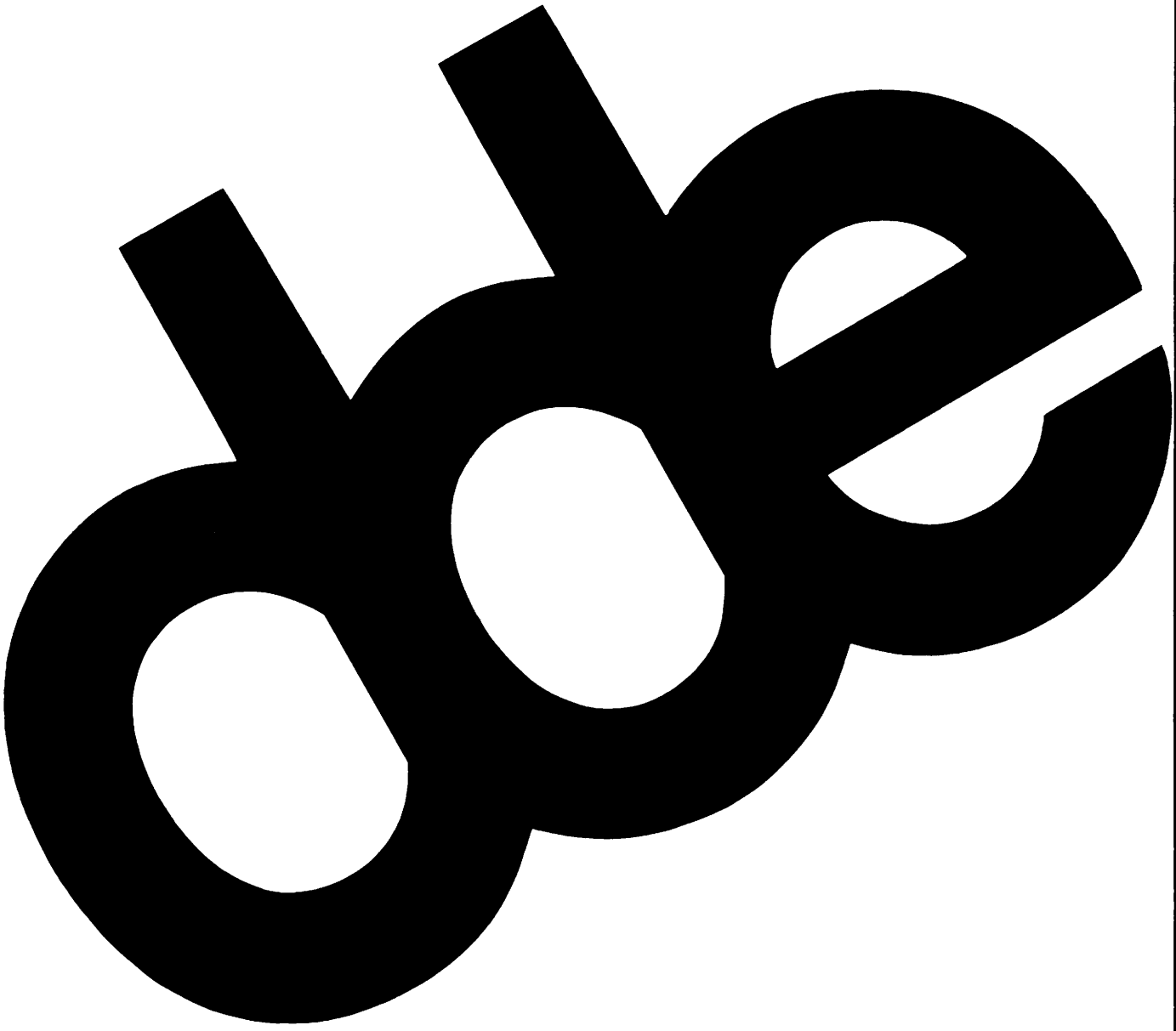
PL/SQL



marts 1995

DDE INTERN ORDRE
131 Kursusafd.
Att: Christian Bjerrgård Pedersen
Herlev Hovedgade 199
2730 Herlev

Herlev, den 04. sep. 1995

Vedr.: Deltagelse I kursus

Hermed bekræftes kursusdeltagelse for:

Kursist : Gitte Engelsholm (ge)
Kursus : Internt – PL/SQL
Dato : Den 18. til 19. september 1995

Kurset afholdes i DDEs undervisningslokaler på Herlev Hovedgade 199, Herlev i tidsrummet fra kl. 9.00 til 16.00

Vedlagt en folder med praktiske oplysninger vedr. kurset.

Vi glæder os til at se dig.

Med venlig hilsen
Dansk Data Elektronik A/S

Winnie Andreassen
Kursuskoordinator

Afmeldingsregler:

Ved afmelding senere end 14 dage før første kursusdag betales fuldt deltagergebyr.

Ændring af deltager sker uden beregning.

1. Indledning

PL/SQL (Procedural Language/SQL) er en overbygning på selve SQL*Plus. Med PL/SQL stilles et værktøj til rådighed, som giver brugeren mulighed for at skrive blok-orienterede kommandoer, der på mange måder ligner traditionelle trediegenerations programmeringsværktøjer. Samtidigt er der mulighed for at udføre almindelige SQL-sætninger indlejret i denne blokstruktur.

Fordelene ved at anvende PL/SQL er følgende:

Blokstruktur
Det er muligt at opdele PL/SQL i logiske blokke, hvor hver enkelt del håndterer en given delopgave. Fejlhåndtering og lignende kan endvidere placeres de steder, hvor den logisk hører hjemme.

Kontrolstruktur
Der findes en lang række muligheder for at kontrollere hvordan et PL/SQL-program skal forløbe. HVIS-SÅ-ELLERS konstruktioner og løkker giver mulighed for at udvikle programmer, der udfører eksempelvis transaktioner og datavalidering.

Ydelse
Ved at anvende PL/SQL sikrer en hurtigere tilgang til databasen, end det er tilfældet ved brug af almindelig SQL.

Produktivitet
PL/SQL kan opfattes som en udvidelse af SQL*Plus, med hvilket man kan udvikle programmer, der samtidigt anvendes i en række Oracle-værktøjer, eksempelvis SQL*Forms.

Anvendelighed
I praksis anvendes PL/SQL i en lang række sammenhænge, for eksempel:

- ved indlæsning og validering af større datamængder
- ved generering af statistiske informationer
- ved datatransaktioner

Eksempeltabeller

I opgaverne anvendes et antal eksempeltabeller. Tabellerne er en del af et ordresystem, for er fiktivt firma. Følgende tabeller findes:

Kunder

Tabellen indeholder alle firmaets kunder.

Navn	Null?	Type	Beskrivelse
KUNDEID	NOT NULL	CHAR(5)	Unikt kundeid
FIRMA		CHAR(40)	Firmaets navn
KONTAKTPERSON		CHAR(30)	Kontaktpersons navn
STILLING		CHAR(30)	Kontaktpersons stilling
LAND		CHAR(15)	Firmaets hjemland

Ordrer

Tabellen indeholder alle firmaets ordrer.

Navn	Null?	Type	Beskrivelse
ORDRENR	NOT NULL	NUMBER(5)	Unikt ordrenr
DATO		DATE	Ordrens dato
KUNDEID	NOT NULL	CHAR(5)	Firma, der afgiver ordre
TOTAL		NUMBER(10,2)	Ordretotal i kroner
FRAGT		NUMBER(10,2)	Fragt i kroner
MEDNR		NUMBER(4)	Den medarbejder, som har håndteret ordren.

Medarb

Tabellen indeholder alle firmaets medarbejdere.

Navn	Null?	Type	Beskrivelse
MEDNR	NOT NULL	NUMBER(4)	Medarbejderens nummer
EFTERNAVN		CHAR(20)	Efternavn
FORNAVN		CHAR(10)	Fornavn
STILLING		CHAR(30)	Stilling
FDATO		DATE	Fødselsdato
ADATO		DATE	Ansættelsesdato
REF		NUMBER(4)	Referenceperson
HYRE		NUMBER(8,2)	Løn

Uddata

Tabellen bruges til at gemme uddata fra eksempelprogrammer.

Navn	Null?	Type	Beskrivelse
TEKST		CHAR(201)	Generelt felt til uddata

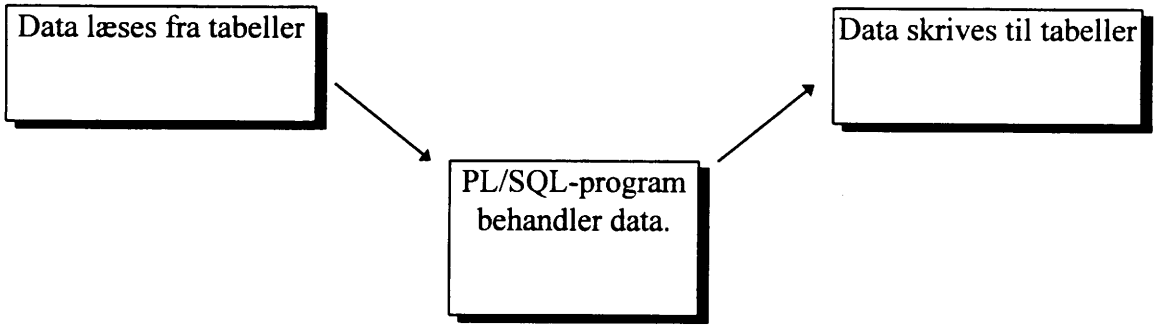
2. Introduktion til PL/SQL

Hvad kan et PL/SQL program ?

Man taler om PL/SQL-programmer, hvilket blot betyder et antal kodesætninger, der tilsammen udfører en given funktion. Et sådant program kan afvikles direkte fra en SQL kommandolinie, men kan også aktiveres på en række andre måder. I SQL*Forms aktiveres PL/SQL ofte når en given hændelse indtræffer - f.eks. ved at brugeren indtaster data i et felt.

PL/SQL programmet arbejder på tabeller, præcist som SQL. Det betyder at PL/SQL kan "se" tabellerne - altså læse data derfra - og ikke mindst kan ændre data i tabellerne - på samme måde som SQL kan. Man kan således lave et PL/SQL-program, der læser data fra tabeller, ændrer disse data, og skriver dem tilbage til en eller flere tabeller.

Uddata fra et PL/SQL program er således noget, der skrives i tabeller. Derfor opretter man ofte en simpel tabel, hvis eneste formål det er, at kunne modtage beskeder fra PL/SQL-programmet. Når programmet er afviklet, kan man derefter læse denne tabel, for at se hvordan afviklingen er forløbet.



Struktur

Et PL/SQL-program kan aktiveres direkte fra SQL*Plus kommandolinien. Man indtaster sit program med en editor og aktiverer dette program på samme måde som SQL-programmer aktiveres. Et PL/SQL program er opbygget efter nedenstående struktur:

```
DECLARE
<Her skrives erklæringer, som PL/SQL programmet
skal anvende>
BEGIN
<Her skrives PL/SQL sætninger>
EXCEPTION
<Her skrives exeptionhandlers - d.v.s. specifikati
oner af hvad der skal ske hvis noget går galt.>
END;
<Her slutter programmet.>
```

DECLARE er et reserveret ord, som angiver placeringen af de objekter, der skal erklæres inden PL/SQL-programmet kan afvikles.

BEGIN angiver starten på programmet. Når programmet afvikles vil de sætninger, der findes efter dette ord, blive afviklet linie for linie.

EXCEPTION angiver placeringen af "fejlfangere" - d.v.s. sætninger, der udføres hvis en eller anden særlig hændelse indtræffer - for eksempel hvis den datatabel, man ønsker at hente data fra er tom.

END angiver afslutningen på programmet.

Kommentarer er tekst i programmet, som ikke medtages af PL/SQL - og som derfor udelukkende fungerer, som beskrivelser af hvad programmet udfører. Der findes to typer kommentarer.

- -- (to tankestreger) sikrer at en linie ikke medtages af PL/SQL.
- /* og */ sikrer at den samlede blok, ikke medtages af PL/SQL.

```
DECLARE
   -- Denne linie medtages ikke af PL/SQL.

BEGIN
   /* Denne blok medtages ikke af PL/SQL, fordi
       den er omkranset af kommentar-tegn. */
END;
```

Eksempel:

I firmaets ordresystem findes en tabel, inventory_table, som indeholder information om hvilke produkter der er på lager. Tabellen indeholder følgende felter:

- PROD_ID
- PRODUCT
- QUANTITY

Hvis man ønsker at se alle produkter, kan dette gøres med nedenstående SQL-kommando:

```
SQL> select * from inventory_table;
```

Forespørgslen giver følgende svar:

```
   PROD_ID PRODUCT            QUANTITY
---------- --------------- ----------
      1234 TENNIS RACQUET             3
      8159 GOLF CLUB                  4
      2741 SOCCER BALL                2
```

Hvis man ønsker at se information om et bestemt produkt, kan dette gøres med nedenstående SQL-kommando:

```
SQL> select * from inventory_table where product = 'GOLF CLUB';
```

Forespørgslen giver følgende svar:

```
   PROD_ID PRODUCT           QUANTITY
---------- -------------- ----------
      8159 GOLF CLUB               4
```

Lad os herefter oprette et simpelt PL/SQL program, der på sammen måde læser information fra tabellen inventory_table. PL/SQL kan ikke skrive resultatet direkte på skærmen, men skal i stedet skrive dette til en tabel. Til dette formål har vi en tabel uddata, der indeholder et enkelt felt, "tekst". Vi ønsker nu at læse data fra inventory_table og skrive disse i tabellen uddata.

