 1-1
GOTO, 28
GREATEST, 14
Groepsfuncties
  syntax, 11

Index

HEXTORAW, 15
IF, 29
IN, 10
INITCAP, 12
INSERT, 21; 29
INSTR, 12
INSTRB, 12
Intervallen, 10
LAST_DAY, 13
LEAST, 14
LENGTH, 12
LENGTHB, 12
LIKE, 10
LN, 11
LOCK TABLE, 5-9; 30
locks, 5-4
LOG, 11
LOOP
  Cursor-gestuurd, 32
  Numeriek, 31
  ongelimiteerde, 31
  WHILE, 31
LOWER, 12
LPAD, 12
LTRIM, 12
lussen
  FOR lussen met cursor, 4-5
Maskers
  syntax, 15
MAX(imum), 11
MIN(imum), 11
MOD, 11
MONTHS_BETWEEN, 13
NEW_TIME, 13
NEXT_DAY, 14
NLS parameters
  NLS_CURRENCY, 17
  NLS_DATE_FORMAT, 17
  NLS_DATE_LANGUAGE, 17
  NLS_ISO_CURRENCY, 17
  NLS_NUMERIC_CHARACTERS, 17
  NLS_SORT, 17
  NLS_TERRITORY, 17
NLS_INITCAP, 12
NLS_LOWER, 12
NLS_UPPER, 13
NLSSORT, 13
NULL, 10; 32
NVL, 14
Omzetting
  string -> nummer, 15
OPEN, 32
OPEN, FETCH en CLOSE, 4-4; 4-11
Operatoren
  Algemene - syntax, 9
  Datumvelden - syntax, 9
  Intervallen - syntax, 10
  Rekenkundige -syntax, 9
  String - syntax, 9
  syntax, 9
  Woordpatronen - syntax, 10
Parameters, 4-7
PL/SQL, 1-2
PL/SQL Engine, 1-4
POWER, 11
PRAGMA, 6-5; 26
PRINT, 34
RAISE, 6-3; 34
RAWTOHEX, 15
REPLACE, 13
ROLLBACK, 34
ROUND, 11
ROWIDTOCHAR, 15
RPAD, 13
RTRIM, 13
SAVEPOINT, 35
SELECT, 35
SELECT ... FOR UPDATE, 5-1
SELECT ... FOR UPDATE OF, 38
SELECT INTO, 4-1
SET operator, 38
SET TRANSACTION, 35
SET TRANSACTION READ ONLY, 5-11
SHOW ERRORS, 39
SIGN, 12
SIN, 12
SINH, 12
SOME, 10
SOUNDEX, 13
SQLCODE en SQLERRM, 6-7
SQRT, 12
Standaard aanwezige packages
  DBMS_SQL, 22
STDDEV, 11
STRUCTURE, 26; 37
Substitutievariabele, 9
SUBSTR, 13
SUBSTRB, 13
SUM, 11
SYSDATE, 14
Tabellen
  p_afdelingen, 5
  p_bezetting, 6
  p_boetebedragen, 4
  p_cursussen, 3
  p_kantoren, 1
  p_mutatie, 1
  p_patienten, 6
  p_personen, 2
  p_rekening, 1
  p_spelers, 3
  p_stafleden, 5
  p_teams, 4
  p_wedstrijden, 4

Index

p_werknemers, 2
p_ziekenhuizen, 5
TAN, 12
TANH, 12
Tijdzones, 13
TO_CHAR, 15
TO_DATE, 15
TO_MULTI_BYTE, 15
TO_NUMBER, 15
TO_SINGLE_BYTE, 15
TRANSLATE, 13
TRUNC
datum, 14
getal, 12
UID, 14
UPDATE, 21; 39
UPPER, 13
USER, 14
USERENV, 14
VAR(iantie), 11
VARIABLE, 40
VSIZE, 14
WHEN OTHERS, 6-2
WHERE CURRENT OF, 5-1; 22; 24; 39
Wildcards, 10

Index

UITWERKINGEN

Oracle Database:
PL/SQL voor ervaren programmeurs

Opdrachten hoofdstuk 3

1 Maak een PL/SQL blok dat de tabel HULPTABEL vult met alle even getallen tussen 0 en 50, behalve de getallen van 20 tot en met 30. Gebruik in dit programma een FOR-loop met index (geen WHILE-loop).

```
begin
  for teller in 1..50 loop
    if mod(teller,2) = 0 and (teller <= 20 or teller >= 30) then
      insert into hulptabel
        values (teller, null, null);
    end if;
  end loop;
end;
/
```

of

```
declare
  v_teller number := 0;
begin
  for teller in 1..25 loop
    v_teller := teller * 2;
    if v_teller <= 20 or v_teller >= 30 then
      insert into hulptabel values (v_teller, null, null);
    end if;
  end loop;
end;
/
```