Dette gøres på følgende måde:

1. **Opret en variabel, hvori data kan gemmes**
2. **Skriv en SELECT-sætning, der henter data fra tabellen til variablen**
3. **Skriv indholdet af variablen til tabellen UDDATA**
4. **COMMIT ændringerne**

```
0.   -- dette program indsætter lagerantal i tabellen uddata.
1.   DECLARE
2.           antal_paa_lager NUMBER(5);
3.   BEGIN
4.           SELECT quantity INTO antal_paa_lager
5.           FROM inventory_table
6.           WHERE product = 'TENNIS RACQUET';
7.
8.           INSERT INTO uddata VALUES to_char((antal_paa_lager));
9.           COMMIT;
10.  END;
```

Forklaring:

0. Kommentar.
1. Her starter PL/SQL-programmet
2. Erklæring af variablen antal_paa_lager som er af typen NUMBER(5)
3. Her starter de sætninger, som vil blive udført når programmet afvikles
4.-6. Med en SELECT-sætning placeres indholdet af feltet quantity i variablen antal_paa_lager
8. Med en SELECT-sætning placeres indholdet af variablen antal_paa_lager i tabellen uddata
9. Ændringerne skrives til databasen
10. Her afsluttes programmet

Ovenstående kodeeksempel er så enkelt, at det uden problemer kunne laves med almindelig SQL. Men en helt afgørende forskel på SQL og PL/SQL er begrebet variable. Hvis man for eksempel havde ønsket at fordoble antallet af tennis-ketchere på lageret - kunne dette gøres ved at indføre nedenstående sætning i linie 7:

```
antal_paa_lager := antal_paa_lager * 2;
```

Sætningen ganger antal_paa_lagers værdi med 2, og placerer resultatet i samme variabel antal_paa_lager. Dette gøres ved hjælp af en tildelingsoperator. Sætningen kan læses som:

```
antal_paa_lager får værdien antal_paa_lager * 2
```

Hermed får det samlede PL/SQL-program følgende form:

```
0.  DECLARE
1.          antal_paa_lager NUMBER(5);
2.  BEGIN
3.          SELECT quantity INTO antal_paa_lager
4.          FROM inventory_table
5.          WHERE product = 'TENNIS RACQUET';
6.          antal_paa_lager := antal_paa_lager * 2;
7.          INSERT INTO uddata VALUES to_char(antal_paa_lager);
8.          COMMIT;
9.  END;
```

Det er muligt at hente flere felter i en select-sætning. Dette kan gøres på følgende måde:

```
SELECT quantity, prod_id INTO antal_paa_lager, produkt from inventory_table;
```

PL/SQL i praksis

Når ovenstående eksempel skal prøves i praksis gøres dette ved at følge nedenstående fremgangsmåde:

1. **Start SQL*Plus. Dette gøres med kommandoen sqlplus brugernavn/password**
2. **Aktivér redigeringsfunktionen med kommandoen ed <filnavn>, f.eks. ed lagerbehold**
3. **Indtast programlinierne - afslut eventuelt med / ("slash") på linien efter sidste END.**
4. **Gem filen og afslut editoren**
5. **Aktivér programmet med kommandoen @<filnavn>, f.eks. @lagerbehold**

Herefter afvikles programmet. Hvis alt går som det skal, kvitterer PL/SQL med følgende meddelelse på skærmen:

```
PL/SQL procedure successfully completed.
```

Endelig kan man undersøge den eller de tabeller, som PL/SQL har ændret. Hvis programmet ikke kan udføres, skrives en fejlmeddelelse på skærmen. Dette gennemgås senere på kurset.

Opgave 2.1

Formål

At oprette og afvikle et enkelt PL/SQL-program.
Tabel: MEDARB, UDDATA

1. Skriv et PL/SQL-program, der gør følgende.

   - Indlæser fornavn for medarbejderen med nummer 1 i en variabel.
   - Skriver medarbejderens fornavn til tabellen uddata

2. Kør programmet og undersøg om det virker korrekt.

3. Udbyg programmet således at det automatisk viser og sletter indholdet af tabellen uddata.

Opgave 2.2

Formål

At oprette og afvikle et enkelt PL/SQL-program, der beregner på variable.
Tabel: MEDARB, UDDATA

1. Skriv et PL/SQL-program, der gør følgende.

   - Indlæser løn for medarbejderen med nummer 4 i en variabel.
   - Reducerer lønnen med 10 %.
   - Skriver den nye løn til tabellen uddata.

2. Kør programmet og undersøg om det virker korrekt.

Opgave 2.3

Formål

At kommentere PL/SQL-programmer.

1. Udbyg programmet, der blev oprettet i opgave 2.2 med passende kommentarer.

2. Kør programmet og undersøg om det virker korrekt.

3. Opbygningen af PL/SQL

Variable

Variable er opbevaringssteder for værdier, f.eks. resultater af forespørgsler eller beregninger. Variable har den store fordel at de kan opbevare en værdi, som kan genanvendes når der er brug for den.

En variable skal **erklæres** før det kan anvendes. Det betyder at PL/SQL skal kende variablens navn og dens type. Typen angiver hvilken slags information variablen kan opbevare. I PL/SQL findes følgende typer:

NUMBER(antal cifre, antal decimaler)	Opbevaring af tal - f.eks. NUMBER(10,2), som kan opbevare tal på maximalt 10 cifre, hvoraf 2 er decimaler. Maksimal værdi er 38 cifre, mens decimalerne kan være -84 til 127. Hvis en NUMBER-variabel erklæres uden angivelse af antal cifre eller decimaler, vælges den maksimale værdi automatisk.
CHAR(længde)	Strenge af tegn. Længde angiver antallet af tegn. Maksimalt kan 255 tegn opbevares.
DATE	Almindelige Oracle datoer, der kan ligge intervallet 4712 F.KR. - 4712 E.KR.
BOOLEAN	Denne type kan kun opbevare værdierne sand eller falsk. Typen findes ikke i almindelig SQL.

Erklæringer af variable sker ved at angive et navn for variablen og en type. F.eks.

```
Navn CHAR(20);

Pris NUMBER(6,2);

Paa_lager BOOLEAN;

Fødselsdag DATE;
```

Den generelle syntaks er:

```
<variabelnavn> <type>;
```

Bemærk at der skal være mellemrum mellem navn og type - og at sætningen afsluttes med semikolon (;).

Bemærk:

- Et variabelnavn må ikke indeholde mellemrum.
- længden må højest være 30 tegn.
- variablens navn skal starte med et bogstav.
- herefter kan følge enhver kombination af bogstaver og tal, samt tegnene $, _ og #.
- der er ingen forskel på små og store bogstaver - f.eks. opfattes Navn og NAVN som ens. Dette gælder dog ikke hvis variabelnavnet er placeret i anførselstegn.

navn	tilladt
adresse	tilladt
telefon nummer	forbudt
fornavn/efternavn	forbudt
adresse2	tilladt
ansættelses_dato	tilladt

Tildeling af værdier til variable

Variablens primære formål er at opbevare værdier. Værdier tilknyttes variable ved **tildelinger**. Dette kan ske på flere måder.

1. Ved anvendelse af tildeling af en konstant værdi:

```
Navn := 'Ole Olsen'
Pris := 1234
```

2. Ved tildeling af en værdi, der er tilknyttet en fremmed variabel eller et udtryk:

```
Pris := GammelPris;
Pris := GammelPris * 1.10;
Pris := PrisUdenMoms * 1.25;
Moms := Pris*0.20;
```

Konstanter

En variabel kan erklæres således at den oprettes med en speciel værdi. Dette gøres på følgende måde:

```
<konstant-navn> CONSTANT <type> := <værdi>;

MOMS CONSTANT NUMBER := 0.25;
```

Tildeling af værdier til variable v.h.a. SQL-sætninger

Ofte ønsker man at hente data fra tabeller, og placere disse i variable. Man har f.eks. en tabel kaldet Kunder, som har felterne Kunde_Navn og Kunde_Nr. Hvis man ønsker at placere Kunde_Navn i en variabel, kan dette gøres på følgende måde:

```
SELECT Kunde_Navn INTO Denne_Kunde FROM Kunder
WHERE Kunde_nr = 2341;
```

I dette eksempel placeres værdien af Kunde_Navn i variablen Denne_Kunde. Naturligvis skal variablen være erklæret før SELECT-sætningen kan aktiveres. På samme måde er det muligt at foretage en beregning v.h.a. selve SELECT-sætningen:

```
SELECT Pris*1.25 INTO PrisMedMoms FROM Produkter
WHERE Produkt_nr = 11;
```

I dette eksempel placeres værdien af feltet Pris*1.25 i variablen PrisMedMoms.

Indsættelse af værdier fra variabel til tabel v.h.a. SQL-sætninger

Det er også muligt at placere værdien af en variabel i en tabel. Dette kan f.eks. gøres ved hjælp af SQL-kommandoerne INSERT eller UPDATE.

```
0.   DECLARE
1.      NyPris number(10,2);
2.      GammelPris number(10,2);
3.
4.   BEGIN
5.     SELECT Pris INTO GammelPris FROM Produkter
6.     WHERE Produkt_nr = 11;
7.
8.     NyPris := GammelPris * 1.10;
9.
10.    UPDATE Produkter SET Pris = NyPris
11.    WHERE Produkt_nr  = 11;
12.
13.    INSERT into Historik values(GammelPris, 11);
14.    COMMIT;
15.  END;
```

I eksemplet hentes en værdi fra tabellen Pris - nemlig prisen for produkt nummer 11. Den nye pris beregnes, og resultatet placeres i en variabel NyPris. Endelig erstattes den gamle pris med den nye, ved hjælp af en UPDATE-kommando. Til sidst placeres den gamle værdi i en historik-tabel, der bruges til at gemme gamle priser. Dette gøres ved hjælp af en INSERT-kommando.

Variable i where-kriterier

En variabel kan anvendes i where-kriteriet i en select-sætning.

```
    SELECT FROM <tabelnavn> WHERE <feltnavn> = <variabel>
SELECT FROM Produkter WHERE prod_id = produkt_der_skal_slettes;
```

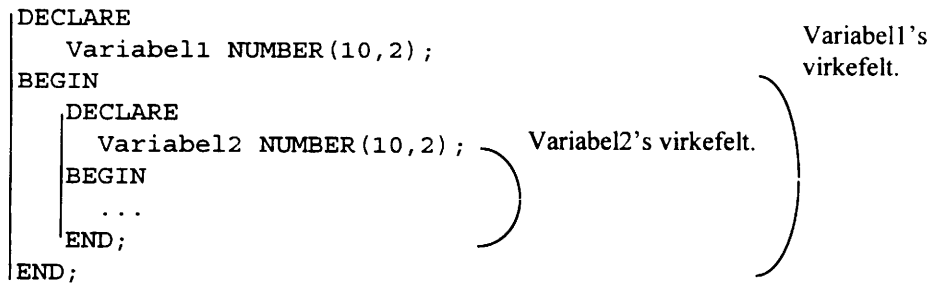
Sletning v.h.a. SQL

Det er også muligt at slette rækker i tabeller v.h.a. data i variable. Dette gøres med følgende generelle syntaks:

```
    DELETE FROM <tabelnavn> WHERE <feltnavn> = <variabel>
DELETE FROM Produkter WHERE prod_id = produkt_der_skal_slettes;
```

Blokke og virkefelter

Variable kan erklæres på forskellige måder, afhængigt af hvordan - og især hvor i programmet - man ønsker at de skal virke. **Dette kaldes variablens virkefelt**. For at forstå virkefelter må man også forstå PL/SQL's blokstruktur.



På ovenstående tegning findes et skitseret program, som anvender to variable - Variabel1 og Variabel2. Variabel1 erklæret i den yderste blok, og kan derfor ses i hele programmet. Variabel2 er erklæret i den inderste blok og kan derfor kun ses i denne blok.

Blokopbygningen af PL/SQL-programmer gør det muligt at skrive pænere programmer, der har en struktur, som er naturlig og logisk for opgavens løsning. Virkefelter betyder at man kan erklære variable, hvor de skal bruges.

Ved navnesammenfald af variable anvendes den variabel, der er placeret tættest på det aktive blokniveau.

Opgave 3.1

Formål

At skrive et PL/SQL-program, der opdaterer data i en tabel.
Tabel: MEDARB, UDDATA

1. Skriv et PL/SQL-program, der gør følgende.

- Indlæser lønnen for medarbejderen med nummer 2 i en variabel.
- hæver lønnen med 15 % og opdaterer feltet hyre.
- skriver den nye og gamle løn i tabellen uddata
- viser resultatet på skærmen - v.h.a. tabellen uddata.
- slette indholdet af tabellen uddata

2. Kør programmet og undersøg om det virker korrekt.

3. Udbyg programmet således at også medarbejderens fornavn og efternavn skrives i tabellen uddata.

4. Kør programmet og undersøg om det virker korrekt.

Opgave 3.2

Formål

At skrive et PL/SQL-program, der sletter data i en tabel.
Tabel: KUNDER, UDDATA

1. Skriv et PL/SQL-program, der gør følgende.

   - sletter kunden med kundeid LETSS
   - registrerer sletningen ved at skrive KUNDEID, FIRMA og dags dato i tabellen uddata.

2. Kør programmet og undersøg om det virker korrekt.

TIP:
Dags dato findes ved at bruge den indbyggede funktion sysdate.

Opgave 3.3

Formål

At skrive et PL/SQL-program, der beregner på datoer
Tabel: KUNDER, UDDATA

1. Skriv et PL/SQL-program, der gør følgende.

   - beregner hvor mange dage medarbejderen med nummer 10 har været ansat.
   - skriver antallet af dage, stilling samt efternavn i tabellen uddata.

2. Kør programmet og undersøg om det virker korrekt.

4. Udtryk

Et udtryk er en PL/SQL-sætning, som beregner en værdi, der typisk placeres i en variabel. Generelt anvendes nedenstående syntaks:

```
Variabel := Udtryk;
```

En variabel, er en opbevaringsplads for en værdi - et udtryk er en sætning, der på den ene eller anden måde beregner en værdi.

```
NyPris := 1200;
NyPris := NyPris * 1.15;
```

Der findes i PL/SQL en række operatorer. Eksempler på disse er + og -, som man kender fra almindelig regning. Når udtryk skal beregnes, må PL/SQL vide i hvilken rækkefølge de enkelte operatorer skal evalueres. Nedenstående skema viser, operatorernes rækkefølge:

	Operator	**Operation**
1.	**, NOT	Potensopløftning, logisk negation
2.	*, /	Multiplikation, division
3.	+, -, ||	Addition, subtraktion, tekstsammenlægning
4.	=, !=, <, >, <=, >=, IS NULL, LIKE, BETWEEN, IN	Sammenligning
5.	AND	Logisk AND
6.	OR	Logisk Or

Parenteser ophæver rækkefølgen, idet udtryk i parenteser beregnes samlet.

Eksempel:

Pris := 100-20*2;	Pris får værdien 60, fordi 20*2 beregnes først.
Pris := (100-20)*2;	Pris får værdien 160, fordi parenteserne beregnes samlet.

Datatyper i udtryk

Beregning af udtryk må ske under hensyntagen til de datatyper, der involveres i beregningen. For eksempel:

```
DECLARE
   PrisIalt NUMBER(7,2);
BEGIN
   PrisIalt := 'Prisen er 1234 kroner';
END;
```

Dette program vil resultere i en fejl, fordi man ikke kan placere tekst i en numerisk variabel. PL/SQL vil normalt forsøge at konvertere til rette datatype, men dette kan - som vist i ovenstående eksempel - ikke altid lade sig gøre. Derfor bør man ved tvivl anvende PL/SQL's konverteringsfunktioner.

```
Overskrift := ' Ole Olsen tjener ' || TO_CHAR(måneds_løn*12);
```

I ovenstående sætning findes en variabel Overskrift, der er af typen CHAR. Variablen ønskes tildelt en værdi, der er en kombination af tekst og tal. Månedsløn er en variabel af typen NUMBER, som i eksemplet skal ganges med 12. For at sikre at der ikke opstår datakonverteringsfejl, anvendes funktionen TO_CHAR(), som konverterer et tal eller en dato til en tekst. Tilsvarende findes funktionerne TO_DATE og TO_NUMBER.

%Type erklæring

Den række af problemer, som kan opstå p.g.a. typer, kan stort set løses ved hjælp af en såkaldt %TYPE erklæring. Ideen er, at man erklærer en variabel, således at den har samme type som en kolonne i en tabel, **uanset** hvilken type denne kolonne har.

Denne generelle syntaks for ved anvendelse af en %Type erklæring er følgende:

```
<Variabelnavn> <Tabel>.<Kolonne>.%TYPE;
```

%TYPE erklæringen bevirker således at variablens type først bestemmes når koden afvikles. Dette sikrer at eventuelle typeskift i tabellen ikke vil kræve nye erklæringer i koden, fordi variablen automatisk "arver" feltets type.

Eksempel:

I nedenstående programstump ses en erklæring af en variable, der oprettes således at den har samme type som feltet CREDITLIMIT i tabellen CUSTOMER. Konstruktionen er hensigtsmæssig fordi variablen gammel_kredit altid anvendes til at læse information fra feltet CREDITLIMIT.

```
0.  DECLARE
1.     gammel_kredit customer.creditlimit%TYPE;
2.  BEGIN
3.     select creditlimit into gammel_kredit
4.     from customer
5.     where custid = 105;
6.     ...
7.     ...
8.  END;
```

Funktioner i udtryk

De fleste funktioner i SQL kan også anvendes i PL/SQL-udtryk. Dette gælder for flg. funktionskategorier:

- numeriske funktioner, der virker på en enkelt række.
- tekstfunktioner, der virker på en enkelt række.
- konverteringsfunktioner
- datofunktioner
- andre

Eksempel:

```
0.  DECLARE
1.     Navn emp.ename%TYPE;
2.  BEGIN
3.     SELECT ename INTO Navn
4.     FROM emp WHERE ename =  'ALLEN';
5.     Navn := INITCAP(Navn);
6.     INSERT INTO uddata VALUES(Navn);
7.     COMMIT;
8.  END;
```

I eksemplet indlæses et medarbejdernavn i variablen Navn. Denne variabel ændres således at første bogstav er stavet med stort - v.h.a. et udtryk, der anvender funktionen INITCAP. Endeligt skrives værdien af Navn til tabellen uddata.

Gruppefunktioner i udtryk

Gruppefunktioner kan ikke anvendes i udtryk. Det gælder nedenstående funktioner:

- AVG
- MIN
- MAX
- COUNT
- SUM
- STDDEV
- VARIANCE

Dog er det muligt at skrive SQL-sætninger i PL/SQL-programmet, som anvender ovennævnte funktioner, og placerer resultatet i en variabel.

Eksempel:

```
0.  DECLARE
1.    antal emp.sal%TYPE;
2.    total emp.total%TYPE;
3.    gsnit emp.total%TYPE;
4.
5.  BEGIN
6.    SELECT count(Sal),sum(sal) INTO antal, total  FROM emp;
7.    gsnit := total/antal;
8.    INSERT INTO uddata VALUES('Gennemsnit er: '|| to_char(gsnit));
9.    COMMIT;
10  END;
.


SQL> select * from uddata;

TEKST
------------------------------
Gennemsnit er: 2073.21
```

Oversigt over funktioner

I det følgende findes en oversigt over de vigtigste funktioner, der kan anvendes i PL/SQL. Funktionerne er inddelt efter følgende kategorier:

- **Numeriske funktioner, der virker på én række.**
- **Tekstfunktioner, der virker på én række.**
- **Konverteringsfunktioner.**
- **Datofunktioner.**
- **Fejlhåndteringsfunktioner.**
- **Gruppefunktioner.**
- **Andre funktioner.**

Numeriske funktioner, der virker på én række.

ABS(n)	Returns the absolute value of n.
CEIL(n)	Returns the smallest integer greater than or equal to n.
FLOOR(n)	Returns the largest integer less than or equal to n.
MOD(m,n)	Returns result of m divided by n; or if n=0, m returned.
POWER(m,n)	Returns m raised to the nth power; n must be an integer.
ROUND(m,n)	Returns m rounded to n (optional) decimal places.
SIGN(n)	Returns -1 if n is negative, 1 if n is positive, 0 if n=0.
SQRT(n)	Returns the square root of n.
TRUNC(m,n)	Strips m to n (opt) places; -n adds n zeros to an integer.

Tekstfuntioner, der virker på én række.

ASCII(string)	Returns collating sequence value of 1st char of string.
CHR(n)	Returns character having ASCII or EBCDIC value of n.
INITCAP(char)	Capitalizes each word's first letter, others lowercase.
INSTR(char1, char2, m, n)	Beginning at position m, returns the position of the nth occurrence of char2 in char1; m & n = 0 if absent.
LENGTH(char)	Returns the length of char.
LOWER(char)	Returns char in lowercase.
LPAD(char1, n, char2)	Adds char2 n number of times in front of char1.
LTRIM(char, s)	Removes beginning characters from char, up to the first character not in the s set; s = "if absent.
NLSSORT(char)	Gives char's National Language collating sequence.
REPLACE(char, find, new)	Changes each find in char to new; omit new to erase find. Use TRANSLATE for single characters.
RPAD(char1, n, char2)	Adds char2 n number of times after char1.
RTRIM(char, set)	Removes characters from the end of char until the first character not in set; set = "if absent.

SOUNDEX(char)	Returns string phonetically equivalent to char.
SUBSTR(char, c, n)	Returns n characters in char, starting at char c.
TRANSLATE(char, find, new)	Returns char with each find changed to new.
UPPER(char)	Returns char in UPPERCASE.

Konverteringsfunktioner.

TO_CHAR(expr, fmt)	Turns a number or date expr into characters shown in the format specified by fmt (optional).
TO_DATE(char, fmt)	Turns a char date in fmt format into a date value.
TO_NUMBER(char)	Turns a CHAR number into its NUMBER datatype value.
CHARTOROWID(char)	Turns character values in char into ROWID values.
ROWIDTOCHAR(ROWID)	Turns ROWID values into character values.
CONVERT(char, new, source)	Converts char to a new character set implementation from its source implementation
HEXTORAW(char)	Turns char hexadecimal digits into binary (raw).
RAWTOHEX(raw)	Turns raw (binary) into character (hex) values.

Datofunktioner.

ADD_MONTHS(d, n)	Returns date d with n months more; n = integer.
LAST_DAY(d)	Returns date of the last day of d date's month.
MONTHS_BETWEEN(d, e)	Returns number of months between dates d and e.
NEW_TIME(d, a, b)	Given date d in time zone a, returns date/time in zone b, using time zone abbreviations.
NEXT_DAY(d, day)	Returns date of the day (Tue) coming after date d.
SYSDATE	Returns the current date and time.
ROUND(d, format)	Returns d rounded by a specified format.
TRUNC(d, format)	Returns d truncated by a specified format; TRUNC with no format strips time from a date.

Fejlhåndteringsfunktioner.

SQLCODE	Returns the ORACLE error code of the internal exception that passed control to an exception handler. Or, if the exception is declared by the user within the current (or an enclosing) block, SQLCODE returns +1. Outside an exception handler, it returns 0. SQLLCODE lets you identify exceptions in an OTHERS handler.
SQLERRM(error_code)	Returns the error message for the error_code you specify. To have SQLERRM return the current SQLCODE value's message, omit the error_code. Outside an exception handler, SQLERRM gives the message "normal, successful completion".

Gruppefunktioner.

AVG(D|A expr)	Gives average of expr values across rows.
COUNT(D|A expr)	Gives number of rows where expr is not null.
COUNT(*)	Gives total number of rows, including nulls.
MAX(D|A expr)	Gives maximum expr value across rows.
MIN(D|A expr)	Gives minimum expr value across rows.
STDDEV(D|A expr)	Gives standard deviation of expr values in rows.
SUM(D|A expr)	Gives the sum of expr values across rows.
VARIANCE(D|A expr)	Gives variance of expr values across rows.

D|A = DISTINCT values or ALL (default=ALL). Null values are ignored except in COUNT(*).

Andre funktioner.

DECODE(expr, search1, result1, result2,...default)	Returns result1 if expr = search1,search2,
NVL(expr1, expr2)	Returns expr1 if not null; if null, returns expr2. Returned datatype must match expr1.
UID	Returns the unique integer identifier of each user.
USER	Returns the currently signed on username.
USERENV('ENTRYID')	Returns available auditing entry identifier.
USERENV('LANGUAGE')	Returns language set by LANGUAGE INIT.ORA.
USERENV('SESSIONID')	Returns user's auditing session identifier.
USERENV('TERMINAL')	Returns terminal's operating system identifier.
VSIZE(expr)	Returns the number of bytes used to store the ORACLE internal representation of expr.

Opgave 4.1

Formål

At skrive et PL/SQL-program, der beregner på tal
Tabel: ORDRER, UDDATA

1. Skriv et PL/SQL-program, der gør følgende.

   - beregner det totale ordrebeløb - fragt + total - for ordrenr 10050 - og lægger moms oveni.
   - skriver ordrenr og den samlede total til tabellen uddata.

2. Kør programmet og undersøg om det virker korrekt.

Opgave 4.2

Formål

At skrive et PL/SQL-program, der beregner på totaler.
Tabel: KUNDER, ORDRER, UDDATA

1. Skriv et PL/SQL-program, der gør følgende.

   - beregner gennemsnitstotalen for alle ordrer fra USA.
   - beregner gennemsnitstotalen for alle ordrer fra Frankrig.
   - beregner forskellen mellem de to tal.
   - beregner forskellen i procent mellem de to tal.
   - Skriver begge forskelle til tabellen uddata.

2 Kør programmet og undersøg om det virker korrekt.

Opgave 4.3

Formål

At skrive et PL/SQL-program, der regner på datoer.
Tabel: KUNDER, ORDRER, UDDATA

1. Skriv et PL/SQL-program, der gør følgende.

   - beregner forskellen i ansættelsestid mellem de to medarbejdere, der har været ansat henholdsvist kortest og længst tid i firmaet. Forskellen skal udtrykkes i måneder.
   - skriver forskellen til tabellen uddata.

2 Kør programmet og undersøg om det virker korrekt.

TIP:
For at udtrykke forskellen i måneder skal der anvendes en indbygget funktion.

Opgave 4.4

Formål

At skrive et PL/SQL-program, anvender variable i where-kriteriet.
Tabel: KUNDER, ORDRER, UDDATA

1. Skriv et PL/SQL-program, der gør følgende.

   - finder den største ordre i ordretabellen.
   - finder kundeid for denne ordre.
   - finder navnet på det firma, der har afgivet ordren.
   - skriv firmaets navn til tabellen uddata.

2 Kør programmet og undersøg om det virker korrekt.

3 Udbyg programmet således at det samtidigt finder den gennemsnitlige ordretotal, for alle ordrer, der kommer fra samme land, som den største ordre.

4 Kør programmet og undersøg om det virker korrekt.

5. Betingelser

Betingelser er kontrolstrukturer, der anvendes til at styre hvordan et programs afvikling forløber. V.h.a. betingelser er det muligt at sikre, at kun bestemte kodelinier bliver udført, når en særlig betingelse er opfyldt.

Betingelser er et begreb der kendes fra langt de fleste procedurale trediegenerationsværktøjer, og er et af de punkter, hvor PL/SQL markant adskiller sig fra almindelig SQL.

Eksempel:

I firmaet findes en registrering af de enkelte kunders kreditværdighed. Dette er gjort ved at der for hver kunde findes en kreditgrænse, som fortæller hvor stor en given kundes kredit må være på et givent tidspunkt. På grund af særlige omstændigheder ønsker vi at reducere denne kredit med 10 % for en bestemt kunde, hvis kundens samlede køb ligger under 50.000 kroner. Hvis det samlede køb ligger over, er kreditten uændret. Dette gøres med nedenstående PL/SQL-program, der anvender betingelser.

```
0.   DECLARE
1.      kredit customer.creditlimit%TYPE;
2.      gammel_kredit customer.creditlimit%TYPE;
3.      ordre_total ord.total%TYPE;
4.   BEGIN
5.      SELECT creditlimit INTO gammel_kredit
6.      FROM customer
7.      WHERE custid = 105;
8.
9.      SELECT SUM(total) INTO ordre_total
10.     FROM ord
11.     WHERE custid = 105;
12.
13.     IF ordre_total < 50000 THEN
14.        kredit := gammel_kredit * 0.90;
15.        UPDATE customer SET creditlimit = kredit
16.        WHERE custid = 105;
17.        INSERT INTO uddata VALUES('105:'||to_char(gammel_kredit)
                  ||'/'||to_char(kredit));
18.        COMMIT;
19.     ELSE
20.        INSERT INTO uddata VALUES('Kredit uændret');
21.        COMMIT;
22.     END IF;
23.  END;
24.  /
25.  SELECT * FROM uddata;
```

Hent maks. kredit for kunden.