2 Maak een PL/SQL blok met een FOR loop dat de som van alle getallen van 1 tot en met een opgegeven getal bij elkaar optelt. Wanneer het opgegeven getal bijvoorbeeld de waarde 5 heeft, dan moet de som gelijk worden aan 15 (1+2+3+4+5). De verkregen som moet in de hulptabel komen te staan.

Vóór de optelling begint, moet worden getest of het opgegeven getal wel groter of gelijk is aan 1. Anders moet in de hulptabel de foutmelding 'Getal < 1' komen te staan.

```
accept a_nummer number prompt 'Geef een getal : '

declare
  v_som       number := 0;
  v_getal     varchar2(100);
begin
  v_getal:= &a_nummer;
  if v_getal>=1 then
      for r_positie in 1..v_getal loop
        v_som := v_som + r_positie;
      end loop;
    insert into hulptabel values(v_getal, v_som, null);
  else
    insert into hulptabel values(v_getal, 'Getal < 1', null);
  end if;
end;
/
```

3 Maak een PL/SQL programma met een FOR loop dat de nummers van bankbiljetten controleert op hun geldigheid. Bij een geldig nummer moet de som der afzonderlijke cijfers gedeeld door 9 een geheel getal opleveren. De resultaten kunt u in HULPTABEL zetten.

N.B. bankbiljetten bevatten altijd nummers van 10 cijfers lang; pas zonodig de breedte van numerieke kolommen aan met behulp van het SQL*Plus commando **set numwidth** 10.

```
accept a_nummer number prompt 'Geef het biljetnummer : '

declare
  v_biljet    number;
  v_som       number := 0;
  v_tekst     varchar2(8);

begin
  v_biljet := &a_nummer;
  for r_positie in 1..10 loop
    v_som := v_som + to_number(substr(v_biljet, r_positie, 1));
  end loop;
  if mod(v_som,9) = 0 then
    v_tekst := 'Geldig';
  else
    v_tekst := 'Ongeldig';
  end if;
  insert into hulptabel values(v_biljet, v_tekst, null);
end;
/
```

Optioneel:

4 Maak een programma dat vraagt om een naam. Schrijf vervolgens in de hulptabel een rij waarin in de tweede kolom de opgegeven naam en in de derde kolom een opmerking over die naam.
Deze opmerking is afhankelijk van de naam en wordt als volgt bepaald (gebruik hiervoor CASE):
- Is de naam langer dan 10 karakters, dan is de opmerking "Naam te lang".
- Bevat de naam een punt, dan is de opmerking "Bevat een punt".
- Is de eerste letter geen hoofdletter, dan "Begin geen hoofdletter".
In alle andere gevallen is de opmerking "Geen bijzonderheden".

```
declare
  v_naam varchar2(20) := '&naam';
  v_opmerking varchar2(20);
begin
  v_opmerking := case
                   when length(v_naam) > 10 then 'Naam te lang'
                   when instr(v_naam,'.') <> 0 then 'Bevat een punt'
                   when v_naam <> initcap(v_naam) then 'Begin geen hoofdletter'
                   else 'Geen bijzonderheden'
                 end;
  insert into hulptabel values
    ( null
    , v_naam
    , v_opmerking);
end;
/
```

5 Maak een programma waarmee een ingevoerde naam omgedraaid in de hulptabel terecht komt. Bijvoorbeeld: Marcel wordt Lecram (het eerste karakter wordt hierbij dus een hoofdletters, de rest kleine letters).