Hent det samlede køb.

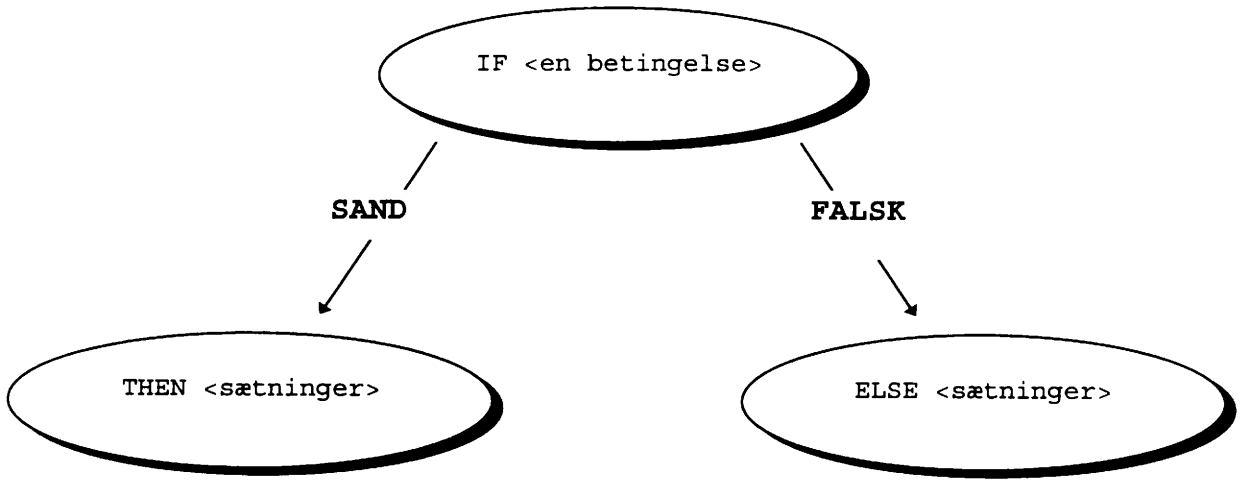
Hvis køb er mindre end 50000, så reducer med 10 %.

Programmet svarer med følgende besked:

```
PL/SQL procedure successfully completed.

TEKST
------------------------------
105: 5000/4500
```

IF-THEN-ELSE



En IF-THEN-ELSE konstruktion opbygges på følgende måde:

```
IF <EN LOGISK BETINGELSE> THEN
<ET ANTAL PL/SQL-SÆTNINGER>
ELSE
<ET ANTAL PL/SQL-SÆTNINGER>
END IF;
```

Oftest opbygges konstruktionen således at et antal inddata læses før IF-sætningen - og at denne sætning styrer hvilke SQL-sætninger - ofte opdateringer - der udføres. Bemærk at IF-THEN-ELSE konstruktionen afsluttes med sætningen END IF, der afsluttes med et semikolon.

IF-THEN-ELSE konstruktioner i PL/SQL afløser på en lang række områder funktionen DECODE, der findes i almindelig SQL.

Logiske operatorer

En central del af en IF-THEN-ELSE konstruktion er et eller flere logiske udtryk, der styrer hvordan koden afvikles. Der findes til opbygning af sådanne udtryk et antal logiske operatorer.

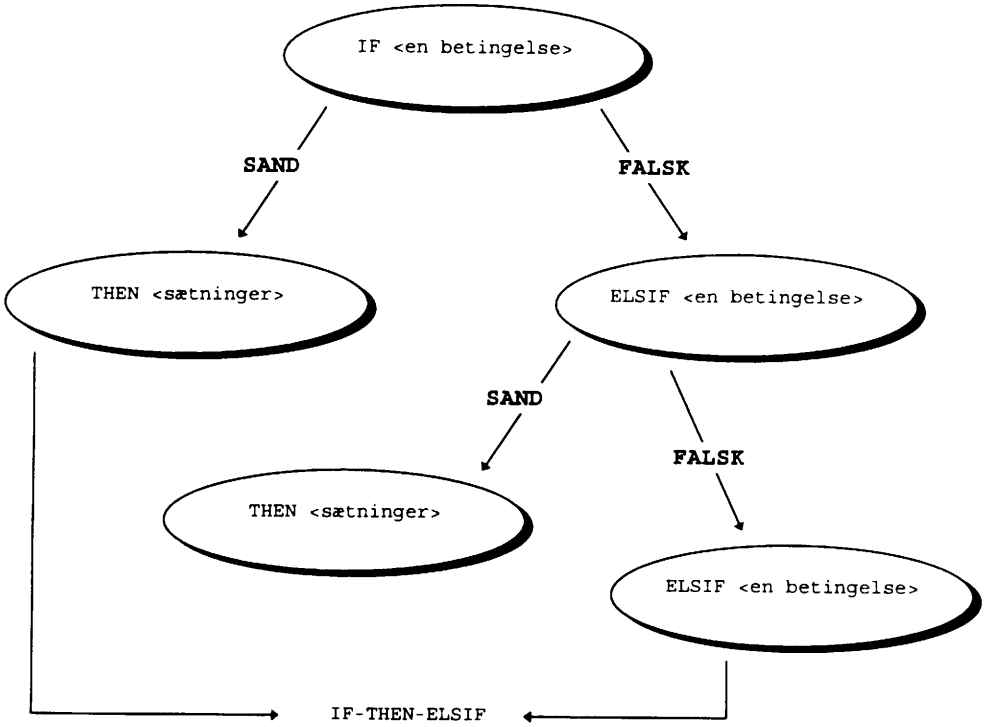
1.	=	Lig med
2.	!=	Ikke lig med
3.	<	Mindre end
4.	>	Større end
5.	<=	Mindre end eller lig med
6.	>=	Større end eller lig med
7.	AND	Logisk AND
8.	OR	Logisk OR
9.	NOT	logisk NOT
9.	IS NULL	har værdien null

Eksempel:

1.	IF A = B	Hvis A er lig B ...
2.	IF A != B	Hvis A er forskellig fra B ...
3.	IF A < B	Hvis A er mindre end B ...
4.	IF A > B	Hvis A er større end B...
5.	IF A > B AND B > C	Hvis A er større end B OG B er større end C ...
6.	IF A > B OR B > C	Hvis A er større end B EL-LER B er større end C ...
7.	IF NOT A<B	Hvis A ikke er mindre end B ...
8.	IF A IS NULL	Hvis A har værdien NULL

IF-THEN-ELSIF

Denne konstruktion gør det muligt at opbygge mere komplekse sætninger.



Eksempel:

```
0.  DECLARE
1.     kredit customer.creditlimit%TYPE;
2.     gammel_kredit customer.creditlimit%TYPE;
3.     ordre_total ord.total%TYPE;
4.  BEGIN
5.     SELECT creditlimit INTO gammel_kredit
6.     FROM customer
7.     WHERE custid = 105;
8.     SELECT SUM(total) INTO ordre_total
9.     FROM ord
10.    WHERE custid = 105;
11.    IF ordre_total < 50000 THEN
12.       kredit := gammel_kredit * 0.90;
13.       UPDATE customer SET creditlimit = kredit
14.       WHERE custid = 105;
15.       INSERT INTO uddata VALUES('105:
              '||to_char(gammel_kredit)||'/'||to_char(kredit));
16.       COMMIT;
17.    ELSIF ordre_total >= 50000 THEN
18.       INSERT INTO uddata VALUES('Kredit uændret');
19.       COMMIT;
20.    END IF;
21. END;
```

Boolske variable og betingelser

Man kan med fordel anvende boolske variable til styring af betingelser. Fordelen er dog udelukkende, at koden bliver mere læseværdig. Metoden kan som hovedregel anvendes, når der i en betingelse skelnes mellem to mulige tilstande.

Når en boolsk variabel skal tildeles en værdi, gøres dette ved at skrive et logisk udtryk på højre side af tildelingstegnet:

```
<Boolsk variabel> := <Et logisk udtryk>;

SkalHaveBonus := Indtjening > 100000;

AltidFalsk := 1 < 1;

AltidSand := 1 < 2;
```

Eksempel:

I nedenstående kodeeksempel styres betingelserne v.h.a. en boolsk variabel.

```
0.   DECLARE
1.      kredit customer.creditlimit%TYPE;
2.      gammel_kredit number(9,2);
3.      ordre_total number(9,2);
4.      ordre_total_ok BOOLEAN;
5.   BEGIN
6.      SELECT creditlimit INTO gammel_kredit
7.      FROM customer
8.      WHERE custid = 105;
9.      SELECT sum(total) INTO ordre_total
10.     FROM ord
11.     WHERE custid = 105;
12.
13.     ordre_total_ok := ordre_total < 50000;
14.
15.     IF ordre_total_ok THEN
16.       kredit := gammel_kredit * 0.90;
17.       UPDATE customer SET creditlimit = kredit
18.       WHERE custid = 105;
19.       INSERT INTO uddata values('105:
              '||to_char(gammel_kredit)||'/'||to_char(kredit));
20.       COMMIT;
21.     ELSIF not ordre_total_ok THEN
22.       INSERT into uddata values('Kredit uændret');
23.       COMMIT;
24.     END IF;
25.  END;
26.  /
27.  SELECT * FROM uddata;
```

Her erklæres den boolske variabel.

Her tildeles den boolske variabel en værdi.

Her anvendes variablen til at styre betingelserne.

Opgave 5.1

Formål

At skrive et PL/SQL-program, der anvender betingelser.
Tabel: ORDRER, MEDARB, UDDATA

1. Skriv et PL/SQL-program, der gør følgende.

   - finder den samlede salg, som medarbejder 4 har haft ansvaret for.
   - hvis dette salg er større end 2 mill. kroner, så skal hans løn hæves med 10 procent. Lønstigningen skal skrives i tabellen medarb.
   - skriv den nye løn, den gamle løn og det samlede salg i tabellen uddata.
   - hvis medarbejderens samlede salg ikke er større end 2. mill. kroner, skal der blot skrives et notat i tabellen uddata, der fortæller at lønnen er uændret.

2 Kør programmet og undersøg om det virker korrekt.

Opgave 5.2

Formål

At skrive et PL/SQL-program, der anvender betingelser.
Tabel: ORDRER, UDDATA

1. Skriv et PL/SQL-program, der gør følgende.

   - finder den gennemsnitlige ordretotal, for firmaet med kundeid QUICK.
   - finder den gennemsnitlige ordretotal, for den samlede ordretabel.
   - herefter skal QUICKS ordre nummer 10865 opdateres efter følgende regler:

     - hvis QUICKS gennemsnitlige ordre er mellem 5000 og 10000 kroner større end det samlede gennemsnit, så reducér fragt for ordre 10865 med 20 %.
     - hvis QUICKS gennemsnitlige ordre er mellem 10000 og 15000 kroner større end det samlede gennemsnit, så reducér fragt for ordre 10865 med 40 %.
     - ellers reducér fragt for ordre 10865 med 60 %.

   - Skriv en passende besked i tabellen uddata, der indeholder den relevante information.

2 Kør programmet og undersøg om det virker korrekt.

Opgave 5.3

Formål

At skrive et PL/SQL-program, anvender boolske variable, til styring af betingelser.
Tabel: ORDRER, MEDARB, UDDATA

1. Omskriv programmet, der blev lavet i opgave 5.1., således at det anvender en boolsk variabel til styring af betingelsen.

2 Kør programmet og undersøg om det virker korrekt.

6. Løkker og kontrolstrukturer

En af de helt afgørende forskelle mellem SQL og PL/SQL løkkestrukturer. Mens man i SQL ikke behøver at tænke over hvordan en handling skal udføres - fordi man blot specificerer hvad der ønskes gjort - kan man i PL/SQL skrive programmer, med fuld kontrol over programmets forløb.

Løkkebegrebet findes i langt de fleste procedurale programmeringssprog og er også en integreret del af PL/SQL.

Løkker kan principielt beskrives som programstrukturer, der sikrer at en given mængde kodesætninger udføres indtil en betingelse er opfyldt. En løkkes gennemløb benævnes ofte med ordet **iteration** - man taler derfor om en løkkes **iterationer**.

Eksempel 1:

```
┌─→ Sålænge <en betingelse> udfør
│      ...
│      ...
│      ...
└── Hop til start
```

Eksempel 2:

```
┌─→ Udfør løkken
│      ...
│      ...
│      hvis <en betingelse> så gå til afslut
└── Hop til start
    afslut
```

Fælles for eksemplerne er at de ikke viser konkrete PL/SQL løkker, men derimod principielle opbygninger.

For begge eksempler gælder at kodelinierne - som er symboliseret med tre prikker - udføres det antal gange som løkken "kører". Hvis løkken kører 20 gange, udføres koden 20 gange.

I PL/SQL findes en lang række anvendelsesmuligheder for løkker og ved hjælp af såkaldte Cursors - som gennemgås senere i materialet - er der mulighed for at lave meget avancerede programmer.

Det simple loop

Det simpel loop er den enkleste struktur, som kan anvendes, når man ønsker at udføre et antal kodesætninger flere gange. Den generelle syntaks er følgende:

```
LOOP
   <sætninger som skal udføres i loop'et>
END LOOP;
```

Loopet har ikke en indbygget stopbetingelse og vil derfor fortsætte i det uendelige, medmindre man i sætningerne indeni loopet placerer en sætning, der standser løkken.

Eksempel:

Nedenstående program indsætter værdierne 1-10 i tabellen uddata.

```
1.  DECLARE
2.     counter NUMBER(5,0) ;
3.  BEGIN
4.     counter := 1;
5.     LOOP
6.       INSERT INTO uddata values(to_char(counter));
7.       counter := counter + 1;
8.       IF counter > 10 THEN
9.          COMMIT;
10.         EXIT ;
11.      END IF;
12.    END LOOP;
13. END;
14. /
15. SELECT * FROM uddata;
```

Tællervariabel tildeles en startværdi

Løkken starter.

Hvis tæller er mere end 10, så commit og stop løkken.

Løkken slutter.

Programmet svarer med følgende meddelelse:

```
PL/SQL procedure successfully completed.

TEKST
-----
1
2
3
4
5
6
7
8
9
10

10 rows selected.
```

Kommandoen Exit

Kommandoen EXIT afbryder et loop, således at programudførelsen genoptages i første linie efter sætningen END LOOP. EXIT kan kun anvendes indenfor loopets grænser - d.v.s. mellem sætningerne LOOP og END LOOP. Hvis der ikke findes en EXIT-kommando i et simpelt loop vil løkken ikke stoppe.

EXIT-kommandoen kan udbygges således at den anvender en betingelse. Den generelle syntaks er som følger:

```
EXIT WHEN <en betingelse>;
```

Eksempel 1:

```
EXIT WHEN counter > 20;

EXIT WHEN SidstePostFundet;
```

I ovenstående eksempler viser nederste linie hvordan en boolsk variabel kan anvendes til at afbryde en løkke.

Eksempel 2:

```
1.  DECLARE
2.     counter NUMBER(5,0);
3.     slut BOOLEAN;
4.  BEGIN
5.     counter := 1;
6.     LOOP
7.       INSERT INTO uddata values(to_char(counter));
8.        counter := counter + 1;
9.        slut := counter > 10;
10.       EXIT WHEN slut;
11.    END LOOP;
12.    COMMIT;
13. END;
14. /
```

Tip:

Generelt bør den indbyggede betingelse i EXIT-kommandoen, kun anvendes når man ønsker at fange en stopmulighed, der ligger udenfor det normale. Dette sikrer en mere læsebar kode, og øger funktionaliteten i de avancerede løkker. I det simple loop kan man dog anvende EXIT-kommandoens betingelse, uden disse overvejelser.

LABELS og GOTO-kommandoer

LABELS og GOTO-kommandoer er strukturer, der sikrer at man kan hoppe mellem særlige områder i koden, og de findes begge i langt de fleste procedurale programmeringssprog. Oftest anvendes kommandoerne sammen med IF-sætninger. Den generelle syntaks er som følger:

```
IF <en betingelse> THEN
  GOTO <label>;
END IF;
<<label>>
```

En LABEL er et sted i koden, som man ønsker programafviklingen skal fortsætte ved, når betingelsen bliver sand.

Eksempel:

I nedenstående eksempel indsættes værdierne 1-10 i tabellen uddata. Dette styres af et simpelt loop, som afsluttes v.h.a. en GOTO-kommando. Hvis betingelsen er sand hoppes til label SLUT.

```
1.   DECLARE
2.      counter NUMBER(5,0);
3.   BEGIN
4.      counter := 1;
5.      LOOP
6.         INSERT INTO uddata values(to_char(counter));
7.         counter := counter + 1;
8.         IF counter > 10 THEN
9.            GOTO slut;
10.        END IF;
11.     END LOOP;
12.     <<slut>>
13.     COMMIT;
14.  END;
15.  /
```

Bemærk:

SLUT skal skrives i dobbelte knækparenteser (<<>>). For en label gælder endvidere at den skal efterfølges af en gyldig sætning.

Tip:

Selvom GOTO-kommandoer er funktionelle og kan anvendes på mange måder, bør de bruges med en vis forsigtighed. Grunden hertil er at de slører strukturen i programmet, hvilket kan gøre vedligeholdelse og dokumentation vanskelig.

For loop

Et for-loop findes i langt de fleste programmeringssprog, og har mange fælles egenskaber med det simple loop. Den afgørende forskel er, at et man med et for-loop præcist kan angive hvor mange gange loopet skal udføres. Denne generelle syntaks er som følger:

```
FOR <tællervariabel> IN 1..<UDTRYK EL. KONSTANT>
LOOP
   ...
   ...
END LOOP;
```

Ideen er således at tællervariablen hele tiden holder styr på, hvor langt man er i rækken 1-<udtryk>. Hvis der står 1-20 får tællervariablen denne værdi, når løkken gennemløbes - i dette tilfælde - 20 gange.

Eksempel:

I nedenstående program indsættes værdierne 1 til 10 i tabellen uddata.

```
1.  BEGIN
2.     FOR i IN 1..10
3.     LOOP
4.        insert into uddata values(to_char(i));
5.     END LOOP;
6.     commit;
7.  END;
8.  /
```

Bemærk:

Tællevariablen, som i ovennævnte eksempel hedder i, behøver ikke at blive erklæret. Hvis dette ikke gøres, kan variablen dog ikke anvendes udenfor løkken. Løkkens grænse er i eksemplet styret af tallet 10 - d.v.s. at løkken gennemløbes 10 gange. Tallet 10 kunne have været erstattet af et udtryk.

Det er muligt at få løkken til at tælle baglæns. Dette gøres på følgende måde:

```
1.   BEGIN
2.      FOR i IN REVERSE 1..10
3.      LOOP
4.         insert into uddata values(to_char(i));
5.      END LOOP;
6.      commit;
7.   END;
8.   /
```

I dette tilfælde vil tællevariablen have værdien 10 ved første gennemløb, og værdien 1 ved sidste.

While loop

While-konstruktionen udføres sålænge en given betingelse er opfyldt. Hvis betingelsen ikke ændres efterhånden som iterationerne skrider frem, er løkken uendelig. Den generelle form er som følger:

```
WHILE <udtryk> <sammenligningsoperator> <udtryk> LOOP
   ...
   ...
END LOOP;
```

Oftest anvendes en tællevariabel til at styre antal gennemløb. Denne tællevariabel tælles op eller ned i begyndelsen eller slutningen af løkken.

Eksempel:

I nedenstående programeksempel indsættes værdierne 1-10 i tabellen UDDATA. Tælleren i styrer antal iterationer.

```
1.  DECLARE
2.     i NUMBER(3);
3.  BEGIN
4.     i := 1;
5.     WHILE i < 11 LOOP
6.        INSERT INTO uddata values(to_char(i));
7.        i := i + 1;
8.     END LOOP;
9.     COMMIT;
10. END;
11. /
```

Bemærk:

Det er muligt at afbryde While-løkken v.h.a. kommandoen EXIT. Dette bør kun gøre for at "fange" særlige situationer, der gør While-løkkens oprindelige stopbetingelse ugyldig. Brug af EXIT i While-løkker bør betragtes som brud på programmets naturlige struktur, og kan derfor sløre denne.

Eksempel:

I det følgende gives et eksempel på, hvordan et While-loop kan evalueres på en værdi, der hentes fra tabeller, i løkkens krop. Dermed opnås en funktionalitet, hvor værdierne i tabellen styrer antallet af iterationer. Programmet søger gennem EMP-tabellen og finder den medarbejder, hvis løn er større end 4500. Programmet søger ikke "hovedløst" gennem alle medarbejdere, men starter med den der tjener mindst, og tager derefter hele tiden den pågældende medarbejders chef. Undervejs udskrives den aktuelle medarbejder til tabellen uddata. Når while-løkken afsluttes, skrives den aktuelle person til tabellen uddata.

```
1.   DECLARE
2.       hyre            emp.sal%TYPE;
3.       min_hyre        emp.sal%TYPE;
4.       chef            emp.mgr%TYPE;
5.       efternavn       emp.ename%TYPE;
6.       nr              emp.empno%TYPE;
7.       start_person    emp.empno%TYPE;
8.       i               number(2);
9.
10.  BEGIN
11.      i := 1;
12.      SELECT min(sal) into min_hyre from emp;
13.      SELECT empno into start_person from emp
14.          WHERE sal=min_hyre;
15.      SELECT sal, mgr INTO hyre, chef FROM emp
16.          WHERE empno = start_person;
17.
18.      WHILE hyre < 4500 LOOP
19.        SELECT sal, mgr, ename, empno INTO hyre, chef, efternavn ,nr
20.        FROM emp
22.        WHERE empno = chef;
23.        INSERT INTO uddata VALUES(to_char(i)||'/'||to_char(nr));
24.        i := i +1;
25.      END LOOP;
26.
27.      INSERT INTO uddata VALUES (to_char(hyre)||'/'||efternavn);
28.      COMMIT;
29.  END;
30.  /
31.  SELECT * FROM uddata;
```

Programmet svarer med følgende meddelelse:

```
PL/SQL procedure successfully completed.

TEKST
------------------------------
1/7902
2/7566
3/7839
5000/KING
```

Opgave 6.1

Formål

At skrive et PL/SQL-program, der anvender løkker.
Tabel: UDDATA

1. Skriv et PL/SQL-program, der gør følgende.

   - indsætter værdierne 100-1000, med et spring på 100, i tabellen uddata, v.h.a. en simpel løkke.

2 Kør programmet og undersøg om det virker korrekt.

3. Modificér programmet så det i stedet anvender en FOR-LØKKE, og indsætter værdierne 100-200.

4 Kør programmet og undersøg om det virker korrekt.

5. Modificér programmet så det i stedet anvender en WHILE-LØKKE.

6. Kør programmet og undersøg om det virker korrekt

Opgave 6.2

Formål

At skrive et PL/SQL-program, der anvender simple løkker.
Tabel: MEDARB, UDDATA

1. For alle medarbejdere er der registreret den person de refererer til. Dette gælder dog ikke for direktøren. Der skal laves et PL/SQL, som viser "kommandovejen" fra piccoloen, som har nummer 11, og til direktøren. Programmet skal udskrive de involverede medarbejderes numre. Brug eventuelt følgende programskitse:

```
DECLARE
   denne_person medarb.ref%TYPE;
   næste_person medarb.ref%TYPE;
BEGIN
  denne_person := 11
  loop
     -- find næste_person som denne_person refererer til
     -- indsæt et notat om denne_person i uddata
     -- hvis næste_person er null så slut
     -- sæt næste_person lig med denne_person
   end loop
END;
```

2. Undersøg om programmet fungerer korrekt, og test det eventuelt med andre startpersoner.

3. Overvej hvordan andre løkkestrukturer kunne anvendes i stedet for den valgte.

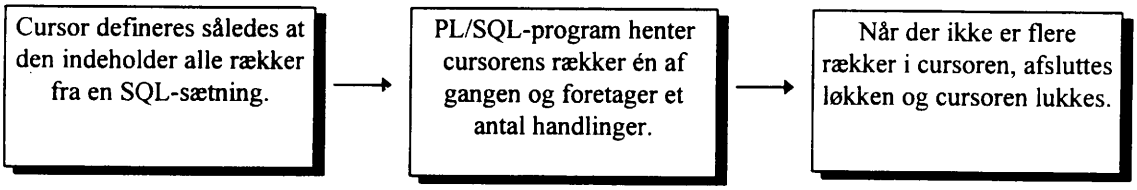
7. Cursors

Fælles for de SQL-sætninger, der har været anvendt hidtil er at de returnerer netop én række. Hvis en almindelig SQL-sætning - i et PL/SQL program - returnerer mere end én række opstår der en fejl. Dette skyldes at uddata fra SQL-sætningen indlæses i variable.

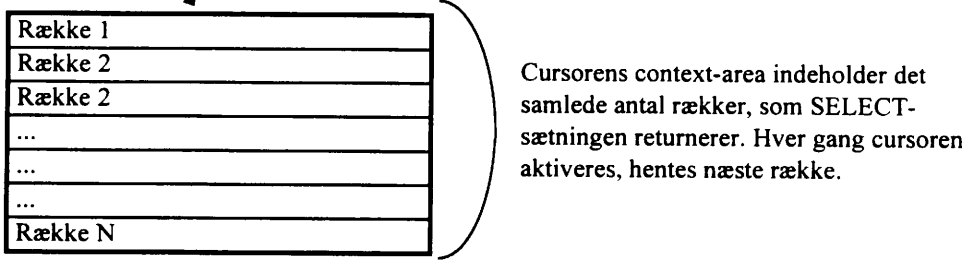
Oftest ønsker man dog at bruge en SQL-sætning til at hente et ukendt antal rækker - og man ønsker derefter at gøre et eller andet ved disse rækker. Denne funktionalitet indlejres derfor i en løkke, hvor hvert løkkegennemløb giver næste række. For at kunne opnå denne funktionalitet findes Cursors.

Cursors er navngivne områder i arbejdslageret - "context areas" - hvori man kan gemme og hente information.

Anvendelse af cursor:




```
SELECT ename, empno FROM emp;
```



Eksplicitte cursors

En eksplicit cursor, er en cursor, der erklæres og oprettes af brugeren af PL/SQL. Den eksplicitte cursor oprettes v.h.a. et SQL-udtryk, og returnerer de rækker som SQL'en returnerer.