Het op te geven getal dient ingelezen te worden middels het ACCEPT commando, dat aan het PL/SQL blok voorafgaat (zie ook voorbeelden uit het boek).

```
accept a_naam char prompt 'Geef de naam : '

declare
  v_naam  varchar2(100);
  v_nieuw varchar2(100);
  v_karakter varchar2(1);
begin
  v_naam := '&a_naam';
  for r_positie in reverse 1..length(v_naam) loop
    v_karakter := substr(v_naam, r_positie, 1);
    v_nieuw := v_nieuw||v_karakter;
  end loop;
  insert into hulptabel values(null, initcap(v_nieuw), null);
end;
/
```

6 Maak een PL/SQL blok dat van een zelf op te geven getal n n! (n-faculteit) berekent. (Faculteit wil zeggen: n * (n-1) * (n-2) *...* 2 * 1, bijvoorbeeld 8! is 8*7*6*5*4*3*2*1). Het getal dient weer ingelezen te worden middels het ACCEPT commando, dat aan het PL/SQL blok voorafgaat.

```
accept a_getal number prompt 'Geef een getal a.u.b. :'

declare
  v_getal   number;
  v_invoer  number;

begin
  v_invoer := &a_getal;
  v_getal := v_invoer;

  for r_teller in reverse 1..v_invoer-1 loop
    v_getal := v_getal * r_teller;
  end loop;

  insert into hulptabel values(v_getal,null,to_char(v_invoer)||' faculteit');
end;
/
```

7 Maak een programma dat een string met namen inleest. De namen worden binnen de string gescheiden door komma's. Elke naam wordt als nieuwe rij in kolom2 van de hulptabel weggeschreven. In de rijen wordt in het eerste veld (kolom1) ook steeds een volgnummer weggeschreven, dat bij een nieuw persoon met 1 wordt opgehoogd.

```
accept a_tekst char prompt 'Geef de string : '

declare
  v_tekst  varchar2(100);
  v_stuk   varchar2(100);
  v_stop   varchar2(100) := 'nee';
  v_start  number := 0;
  v_eind   number := 0;
  v_teller number := 1;

begin
  v_tekst := '&a_tekst';

  while v_stop = 'nee' loop
    v_start := v_eind;
    v_eind := instr(v_tekst, ',' ,1, v_teller);

    if v_eind = 0 then
      v_stop := 'ja';
      v_stuk := substr(v_tekst, v_start+1);
    else
      v_stuk := substr(v_tekst, v_start+1, (v_eind)-(v_start)-1);
    end if;

    v_teller := v_teller + 1;
    insert into hulptabel values (v_teller-1, v_stuk, null);
  end loop;
end;
/
```

Opdrachten hoofdstuk 4

1 Directeur Kraay overweegt om de salarissen van zijn werknemers (tabel P_WERKNEMERS) te gaan verhogen. Hij wil een overzicht van de salarissen wanneer hij de volgende salarisverhogingen zou doorvoeren:

* 1% salarisverhoging voor alle klerken en analisten
* 2% salarisverhoging voor alle verkopers
* geen salarisverhoging voor de werknemers met overige functies

Maak een PL/SQL blok dat de nieuwe salarissen berekend en het oude salaris, het nieuwe salaris en de werknemersnaam in de HULPTABEL zet. Gebruik voor het ophalen van de werknemersgegevens een FOR-LOOP met cursor.

```
declare
 cursor c_werknemers is
         select naam
         ,      sal
         ,      functie
         from   p_werknemers;
 v_nieuw p_werknemers.sal%type;
begin
 for r_werknemers in c_werknemers loop
  case
     when r_werknemers.functie='KLERK' or r_werknemers.functie='ANALIST' then
          v_nieuw := r_werknemers.sal*1.01;
     when r_werknemers.functie='VERKOPER' then
          v_nieuw := r_werknemers.sal*1.02;
     else
          v_nieuw := r_werknemers.sal;
  end case;
  insert into hulptabel values (r_werknemers.sal, v_nieuw, r_werknemers.naam);
 end loop;
end;
/
```

2 Maak een PL/SQL blok dat van een opgegeven kantoornummer alle namen van de werknemers met hun salaris in de tabel HULPTABEL zet.

Vraag het kantoornummer op middels het ACCEPT-commando en gebruik voor het ophalen van de werknemersgegevens een FOR-LOOP met cursor.

Er hoef in dit programma nog niet gecontroleerd te worden of het opgegeven kantoornummer een bestaand nummer is. Probeer het programma met verschillende kantoornummers uit.

```
accept a_nummer char prompt 'Geef hier het kantoornummer : '

declare
  cursor c_werknemers(b_kantoor number) is
    select naam
    ,      sal
    from p_werknemers
    where kantnr = b_kantoor;

  v_kantoor number;

begin
  v_kantoor := &a_nummer;
  for r_werknemers in c_werknemers(v_kantoor) loop
    insert into hulptabel values (v_kantoor, r_werknemers.naam, r_werknemers.sal);
  end loop;
end;
/
```

3 Maak een PL/SQL blok dat van een opgegeven ziekenhuisnaam controleert of deze bestaat in de tabel P_ZIEKENHUIZEN. Als deze naam bestaat zet deze dan samen met het aantal bedden in de tabel HULPTABEL met daarbij de tekst 'Dit ziekenhuis is bekend', en anders de opgegeven naam met de tekst 'Dit ziekenhuis is niet bekend'.
Probeer het programma met verschillende namen uit.

```
accept a_ziekenhuis number prompt 'Geef hier de naam van het ziekenhuis :'

declare
  cursor c_ziekenhuis(b_naam varchar2) is
    select ziekhnr
    ,      totbed
    from   p_ziekenhuizen
    where  upper(naam) = upper(b_naam);

  v_ziekenhuisnaam varchar2(30);
  v_ziekhnr    number;
  v_bedden     number;

begin
  v_ziekenhuisnaam := '&a_ziekenhuis';
  open c_ziekenhuis (v_ziekenhuisnaam);
  fetch c_ziekenhuis into v_ziekhnr, v_bedden;
  if c_ziekenhuis%FOUND then
    insert into hulptabel values (v_bedden, 'Dit ziekenhuis is bekend',
                                  v_ziekenhuisnaam);
  else
    insert into hulptabel values (null, 'Dit ziekenhuis is niet bekend',
                                  v_ziekenhuisnaam);
  end if;
close c_ziekenhuis;
end;
/
```

4 Maak een PL/SQL blok dat het volgende uitvoert: uit de tabel P_WERKNEMERS wordt het gemiddelde salaris per functie berekend. Normaal zou u dat oplossen m.b.v. de query:

```
select functie, avg(sal)
from   p_werknemers
group by functie;
```

U gaat nu het gemiddelde berekenen met een PL/SQL-blok, maar dan **zonder** GROUP BY of SQL-functies (AVG, SUM, etc..).

U laat het programma met behulp van een loop steeds een functie uit de tabel ophalen. Per functie berekent u dan het gemiddelde salaris, door het totaal van de salarissen en het aantal mensen dat deze functie uitoefent op elkaar te delen (zonder dus gebruik te maken van de SQL-functie SUM).

Verzamel per functie de volgende gegevens in de hulptabel: het aantal mensen dat de functie uitoefent, de functienaam en het gemiddelde salaris van de functie.

```
declare
  cursor c_functies is
    select distinct functie
    from p_werknemers;

  cursor c_salaris(b_functie varchar2) is
    select sal
    from p_werknemers
    where functie = b_functie;

  v_teller number;
  v_totaal number;

begin
  for r_functie in c_functies loop
    v_teller := 0;
    v_totaal := 0;
    for r_salaris in c_salaris(r_functie.functie) loop
      v_totaal  := v_totaal + r_salaris.sal;
      v_teller  := v_teller + 1;
    end loop;
    insert into hulptabel values(v_teller, r_functie.functie, v_totaal/v_teller);
  end loop;
end;
/
```

Optioneel:

5 Maak een PL/SQL blok dat van een opgegeven ziekenhuisnummer en afdelingsnaam de stafleden die daar werken ophaalt.
Controleer eerst in tabel P_AFDELINGEN of de opgegeven afdelingsnaam en het ziekenhuisnummer bestaan. Als de afdeling in het ziekenhuis niet bestaat zet dan in de HULPTABEL het opgegeven ziekenhuisnummer en de afdelingsnaam met de tekst 'Deze afdeling is niet bekend'.
Controleer, wanneer de afdeling wel in het ziekenhuis bestaat, in de tabel P_STAFLEDEN of er stafleden op de afdeling werken. Zet, als er geen stafleden werken, in de HULPTABEL het opgegeven ziekenhuisnummer en de afdelingsnaam met de tekst 'Geen stafleden aanwezig'. Zet anders het ziekenhuisnummer, de afdelingsnaam en de namen van de stafleden die daar werken in de HULPTABEL.