Arbejdet med eksplicitte cursors omfatter fire forskellige områder:

Erklæring: I DECLARE-sektionen af PL/SQL-programmet erklæres cursoren. Cursoren skal dels have et navn, samt en SQL-sætning, der definerer dens rækker. Den generelle syntaks er:

```
CURSOR <CURSORNAVN> IS <SQL-SÆTNING>;
```

Åbning: Cursoren skal åbnes, således at dens rækker bliver tilgængelige. Dette gøres med en kommando OPEN. Den generelle syntaks er:

```
OPEN <CURSORNAVN>;
```

Hentning: Hver gang næste række skal hentes fra cursoren, skal der anvendes en særlig kommando FETCH. Den generelle syntaks er:

```
FETCH <CURSORNAVN> INTO <LISTE AF VARIABLE>;
```

Lukning: Når man ikke ønsker at arbejde mere med cursoren skal den lukkes. Dette gøres med kommandoen CLOSE. Ved lukning bliver det adresserum, som cursoren har anvendt frigivet. Ved en eventuel genåbning opdateres rækkerne i cursoren, således at eventuelle ændringer bliver medtaget. Den generelle syntaks er:

```
CLOSE <CURSORNAVN>;
```

Eksempel på anvendelse af cursor

I nedenstående program erklæres en cursor - ansatte - som rummer alle ansatte, med felterne empno, ename og salary. Programmet gennemløber en løkke, og reducerer alle lønninger med 10 %, hvorefter resultatet skrives i tabellen uddata.

```
1.   DECLARE
2.     nr   emp.empno%TYPE;
3.     navn emp.ename%TYPE;
4.     hyre emp.sal%TYPE;
5.     CURSOR ansatte IS SELECT empno, ename, sal FROM emp ORDER BY ename;
6.   BEGIN
7.     OPEN ansatte;
8.     LOOP
9.       FETCH ansatte INTO nr, navn, hyre;
10.      EXIT WHEN ansatte%NOTFOUND;
11.      hyre := hyre * 0.90;
12.      INSERT INTO uddata VALUES(to_char(nr)||'/'||navn||'/'||to_char(hyre));
13.    END LOOP;
14.    CLOSE ansatte;
15.    COMMIT;
16.  END;
17.  /
18.  SELECT * FROM uddata;
```

Programmet svarer med følgende meddelelse:

```
PL/SQL procedure successfully completed.

TEKST
---------------
7876/ADAMS/990
7499/ALLEN/1440
7698/BLAKE/2565
7782/CLARK/2205
7902/FORD/2700
7900/JAMES/855
7566/JONES/2677.5
7839/KING/4500
7654/MARTIN/1125
7934/MILLER/1170
7788/SCOTT/2700
7369/SMITH/720
7844/TURNER/1350
7521/WARD/1125
```

Cursor-attributter

En eksplicit cursor har fire forskellige attributter, der kan bruges til at undersøge cursorens tilstand. %NOTFOUND og %FOUND kan bruges til at afbryde løkker på passende tidspunkter. Hvis en FETCH får lov at fortætte efter at der ikke er flere rækker i søgesættet fremkommer følgende fejl:

```
ORA-06503: PL/SQL: Unhandled exception ORA-01002: fetch out of sequence
```

Derfor er det vigtigt at %NOTFOUND testes umiddelbart efter FETCH-kommandoen.

%ROWCOUNT kan bruges til at generere statistik o.lign. mens %ISOPEN primært anvendes ved avancerede problemstillinger.

%NOTFOUND Udtrykket <CURSORNAVN>%NOTFOUND får værdien sand, hvis seneste FETCH-kommando, ikke returnerede en række.

%FOUND Udtrykket <CURSORNAVN>%FOUND får værdien sand, hvis seneste FETCH-kommando returnerede en række.

%ROWCOUNT Udtrykket <CURSORNAVN>%ROWCOUNT returnerer antallet af rækker, der er modtaget af FETCH indtil nu.

%ISOPEN Udtrykket <CURSORNAVN>%ISOPEN får værdien sand, hvis cursoren er åben.

Opdatering af cursor-rækker

Ofte ønsker man at opdatere et antal rækker, som returneres af en cursor. For at dette kan lade sig gøre skal der anvendes en særlig syntaks i erklæringen og i selve opdateringen. I erklæringen skrives følgende:

```
CURSOR <Navn> is <SELECT-sætning>  FOR UPDATE OF <kolonner>;
```

```
CURSOR ansatte IS SELECT empno, job FROM emp FOR UPDATE OF sal;
```

FOR UPDATE sikrer at cursorens rækker låses eksklusivt når cursoren åbnes.

Ved opdateringen af rækken skrives:

```
<SQL-sætning> WHERE CURRENT OF <Cursor-navn>
```

```
UPDATE emp SET sal = sal * 1.15 WHERE current of ansatte;
```

WHERE CURRENT OF er en reference til den række cursoren peger på "lige nu".

Bemærk:

Det er ikke muligt at "comitte" midt i et cursor-gennemløb, hvis cursorens rækker opdateres.

%ROWTYPE-erklæringer

I arbejdet med cursors har man ofte brug for at kunne arbejde på hele rækker i en tabel. Man ønsker måske at slette rækker, opdatere flere felter eller lignende. Dette kan håndteres med en ROWTYPE-erklæring, der er en **sammensat datatype**. En sammensat datatype kendes i en række trediegenerationsprogrammeingssprog, for eksempel RECORDS i Pascal og STRUCTS i C.

ROWTYPE-objektet kan derfor rumme flere værdier, og stiller en syntaks til rådighed, der sikrer, at alle værdier kan læses individuelt. Den generelle syntaks er følgende:

```
<variabel-navn> <tabel eller cursor>%ROWTYPE;
```

Når man adresserer et felt gøre det med følgende syntaks:

```
<variabel-navn>.<felt-navn>;
```

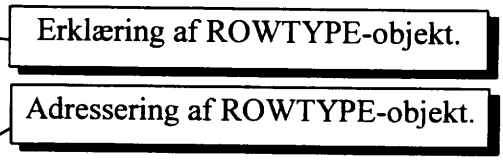
Når man fetcher gøres det med følgende syntaks:

```
fetch <cursornavn> into <rowtype-variabel>
```

Eksempel:

I nedenstående programeksempel anvendes ROWTYPE-erklæringer til at opbevare værdier for cursoren ansatte. Programmet gennemløber alle ansatte og for ansatte der tjener mere end 2000 og ikke er managers - hæves lønnen med 15 procent. En tællevariabel emp_count holder styr på hvor mange rækker der opdateres.

```
1.   DECLARE
2.      CURSOR ansatte IS SELECT empno, ename, sal, job FROM emp
3.             FOR UPDATE OF sal;
4.      ansat_rec ansatte%ROWTYPE;
5.      emp_count NUMBER := 0;
6.   BEGIN
7.     OPEN ansatte;
8.     LOOP
9.       FETCH ansatte INTO ansat_rec;
10.      EXIT WHEN ansatte%NOTFOUND;
11.      IF ansat_rec.sal > 2000 AND ansat_rec.job <> 'MANAGER' THEN
12.         UPDATE emp SET sal = sal * 1.15 WHERE CURRENT OF ansatte;
13.         emp_count := emp_count + 1;
14.      END IF;
15.    END LOOP;
16.    INSERT INTO uddata VALUES(to_char(emp_count));
17.    COMMIT;
18.  END;
```



Manipulation af %ROWTYPES

Det er muligt at tildele værdier mellem ROWTYPE-objekter med en enkelt sætning. Dette er muligt, hvis begge objekter har samme struktur.

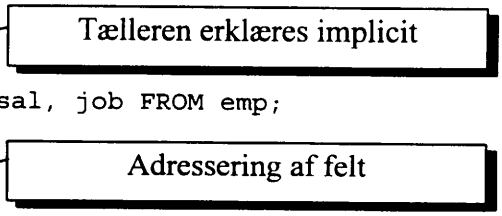
Eksempel:

```
1. DECLARE
2.    ansatte emp%ROWTYPE;
3.    denne_ansatte emp%ROWTYPE;
4. BEGIN
5.    ...
6.    denne_ansatte := ansatte;
7.    ...
8. END;
```

Cursor FOR-løkker

Det er muligt at anvende For-løkken, på en særlig måde i.f.m. cursors. Fordelen er at man ikke behøver at erklære tællere eller at åbne/lukke cursoren - dette gøres automatisk.

Eksempel:



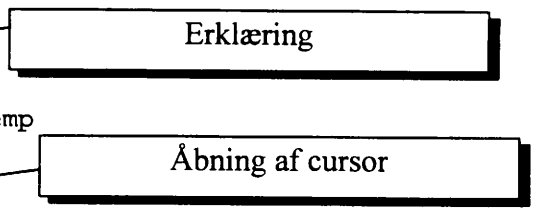

```
1.  DECLARE
2.     CURSOR ansatte IS SELECT empno, ename, sal, job FROM emp;
3.  BEGIN
4.    FOR rec in ansatte LOOP
5.      insert into uddata values(rec.ename);
6.      commit;
7.    END LOOP;
8.  END;
```

Cursors, der anvender parametre

Det er muligt at erklære sin cursor, således at der kan angives en parameter-værdi, når cursoren åbnes. Dette gøres på følgende måde:

```
CURSOR <Cursor-navn>  (<Parameter> <Type>) IS <SQL-Sætning>;
```

Eksempel:




```
1.  DECLARE
2.    cursor c1 (hyre NUMBER) IS
3.                  select ename, sal from emp
4.                  where sal > hyre;
5.  BEGIN
6.    FOR emps in c1 (3000)
7.    LOOP
8.       insert into uddata values (emps.ename );
9.       COMMIT;
10.   END LOOP;
11. END;
```

Cursors, der oprettes dynamisk

Det er muligt at oprette cursors, som ikke erklæres. Dette gøres ved at skrive et SQL-udtryk direkte i for-løkken:

```
FOR <tæller> IN (<SQL-Sætning>)
LOOP
...
END LOOP;
```

Eksempel:

```
1.  BEGIN
2.    FOR emps in (select ename from emp)
3.    LOOP
4.       insert into uddata values (emps.ename);
5.       COMMIT;
6.    END LOOP;
7.  END;
```

Implicitte cursors

Selv når man ikke eksplicit åbner en cursor, findes der en implicit cursor, som PL/SQL opretter. Denne cursor kan ikke åbnes eller lukkes, men det er alligevel muligt at læse dens attributter. Den implicitte cursor hedder altid SQL%, og har følgende attributter.

%NOTFOUND Udtrykket SQL%NOTFOUND får værdien sand, hvis INSERT, UPDATE eller DELETE ikke har påvirket/manipuleret en række. Det samme gælder ved en ENKELT-RÆKKE SELECT-sætning.

%FOUND SQL%FOUND får værdien sand, hvis INSERT, UPDATE eller DELETE ikke har påvirket/manipuleret en række. Det samme gælder hvis ENKELT-RÆKKE SELECT-sætning har returneret en eller flere rækker.

%ROWCOUNT Udtrykket SQL%ROWCOUNT returnerer antallet af rækker, der påvirkes af en INSERT, UPDATE eller DELETE. Ved en enkelt-række SQL returneres antallet af rækker.

%ISOPEN Udtrykket SQL%ISOPEN returnerer altid værdien FALSK, hvilket skyldes at den implicitte cursor altid lukkes automatisk umiddelbart efter at den er afviklet.

Eksempel:

I nedenstående programstump anvendes den implicitte cursor SQL%ROWCOUNT til at undersøge hvor mange rækker der slettes fra tabellen emp.

```
1.  DECLARE
2.     temp NUMBER;
3.  BEGIN
4.     DELETE FROM emp WHERE ename like 'M%';
5.     temp := SQL%ROWCOUNT;
6.     INSERT INTO uddata VALUES (to_char(temp));
7.     COMMIT;
8.  END;
```

Opgave 7.1

Formål

At skrive et PL/SQL-program, der anvender cursors.
Tabel: MEDARB, UDDATA

1. Skriv et PL/SQL-program, der gør følgende:

   - opretter en cursor, der indeholder felterne mednr, stilling og hyre fra tabellen medarb.
   - hæver lønnen med 15 %, for alle medarbejdere og skriver den enkelte medarbejders mednr, stilling og nye løn, til tabellen uddata. Programmet skal ikke opdatere andre tabeller.

2. Afprøv programmet og undersøg om det fungerer korrekt.

3. **Ekstra:**
   Udbyg programmet således at den samlede lønstigning akkumuleres i en variabel, der skrives til tabellen uddata, inden programmet slutning.

Opgave 7.2

Formål

At skrive et PL/SQL-program, der anvender cursors.
Tabel: ORDRER, UDDATA

1. Skriv et PL/SQL-program, der gør følgende.

   - opretter en cursor, som indeholder kundeid og total for alle ordrer, hvor totalen er større end 70000 kroner.
   - skriver kundeid og total for hver ordre til tabellen uddata.

2 Kør programmet og undersøg om det virker korrekt.

3. Udbyg programmet så det kun tager de første tre ordrer, der opfylder betingelserne.

Opgave 7.3

Formål

At skrive et PL/SQL-program, der anvender cursors.
Tabel: ORDRER, MEDARB, UDDATA

På grund af fejl i data findes der et antal poster i tabellen ordrer, hvor det tilhørende firma ikke findes i tabellen kunder. Derfor skal der laves et program, som finder disse ordrer, og opretter poster i kundetabellen.

1. Skriv et PL/SQL-program, der gør følgende.

   - opretter en cursor, som finder kundeid for de firmaer, der har ordrer, men som ikke findes i kundetabellen.
   - opretter poster i kundetabellen for hvert firma. De nye poster skal naturligvis indeholde kundeid, men kan i sagens natur ikke indeholde firmaoplysninger, kontaktperson o.s.v. Derfor indsættes en markering, der sikrer at de nye poster kan findes igen - f.eks. (Kundeid,'Ukendt firma',Ukendt kontaktperson',Ukendt stilling',Ukendt land').

2. Kør programmet og undersøg om det virker korrekt. Vis du har brug for at teste programmet flere gange, kan dette gøres ved at slette en eller flere tilfældig rækker fra tabellen kunder.

Opgave 7.4

Formål

At skrive et PL/SQL-program, der anvender rowtypes og cursors til opdatering af felter.
Tabel: MEDARB, UDDATA

Da firmaet har klaret sig godt, skal der gives en generel lønstigning. Der er afsat en pulje på 30000 kroner, som skal fordeles. Fordelingen er konstrueret således at man starter med den person, der tjener mindst og giver ham 10 % lønstigning. Denne lønstigning "spiser" en del af den samlede pulje. Dernæst gives en stigning til den person, der tjener næstmindst - o.s.v. Sådan bliver man ved indtil, der ikke er flere medarbejdere eller lønpuljen er brugt op.

Dette kan gøres efter følgende algoritme:

```
lønpuljen := 30000;
loop
   find den næste medarbejder;
   afslut, hvis der ikke er flere medarbejdere;
   afslut hvis lønpuljen er så lille, at der ikke kan gives en 10 % løn-
   stigning;
   opdatér medarbejderens løn;
   beregn, hvor mange kroner, der er tilbage i lønpuljen;
   skriv passende information til tabellen uddata;
end loop;
```

1. Skriv et PL/SQL, der foretager fordeling af lønpuljen - prøv at bruge rowtypeerklæring. Husk at programmet skal skrive passende information til tabellen uddata.