Probeer het programma met verschillende afdelingsnamen en ziekenhuisnummers uit.

```
declare
 cursor c_afdelingen (b_naam varchar2, b_ziekhnr number) is
        select afdnr
        from p_afdelingen
        where upper(naam)=upper(b_naam)
        and   ziekhnr=b_ziekhnr;
 cursor c_stafleden (b_ziekhnr number, b_afdnr number) is
        select naam
        from p_stafleden
        where ziekhnr=b_ziekhnr
        and   afdnr=b_afdnr;

 v_naam varchar2(40);
 v_ziekhnr number;
 v_afdnr p_afdelingen.afdnr%type;
 v_staflid p_stafleden.naam%type;

begin
 v_naam:='&a_naam';
 v_ziekhnr:=&a_ziekhnr;
 open c_afdelingen(v_naam, v_ziekhnr);
 fetch c_afdelingen into v_afdnr;
   if c_afdelingen%notfound then
      insert into hulptabel values (v_ziekhnr, v_naam, 'Deze afdeling is onbekend');
   else
      open c_stafleden(v_ziekhnr, v_afdnr);
      fetch c_stafleden into v_staflid;
      if c_stafleden%notfound then
         insert into hulptabel values (v_ziekhnr, v_naam, 'Geen stafleden aanwezig');
      else
         while c_stafleden%found loop
            insert into hulptabel values (v_ziekhnr, v_naam, v_staflid);
            fetch c_stafleden into v_staflid;
         end loop;
      end if;
      close c_stafleden;
   end if;
 close c_afdelingen;
end;
/
```

Opdracht hoofdstuk 5

1 Maak één PL/SQL programma dat in de tabel P_WERKNEMERS de onderstaande veranderingen voor de juiste rijen doorvoert.
Geef bij de opdracht het commandoscriptje de naam PL51.SQL.

In verband met een reorganisatie worden de werknemers van de kantoren Boekhouding en Verkoop verwisseld. Daarom moet de tabel P_WERKNEMERS worden aangepast; de medewerkers van kantoor 10 wisselen met de medewerkers van kantoor 30.

Verder krijgt iedere medewerker een salarisverhoging.

- de salarissen tot en met 2000 met 8%;
- de salarissen van 2000 tot en met 2500 met 6%;
- de salarissen van 2500 tot en met 3500 met 4%;
- de salarissen boven 3500 met 2%.

Omdat het bedrijf gaat internationaliseren moeten daarnaast alle salarissen, nu in euro's, worden omgezet naar dollars (houd een koers aan van € 1,05 voor 1 dollar).

```
declare
  cursor c_werknemers is
    select *
    from p_werknemers
    for update;

  v_verhoging number;
  v_kantnr    number;

begin
  for r_werknemers in c_werknemers loop
    case r_werknemers.kantnr
      when 10 then v_kantnr := 30;
      when 30 then v_kantnr := 10;
      else v_kantnr := r_werknemers.kantnr;
    end case;

    case
      when r_werknemers.sal <= 2000 then v_verhoging := 1.08;
      when r_werknemers.sal <= 2500 then v_verhoging := 1.06;
      when r_werknemers.sal <= 3500 then v_verhoging := 1.04;
      when r_werknemers.sal > 3500 then  v_verhoging := 1.02;
    end case;

    update p_werknemers
      set sal = round((sal * v_verhoging)/1.05)
      ,   kantnr = v_kantnr
      where current of c_werknemers;

  end loop;
end;
/
select * from p_werknemers
/
rollback;
```

Optioneel:

2 Maak één PL/SQL programma dat in de tabellen P_KANTOREN en P_WERKNEMERS de onderstaande veranderingen voor de juiste rijen doorvoert:

- De gebruiker moet een kantoornaam kunnen opgeven.
- Wanneer de opgegeven kantoornaam in de tabel P_KANTOREN voorkomt, moet deze kantoornaam in de tabel P_KANTOREN gewijzigd worden in kleine letters.
- Voor de werknemers die in het opgegeven kantoor werken en als functie Manager of Directeur hebben, moet de toeslag in 500 euro worden gewijzigd.
- Voor alle overige werknemers in het opgegeven kantoor moet de toeslag in 1000 euro worden gewijzigd.

```
declare
  cursor  c_kantoren (b_naam varchar2) is
          select kantnr
          from p_kantoren
          where upper(naam)=upper(b_naam)
          for update;

  cursor  c_werknemers (b_kantnr number)is
          select naam
          ,      sal
          ,      functie
          from p_werknemers
          where kantnr=b_kantnr
          for update;
  v_kantnr  p_kantoren.kantnr%type;
  v_toeslag p_werknemers.toeslag%type;

begin
  open c_kantoren('&a_naam');
  fetch c_kantoren into v_kantnr;
  if c_kantoren%found then
    update p_kantoren set plaats=lower(plaats) where current of c_kantoren;
    for r_werknemers in c_werknemers (v_kantnr) loop
       if r_werknemers.functie='MANAGER' or r_werknemers.functie='DIRECTEUR' then
          v_toeslag:=500;
       else
          v_toeslag := 1000;
       end if;
       update p_werknemers set toeslag=v_toeslag where current of c_werknemers;
    end loop;
  end if;
  close c_kantoren;
end;
/

rollback;
```

Opdrachten hoofdstuk 6

1 Maak opgave 3 van hoofdstuk 3 nogmaals, maar dan gebruik makend van minimaal drie exceptions.

```
accept a_nummer number prompt 'Geef het biljetnummer : '

declare
  v_biljet    number;
  v_som       number := 0;
  v_tekst     varchar2(8);

  e_geen_tien   exception;
  e_negatief    exception;
  e_breuk       exception;
  e_niet_geldig exception;

begin
  v_biljet := &a_nummer;
  if v_biljet < 0 then
    raise e_negatief;
  elsif v_biljet <> trunc(v_biljet) then
    raise e_breuk;
  elsif length(v_biljet) <> 10 then
    raise e_geen_tien;
  end if;

  for r_positie in 1..10 loop
    v_som := v_som + to_number(substr(v_biljet, r_positie, 1));
  end loop;

  if mod(v_som,9) = 0 then
    insert into hulptabel values(v_biljet, 'Geldig', null);
  else
    raise e_niet_geldig;
  end if;

exception
  when e_geen_tien then
    insert into hulptabel values(v_biljet, 'Geen 10 cijfers opgegeven', null);
  when e_negatief then
    insert into hulptabel values(v_biljet, 'Nummer moet positief zijn', null);
  when e_breuk then
    insert into hulptabel values(v_biljet, 'Nummer moet geheel zijn', null);
  when e_niet_geldig then
    insert into hulptabel values(v_biljet, 'Niet geldig', null);
end;
/
```

2 Maak een programma dat van een opgegeven naam van een werknemer de naam van de manager ophaalt. De werknemer en zijn manager worden opgeslagen in de HULPTABEL.

Maak in het programma weer gebruik van het ACCEPT-commando om de naam van de werknemer in te lezen. Gebruik in ieder geval de volgende twee exceptions :
- e_geen_werkn (als de werknemer niet in de tabel P_WERKNEMERS voorkomt),
- e_geen_manager (als de werknemer geen manager heeft).

```
accept a_naam char prompt 'Geef hier de naam van de werknemer : '

declare
  cursor c_werknemer (b_naam varchar2) is
    select mgr
    from p_werknemers
    where upper(naam) = upper(b_naam);
  cursor c_manager(b_persnr number) is
    select naam
    from p_werknemers
    where persnr = b_persnr;

  v_naam    varchar2(10);
  v_manager varchar2(10);
  v_mgr     number;

  e_geen_werkn   exception;
  e_geen_manager exception;


begin
  v_naam := upper('&a_naam');
  open c_werknemer(v_naam);
  fetch c_werknemer into v_mgr;
  if c_werknemer%notfound then
    close c_werknemer;
    raise e_geen_werkn;
  else
    close c_werknemer;
    if v_mgr is null then
      raise e_geen_manager;
    else
      open c_manager(v_mgr);
      fetch c_manager into v_manager;
      close c_manager;
    end if;
  end if;
  insert into hulptabel values(null, v_naam, v_manager);

exception
  when e_geen_werkn then
    insert into hulptabel values (null, v_naam, 'Dit is geen bestaande werknemer');
  when e_geen_manager then
    insert into hulptabel values (null, v_naam, 'Deze persoon heeft geen manager');
end;
/
```

Opdrachten hoofdstuk 6

3 U gaat een programma schrijven dat de gebruiker vraagt om een getal op te geven tussen 1 en 1000. Het programma moet testen of het getal een priemgetal is. Een priemgetal is een getal dat alleen deelbaar is door 1 en door zichzelf, zoals bijvoorbeeld 2, 11, 17 en 31.

Schrijf de analyse weg in een hulptabel, gebruik makend van de volgende exceptions:
- het getal is 0 ;
- het getal is groter dan 1000 ;
- het getal is negatief ;
- het getal is niet geheel ;
- het getal is geen priemgetal. Noem ook de eerste deler.