2. Test programmet og undersøg om det fungerer korrekt.

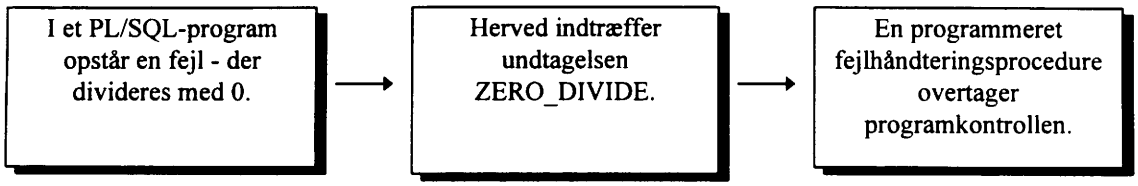
3. **Ekstra:**
   Udbyg programmet således at den procentvise lønstigning er højest for de medarbejdere, der tjener mindst.

8. Fejlhåndtering

Hvad er fejlhåndtering ?

Et PL/SQL-program udfører ikke altid sine opgaver som forventet. Programmeringsfejl, systemfejl o.a. er årsager til forskellige fejlsituationer, der kan opstå mens programmet kører - såkaldte runtimefejl. Der findes i PL/SQL en række forskellige måder, hvorpå fejlsituationer kan fanges og håndteres af systemet.

I PL/SQL findes begrebet EXCEPTIONS (undtagelser), som er den hændelse, der indtræffer når en fejl opstår. Derfor taler man om fejlhåndtering som EXCEPTION-handling (håndtering af undtagelser).



Man skelner mellem to typer fejl:

- **Interne exeptions**

Et antal foruddefinerede fejlsituationer, f.eks. division med 0.

- **brugerdefinerede exeptions**

Brugerdefinerede fejlsituationer, hvor brugeren selv afgør hvornår fejlen er indtruffet.

Fordelene ved fejlhåndtering er mange. Først og fremmest bliver programmerne mere stabile og sikre. Derudover kan en fornuftig fejlhåndteringsstrategi sikre, at man ikke behøver at skrive den samme kode mange gange.

Eksempel:

I et skitseret program skal der udføres tre SELECT-sætninger. For hver af de tre sætninger skal der undersøges, hvorvidt der returneres 0 rækker. Dette kunne gøres på nedenstående måde. Pointen er, at der skrives en fejlhåndteringskommando for hver situation, hvor fejlen kan opstå, og det er uhensigtsmæssigt.

```
DECLARE
...
BEGIN
  SELECT ...
   /* her checkes for fejl */
  SELECT ...
   /* her checkes for fejl */
  SELECT ...
   /* her checkes for fejl */
END;
```

Hvis man anvender exception-handling kommer programskitsen til at se ud som følger:

```
DECLARE
...
BEGIN
  SELECT ...
  SELECT ...
  SELECT ...
EXCEPTION
  WHEN NO_DATA_FOUND THEN ...
END;
```

Fordelen ved dette program er at der kun findes én fejlhåndtering, som er placeret under programsektionen exception. Når den interne hændelse NO_DATA_FOUND indtræffer, udføres programsætningerne i denne sektion, uanset hvor hændelsen indtræffer. Således får et program, der anvender exception-handling følgende generelle struktur:

```
DECLARE
   <erklæringer af variable, konstanter og cursors>
BEGIN
   <PL/SQL-sætninger>
EXCEPTION
   <specifikationer af hvad der skal ske når bestemte hændelser
     indtræffer>
END;
```

Generel syntaks

Den generelle syntaks for exception-handling er følgende:

```
EXCEPTION
   WHEN <exception-navn> THEN <PL/SQL-sætninger>;
   WHEN <exception-navn> OR <exception-navn> THEN <PL/SQL-sætninger>;
   ...
   WHEN OTHERS THEN <PL/SQL-sætninger>;
```

System exceptions

System-exceptions er de hændelser der automatisk indtræffer i forskellige situationer. Nedenstående liste viser de mest almindelige:

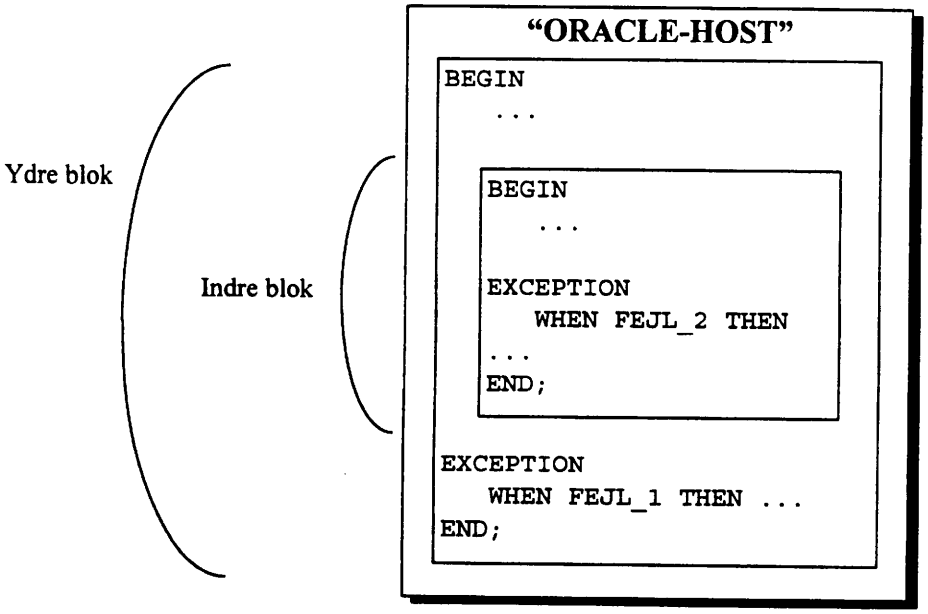
PL/SQL Exception	Returnerer	Fejlkode	Exception indtræffer ved
DUP_VAL_ON_INDEX	-1	ORA-00001	INSERT/UPDATE tried to create two rows w/ dup values in UNIQUE cols
INVALID_CURSOR	-1001	ORA-01001	Stmt specified an invalid cursor
INVALID_NUMBER	-1722	ORA-01722	CHAR to NUMBER failed: not number
LOGIN_DENIED	-1017	ORA-01017	Invalid username/password used
NO_DATA_FOUND	+100	ORA-01403	SELECT returned zero rows
NOT_LOGGED_ON	-1012	ORA-01012	ORACLE call without ORACLE logon
PROGRAM_ERROR	-6501	ORA-06501	PL/SQL had an internal problem
STORAGE_ERROR	-6500	ORA-06500	PL/SQL memory full or corrupted
TIMEOUT_ON_RESOURCE	-51	ORA-00051	Resource not available in time
TOO_MANY_ROWS	-1427	ORA-01427	SELECT returned more than one row
VALUE_ERROR	-6502	ORA-06502	Arith/num/string/conversion error
ZERO_DIVIDE	-1476	ORA-01476	Attempted to divide a number by 0

Fejlkoder

Som vist i ovenstående tabel, er der til en givent exception tilknyttet en exception-navn, returværdi og en fejlkode. Alle returværdier er negative tal, undtagen NO_DATA_FOUND. Fejlkoderme af formen ORA-<TAL> er Oracles almindelige fejlkoder.

Virkefelter for exceptions

Exceptions kan optræde på alle blokke i et PL/SQL-program. Det betyder at det er muligt at have exceptions, der kun gælder for en bestemt blok.



På figuren overfor ses et skitseret program, der rummer to blokke. I hver blok findes en exception. Hvis FEJL_2 opstår i indre blok, udføres den tilknyttede exception, og programafviklingen genoptages i omgivende blokninveau - i dette tilfælde den yderste blok. Hvis FEJL_1 var opstået på indre niveau, som ikke rummer exception for denne fejl, ville ydre bloks exception være blevet udført og programmet ville overgive kontrol til hosten. Hvis der opstår en fejl, som ikke håndteres af en exception overgives kontrollen til hosten og en fejlmeddelelse skrives på skærmen.

Eksempel:

I nedenstående programeksempler undersøges en tabel, og alle rækker i tabellen behandles v.h.a. en cursor-for løkke. Hvis der opstår en fejl med én af rækkerne ønsker vi at registrere fejlen, men i øvrigt at fortsætte programafviklingen for eventuelt resterende rækker. Derfor placeres den tilhørende exception i en blok, der er placeret inden i for-løkken. Dette vil nemlig bevirke, at programmet efter udførelse af den tilhørende exception, vil give kontrol til omkringliggende blok - i dette tilfælde for-løkken.

Første eksempel viser programmet uden fejlhåndtering - andet eksempel viser det med fejlhåndtering. Konkret gennemløbes emp-tabellen og der foretages en beregning. Beregningen består i at man beregner hvor mange ansatte der kunne være i organisationen for hver medarbejders løntrin.

Eksempel 1 - ingen fejlhåndtering:

```
1.    DECLARE
2.       cursor ansatte IS SELECT ename, sal, job FROM emp;
3.       hyre_total emp.sal%TYPE;
4.       hyre_andel NUMBER(4,2);
5.    BEGIN
6.       SELECT sum(sal) INTO hyre_total FROM emp;
7.       FOR ansat IN ansatte
8.       LOOP
9.          hyre_andel := hyre_total / ansat.sal;
10.         INSERT INTO uddata VA  LUES(ansat.ename||'/'||to_char(hyre_andel));
11.         COMMIT;
12.      END LOOP;
14.   END;
```

Hvis der forekommer et 0 i feltet sal i en af rækkerne, gives følgende besked:

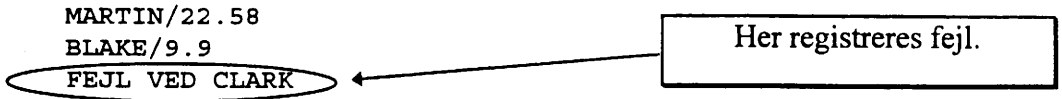
```
ERROR at line 8:
ORA-06503: PL/SQL: error 0
Unhandled exception ORA-01476:
divisor is equal to zero which was raised in a statement starting at line 9
ORA-06503: PL/SQL: error 995 - unhandled exception # -1476
```

Eksempel 2 - med fejlhåndtering:

```
1.    DECLARE
2.       CURSOR ansatte IS SELECT ename, sal, job FROM emp;
3.       hyre_total emp.sal%TYPE;
4.       hyre_andel NUMBER(4,2);
5.    BEGIN
6.       SELECT sum(sal) INTO hyre_total FROM emp;
7.       FOR ansat IN ansatte
8.       LOOP
9.         BEGIN
10           hyre_andel := hyre_total / ansat.sal;
11.          INSERT INTO uddata VALUES(ansat.ename||'/'||to_char(hyre_andel));
12.          COMMIT;
13.        EXCEPTION
14.          WHEN zero_divide THEN
15.             INSERT  INTO uddata  VALUES('FEJL VED '||ansat.ename);
16.          COMMIT;
17.        END;
18.      END LOOP;
19.   END;
```

Programmet svarer ved kørsel med fejl på følgende måde:

```
JONES/9.49
MARTIN/22.58
BLAKE/9.9
FEJL VED CLARK
SCOTT/8.18
KING/4.91
TURNER/18.82
ADAMS/25.66
JAMES/29.71
FORD/8.18
MILLER/21.71
SMITH/35.28
ALLEN/17.64
WARD/22.58
```



Bemærk at fejlen opstår allerede efter 3. række - og at programmet alligevel forstætter.

Brugerdefinerede exceptions

Det er muligt at definere egne exceptions. Fordelen herved er at man kan generalisere sin kode - fordi et givent fejlcheck kun behøver at være placeret ét sted. Den generelle syntaks er følgende:

```
DECLARE
   <exception-navn> EXCEPTION;
BEGIN
   ...
   IF <BETINGELSE FOR EXCEPTION> THEN
       RAISE <EXCEPTION-NAVN>;
   ELSE
      ...
   END IF;
EXCEPTION
   WHEN <EXCEPTION-NAVN> THEN <PL/SQL-SÆSTNINGER>;
END;
```

Eksempel:

Følgende eksempel viser brugen af brugerdefinerede exceptions. I tabellen DEPT findes en registrering af firmaets afdelinger. Tabellen ser ud som følger:

```
    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
```

I forbindelse med datavalidering ønsker vi at undersøge om springet mellem afdelingsnumrene er præcis 10. Hvis dette ikke er tilfældet skal der skrive en fejlmeddelelse til tabellen uddata.

```
1.   DECLARE
2.      fejl_i_id EXCEPTION;
3.      CURSOR afd IS SELECT deptno, dname, loc FROM dept;
4.      forrige_id dept.deptno%TYPE;
5.      fejl_id dept.deptno%TYPE;
6.   BEGIN
7.     forrige_id := 0;
8.     FOR denne_afd IN afd
9.     LOOP
10       IF forrige_id + 10 <> denne_afd.deptno
11.      THEN
12.        fejl_id := denne_afd.deptno;
13.        RAISE fejl_i_id;
14.      ELSE
15.        INSERT INTO uddata VALUES('AFD '||to_char(denne_afd.deptno)||' - OK');
16.        forrige_id := denne_afd.deptno;
17.        COMMIT;
18.    END IF;
19.    END LOOP;
20.  EXCEPTION
21.     WHEN fejl_i_id THEN
22.       INSERT INTO uddata VALUES('Fejl ved afd. '||to_char(fejl_id));
23.  END;
```

Hvis fejl så RAISE exception

EXCEPTION-kode

Ved kørsel uden fejl svarer programmet med følgende:

```
TEKST
------------------------------
AFD 30 - OK
AFD 40 - OK
AFD 10 - OK
AFD 20 - OK
```

Antag at tabellen ser ud som følger:

```
    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        32 SALES          CHICAGO
        40 OPERATIONS     BOSTON
```

Så vil programmet svare med følgende indhold af tabellen uddata:

```
TEKST
------------------------------
Fejl ved afd. 32
AFD 10 - OK
AFD 20 - OK
```

Avanceret fejlhåndtering

Når en fejl opstår - og denne fejl ikke fanges af en exception - skrives der en meddelelse på skærmen. I det følgende gennemgås, hvordan det er muligt at skrive denne besked til en tabel. Oftest anvender man metoden til at beskrive fejl, der ikke fanges af én exception-handler - se nedenstående programskitse:

```
DECLARE
...
BEGIN
   ...
   EXCEPTION
   ...
   WHEN OTHERS THEN <SKRIV BESKED I UDDATA-TABEL>
END;
```

Det handler om at skrive de fejl, der ikke fanges af de foruddefinerede handlers. For at kunne opnå denne funktionalitet, skal der anvendes to funktioner:

SQLCODE Returnerer nummeret på den seneste fejl. Alle systemfejlnumre er negative undtagen fejl +100, som betyder NO_DATA_FOUND. Brugerdefinerede fejl er positive.