Verder:
- behandel het getal 1 en 2 apart; dit zijn priemgetallen.
- De overige getallen deelt u eerst door 2 om te kijken of ze even zijn; dit zijn geen priemgetallen
- De oneven getallen deelt u eerst door 3, vervolgens door 5, 7, 9, 11 etc... zolang de deler kleiner of gelijk is aan de wortel van het getal.

```
accept a_getal number prompt 'Geef een getal tussen 1 en 1000 : '

declare
  v_deler      number := 3;
  v_invoer     number;
  e_nul        exception;
  e_duizend    exception;
  e_negatief   exception;
  e_nietgeheel exception;
  e_geenpriem  exception;

begin
  v_invoer := &a_getal;
  if v_invoer = 0 then
    raise e_nul;
  elsif v_invoer> 1000 then
    raise e_duizend;
  elsif v_invoer< 0 then
    raise e_negatief;
  elsif mod(v_invoer,1) <> 0 then
    raise e_nietgeheel;
  elsif mod(v_invoer,2) = 0 then
    v_deler := 2; raise e_geenpriem;
  end if;

  if v_invoer in (1,2) then
    insert into hulptabel values (null,v_invoer,'! PRIEMGETAL !');
  else
    while v_deler <= sqrt(v_invoer) loop
      if mod(v_invoer,v_deler) = 0 then
        raise e_geenpriem;
      else
        v_deler := v_deler + 2;
      end if;
    end loop;
    insert into hulptabel values (null,v_invoer,'! PRIEMGETAL !');
  end if;

exception
  when e_nul then
    insert into hulptabel values(null, v_invoer, 'Het getal is nul');
  when e_duizend then
    insert into hulptabel values(null, v_invoer, 'Het getal is groter dan 1000');
  when e_negatief then
    insert into hulptabel values(null, v_invoer,' Het getal is negatief');
  when e_nietgeheel then
    insert into hulptabel values(null, v_invoer, 'Het getal is niet geheel');
  when e_geenpriem then
    insert into hulptabel values(v_deler,v_invoer ,'Het getal is geen priemgetal');
end;
/
```

4 Maak een programma dat controleert of een ingevoerde string een postcode kan zijn.
Neem de volgende exceptions op:

- De laatste twee karakters zijn geen hoofdletters.
- Tussen de cijfers en de twee letters bevindt zich geen spatie.
- De eerste 4 karakters zijn geen cijfers
- De laatste twee karakters zijn geen tekst.

```
accept a_postcode char prompt 'Geef de postcode : '

declare
  v_postcode      varchar2(7);

  e_numeriek      exception;
  e_karakter      exception;
  e_lengte        exception;
  e_hoofdletter   exception;
  e_spatie        exception ;

begin
  v_postcode := '&a_postcode';

  if substr(v_postcode,5,1) <> ' ' then
   raise e_spatie;
  elsif length(v_postcode) < 7 then
    raise e_lengte;
  else
    for r_teller in 1..4 loop
      if substr(v_postcode,r_teller,1) not between '0' and '9' then
        raise e_numeriek;
      end if;
    end loop;

    for r_teller in 6..7 loop
      if substr(v_postcode,r_teller,1) between '0' and '9' then
        raise e_karakter;
      elsif substr(v_postcode,r_teller, 1) = lower(substr(v_postcode,r_teller,1)) then
        raise e_hoofdletter;
      end if;
    end loop;
  end if;

  insert into hulptabel values (null, v_postcode, 'Dit is een geldige postcode');

exception
  when e_numeriek then insert into hulptabel values(null, v_postcode,
    'De eerste 4 posities zijn geen numerieke waarden');
  when e_lengte then insert into hulptabel values(null, v_postcode,
    'De postcode moet in 7 karakters worden opgegeven');
  when e_karakter then insert into hulptabel values(null, v_postcode,
    'De laatste 2 posities zijn geen karakter-waarden');
  when e_spatie then insert into hulptabel values(null, v_postcode,
    'Tussen de cijfers en de twee letters moet een spatie staan');
  when e_hoofdletter then insert into hulptabel values(null, v_postcode,
    'Laatste 2 karakters moeten hoofdletters zijn');
end;
/
```

Het cursieve deel hadden we ook als volgt kunnen oplossen, waarbij we tevens de posities krijgen van de karakters waar een fout optreedt: Exception e-numeriek was dan overbodig geworden.

```
  for r_positie in 1..4 loop
    begin
      v_nummer := to_number(substr(v_postcode, r_positie, 1));
    exception
      when value_error then
        insert into hulptabel values (r_positie, v_postcode, 'Geen numerieke waarde');
    end;
  end loop;
```

Opdrachten hoofdstuk 6

Deze opgave is vanaf Oracle 10g ook op te lossen door gebruik te maken van reguliere expressies. Het programma kan er dan als volgt uitzien:

```
accept a_postcode char prompt 'Geef de postcode : '

declare
  v_postcode      varchar2(7);

  e_numeriek      exception;
  e_karakter      exception;
  e_lengte        exception;
  e_hoofdletter   exception;
  e_spatie        exception ;

begin
  v_postcode := '&a_postcode';
  if length(v_postcode) <> 7 then
    raise e_lengte;
    elsif not regexp_like(v_postcode,'^[[:digit:]]{4}') then
      raise e_numeriek;
    elsif not regexp_like(v_postcode,'^.{5}[[:alpha:]]{2}$') then
      raise e_karakter;
    elsif not regexp_like(v_postcode,'^.{5}[[:upper:]]{2}$') then
      raise e_hoofdletter;
    elsif not regexp_like(v_postcode,'^[[:digit:]]{4}[[:blank:]].{2}') then
      raise e_spatie;
  end if;
  insert into hulptabel values (null, v_postcode, 'Dit is een geldige postcode');

exception
  when e_numeriek then insert into hulptabel values(null, v_postcode,
    'De eerste 4 posities zijn geen numerieke waarden');
  when e_lengte then insert into hulptabel values(null, v_postcode,
    'De postcode moet in 7 karakters worden opgegeven');
  when e_karakter then insert into hulptabel values(null, v_postcode,
    'De laatste 2 posities zijn geen karakterwaarden');
  when e_spatie then insert into hulptabel values(null, v_postcode,
    'Tussen de cijfers en de twee letters moet een spatie staan');
  when e_hoofdletter then insert into hulptabel values(null, v_postcode,
    'Laatste 2 karakters moeten hoofdletters zijn');
end;
/
```

Opdracht hoofdstuk 7

```
declare

  -- selecteert de mutatierecords. plaats lock tbv update van code
  cursor c_mutatie is
    select *
    from   p_mutatie
    where  code = 0
    order  by volgnr
    for update of code;
  -- selecteert het saldo van degene die iets wil overschrijven/opnemen.
  -- ook lock plaatsen ivm bijweren saldo
  cursor c_rekening_van (b_nummer number) is
    select sal
    from   p_rekening
    where  rek_nr = b_nummer
    for update of sal, datum;
  -- selecteert niets, fungeert alleen ter controle van doel-rekeningnummer
  -- plaatst tevens een lock op de rij tbv de update van het saldo.
  cursor c_rekening_naar (b_nummer number) is
    select null
    from   p_rekening
    where  rek_nr = b_nummer
    for update of sal, datum;

  v_code  p_mutatie.code%type;
  v_saldo p_rekening.sal%type;
  v_dummy varchar2(1);

begin

  for r_mutatie in c_mutatie loop
    v_code := 5;
    open c_rekening_van(r_mutatie.van_rek);
    fetch c_rekening_van into v_saldo;
    if c_rekening_van%notfound then
      v_code := 3;
    else
      open c_rekening_naar(r_mutatie.nr_rek);
      fetch c_rekening_naar into v_dummy;
      if c_rekening_naar%notfound then
        v_code := 3;
      else
        if r_mutatie.soort_tr = 'A' then
          if v_saldo < r_mutatie.bedrag then
            v_code := 2;
          else
            update p_rekening
            set    sal = sal - r_mutatie.bedrag
            ,      datum = r_mutatie.datum
            where  current of c_rekening_van;
            v_code := 1;
          end if;
        elsif r_mutatie.soort_tr = 'B' then
          update p_rekening
          set    sal = sal + r_mutatie.bedrag
          ,      datum = r_mutatie.datum
          where  current of c_rekening_van;
          v_code := 1;
        elsif r_mutatie.soort_tr = 'O' then
          if r_mutatie.van_rek = r_mutatie.nr_rek then
            v_code := 4;
          elsif v_saldo < r_mutatie.bedrag then
            v_code := 2;
          else
```

Vervolg van de oplossing op de volgende pagina.

```
            update p_rekening
            set    sal = sal - r_mutatie.bedrag
            ,      datum = r_mutatie.datum
            where  current of c_rekening_van;
            update p_rekening
            set    sal = sal + r_mutatie.bedrag
            ,      datum = r_mutatie.datum
            where  current of c_rekening_naar;
            v_code := 1;
          end if;
        end if;
      end if;
      close c_rekening_naar;
    end if;
    update p_mutatie
    set    code = v_code
    where  current of c_mutatie;
    close c_rekening_van;
  end loop;

end;
/
```