SQLERRM Returnerer beskrivelsen af fejlen - altså en tekststreng.

Eksempel:

Nedenstående program gennemløber tabellen emp, og skriver værdier til tabellen uddata. Hvis NO_DATA_FOUND exception indtræffer fanges fejlen - i alle andre tilfælde skrives fejlnummer og -tekst i tabellen.

```
1.    DECLARE
2.       CURSOR ansatte IS SELECT ename, sal, job FROM emp;
3.       hyre_total emp.sal%TYPE;
4.       hyre_andel NUMBER(4,2);
5.       fejl_nummer NUMBER;
6.       fejl_tekst  CHAR(100);
7.    BEGIN
8.       SELECT sum(sal) INTO hyre_total FROM emp;
9.       FOR ansat IN ansatte
10       LOOP
11.         BEGIN
12.           hyre_andel := hyre_total / ansat.sal;
13.           INSERT INTO uddata VALUES(ansat.ename||'/'||to_char(hyre_andel));
14.           COMMIT;
15.         EXCEPTION
16.         WHEN no_data_found THEN
17.            INSERT INTO uddata  values('INGEN DATA ');
18.            COMMIT;
19.         WHEN OTHERS THEN
20.            fejl_nummer := SQLCODE;
21.            fejl_tekst := substr(SQLERRM,1,100);
22.            INSERT INTO uddata VALUES('FEJL: '||to_char(fejl_nummer)||'
                     '||fejl_tekst);
23.            COMMIT;
24.         END;
25.      END LOOP;
26.   END;
```

Programmet svarer - ved fejl - med følgende indhold af tabellen uddata:

```
WARD/22.58
JONES/9.49
MARTIN/22.58
BLAKE/9.9
FEJL: -1476 ORA-01476: divisor is equal to zero
SCOTT/8.18
KING/4.91
TURNER/18.82
ADAMS/25.66
JAMES/29.71
FORD/8.18
MILLER/21.71
SMITH/35.28
ALLEN/17.64
```

Generering af fejltabel

Det er muligt at kalde funktionen SQLERRM med et parameter, som er et fejl-nummer, og på denne måde få returneret den aktuelle beskrivelse. Dette gøres på følgende måde:

```
DECLARE
   msg CHAR(200);
BEGIN
   FOR num IN 1000..1010
   LOOP
      msg := SQLERRM(-num);
      INSERT INTO uddata VALUES (msg);
   END LOOP;
END;


SQL> select * from uddata;

TEKST
--------------------------------------------------------------------------------

ORA-01001: invalid cursor
ORA-01002: fetch out of sequence
ORA-01003: no statement parsed
ORA-01004: default username feature not supported; logon denied
ORA-01005: null password given; logon denied
ORA-01006: bind variable does not exist
ORA-01007: variable not in select list
ORA-01008: not all variables bound
ORA-01009: missing mandatory parameter
ORA-01010: invalid OCI operation
ORA-01000: maximum open cursors exceeded
```

Opgave 8.1

Formål

At skrive et PL/SQL-program, der anvender fejlhåndtering.
Tabel: ORDRER, UDDATA

1. Skriv et PL/SQL-program, der gør følgende.

   - opretter en cursor, der indeholder kundeid, ordrenr, total og fragt for alle ordrer, der kommer fra firmaet ALFKI.
   - beregner for hver ordre fragtens andel af totalen i procent og skriver ordrenr og procentsatsen til tabellen uddata.

2. Kør programmet og se hvad der sker.

3. Udbyg programmet således at ZERO_DIVIDE-fejl opfanges, ved at der skrives en passende besked i tabellen uddata og at programmet derefter terminerer.

4. Kør programmet og se hvad der sker.

5. Udbyg programmet således afviklingen fortsætter efter en ZERO_DIVIDE-fejl - og at programmet fortæller hvilken ordre, der forårsagede fejlen.

6. Kør programmet og se hvad der sker.

Opgave 8.2

Formål

At skrive et PL/SQL-program, der anvender brugerdefineret fejlhåndtering.
Tabel: ORDRER, UDDATA

I tabellen ordrer er alle ordrer nummereret fortløbende, startende med nummer 10000. Der skal laves et program, som gennemløber tabellen og stopper med en besked i tabellen uddata, hvis der findes en ordre, som ikke er nummereret korrekt. Eksempelvis vil talrækken 1,2,3,5,6 give fejl ved 5, fordi 3+1 ikke er 5. Ved opgaveløsningen kan følgende algoritme anvendes:

```
forrige_ordre := 9999;
loop
   fetch næste_ordre;
   exit when ...
   hvis næste_ordre <> forrige_ordre + 1 så
       ordre_med_fejl := næste_ordre;
       raise brugerfejl;
   end if;
   forrige_ordre := næste_ordre;
end loop
EXCEPTION
  when brugerfejl then ...
```

1. Skriv et PL/SQL-program, der gør følgende.

   - opretter en cursor, der indeholder ordrenr fra tabellen ordrer.
   - sætter en variabel lig 9999 - fordi dette tal er startpositionen.
   - gennemløber en løkke, og for hvert gennemløb undersøger om ordren er præcis en værdi større end forrige. Hvis dette ikke er tilfældet, skal en brugerdefineret exception aktiveres, og programmet skal terminere.

2. Kør programmet og undersøg om det fungerer korrekt.

Opgave 8.3

Formål

At skrive et PL/SQL-program, der anvender avanceret fejlhåndtering.
Tabel: UDDATA

1. Skriv et PL/SQL-program, der skriver oracle-fejl 2000-2010 til tabellen uddata.

2. Kør programmet og undersøg om det fungerer korrekt.

Dansk Data Elektronik A/S

	Initials	Page
PL/SQL - kursus	Date 6/11-95	Project

Ora7 triggers, procedures for validering skrives i PL/SQL
ProC (indlejret SQL i C) precompiler

```
DECLARE
        hyre NUMBER(5)          } flere erklæringer
BEGIN
        SELECT sal INTO hyre
        FROM emp
        WHERE sal = 1600;
        hyre := hyre * 1.15;
        INSERT INTO uddata VALUES (to_char(hyre));
        COMMIT;
END
/
select * from uddata
delete from uddata where 1=1;
commit
```

Variable type : VAR CHAR, , BOOLEAN
↑ findes ikke i

Variable navn : højst 30 tegn, det første skal være char

Konstater : <konst.navn> CONSTANT <type> := <værdi>;

desc emp
└ SQL-PLUS kommando

ROWTYPE = objekt, record

CURSOR = pegepind til række i tabel

```
DECLARE
    CURSOR ansatte IS SELECT empno, ename, sal FROM emp ORDER BY ename
BEGIN
    OPEN ansatte
    LOOP
        FETCH ansatte INTO nr, navn, lyre
        EXIT WHEN ansatte%NOTFOUND

        INSERT INTO
    END LOOP
    CLOSE ansatte
```

CREATE TRIGGER test
BEFORE update
ON emp

[PL/SQL

: OLD. ENAME
: NEW. ENAME

```
-- OPGAVE 2.1

DECLARE
   navn CHAR(10);

BEGIN
  select fornavn into navn from medarb where mednr = 1;
  insert into uddata values(navn);
END;
/
SELECT * FROM UDDATA;
DELETE FROM UDDATA WHERE 1=1;
COMMIT;
```

```
-- OPGAVE 2.2

DECLARE
   denne_hyre number(8,2);

BEGIN
  select hyre into denne_hyre from medarb where mednr = 4;
  denne_hyre := denne_hyre * 0.9;
  insert into uddata values (denne_hyre);
END;
/
select * from uddata;
delete from uddata where 1=1;
commit;
```

```
-- OPGAVE 2.3
/* Programmet indlæser medarbejder 4's løn i en variabel, denne_hyre, og   gr

DECLARE
   denne_hyre number(8,2);

BEGIN
  select hyre into denne_hyre from medarb where mednr = 4;
  denne_hyre := denne_hyre * 0.9;
  insert into uddata values (denne_hyre);
END;
/
select * from uddata;
delete from uddata where 1=1;
commit;
```

```
-- OPGAVE 3.1

DECLARE
  gammel_hyre NUMBER(8,2);
  ny_hyre NUMBER(8,2);

BEGIN
  select hyre into gammel_hyre from medarb where mednr = 2;
  ny_hyre := gammel_hyre * 1.15;
  update medarb set hyre = ny_hyre where mednr = 2;
  insert into uddata values (to_char(gammel_hyre)||' '||to_char(ny_hyre)
END;
/
select * from uddata;
delete from uddata where 1=1;
commit;
```

```
-- OPGAVE 3.2

DECLARE
  id CHAR(5);
  navn CHAR(40);

BEGIN
  select kundeid, firma into  id, navn from kunder where kundeid = 'LETS
  delete from kunder where kundeid = 'LETSS';
  insert into uddata values (id||' * '||navn||' * '||to_char(sysdate));
END;
/
select * from uddata;
delete from uddata where 1=1;
```

```
-- OPGAVE 3.3

DECLARE
   dage NUMBER(4);
   st CHAR(30);
   navn CHAR(20);
   a DATE;

BEGIN
  select adato, stilling, efternavn into a, st, navn from medarb
    where mednr = 10;
  dage := sysdate - a;
  insert into uddata values (st||' '||navn||' har arbejdet i '||
         to_char(dage)||' dage.');

END;
/
select * from uddata;
delete from uddata where 1=1;
commit;
```

```
-- OPGAVE 4.1

DECLARE
   denne_dag DATE;
   tot number;
   fra  number;
   ialt number(8,2);
begin
   select total, fragt into tot, fra from ordrer where ordrenr = 10050;
   ialt := (tot + fra)*1.25;
   insert into uddata values ('10050 '||to_char(ialt));
   commit;
end;
/
select * from uddata;
delete from uddata where 1=1;
commit;
```

```
-- OPGAVE 4.2

DECLARE
  snit_usa ordrer.total%TYPE;
  snit_fra ordrer.total%TYPE;
  forskel ordrer.total%TYPE;
  procent NUMBER(5,2);

BEGIN
  select avg(total) into snit_usa from ordrer where kundeid in
    (select kundeid from kunder where land = 'USA');
  select avg(total) into snit_fra from ordrer where kundeid in
    (select kundeid from kunder where land = 'Frankrig');
  forskel := snit_usa - snit_fra;
  procent := (snit_fra/snit_usa) * 100;
  insert into uddata values ('forskellen er '||to_char(forskel));
  insert into uddata values ('forskel i procent '||to_char(procent));
END;
/
select * from uddata;
delete from uddata where 1=1;
commit;
```

```
-- OPGAVE 4.3

DECLARE
  mindst DATE;
  maks   DATE;
  forskel NUMBER(3);
BEGIN
  select min(adato) into mindst from medarb;
  select max(adato) into maks from medarb;
  forskel := months_between(maks,mindst);
  insert into uddata values ('Forskellen er '||to_char(forskel)||' måned   );
END;
/
select * from uddata;
delete from uddata where 1=1;
commit;
```

```
-- OPGAVE 4.4

DECLARE
  mtotal ordrer.total%TYPE;
  kunde ordrer.kundeid%TYPE;
  firma_n kunder.firma%TYPE;
  land_n kunder.land%TYPE;
  lande_t ordrer.total%TYPE;

BEGIN
  select max(total) into mtotal from ordrer;
  select kundeid into kunde from ordrer where total = mtotal;
  select firma, land into firma_n, land_n from kunder where
    kundeid = kunde;
  insert into uddata values('Største ordre afgivet af '||firma_n);
  select sum(total) into lande_t from ordrer where kundeid in (
    select kundeid from kunder where land = land_n);
  insert into uddata values ('Lande-total '||to_char(lande_t));
END;
/
select * from uddata;
delete from uddata where 1=1;
commit;
```

```
-- OPGAVE 5.1

DECLARE
   samlet_salg NUMBER(10,2);
   g_hyre medarb.hyre%TYPE;
   n_hyre medarb.hyre%TYPE;

BEGIN
   select sum(total) into samlet_salg from ordrer where mednr = '4';
   if samlet_salg > 2000000 then
      select hyre into g_hyre from medarb where mednr = '4';
      n_hyre := g_hyre * 1.1;
      insert into uddata
       values('Gam: '||to_char(g_hyre)||' Ny: '||to_char(n_hyre));
    else
       insert into uddata values('HYRE IKKE ÆNDRET');
    end if;
END;
/
select * from uddata;
delete from uddata where 1=1;
commit;
```

```
-- Opgave 5-2

DECLARE
   gsnit ordrer.total%TYPE;
   gsnit_alle ordrer.total%TYPE;
   forskel ordrer.total%TYPE;

BEGIN
  select  avg(total) into gsnit from ordrer where kundeid = 'QUICK';
  select avg(total) into gsnit_alle from ordrer;
  forskel := gsnit-gsnit_alle;

  IF forskel < 10000 AND forskel >= 5000 then
    insert into uddata values('5-10000');
    update ordrer set fragt = fragt * 0.8 where ordrenr = 10865;
    commit;
  ELSIF forskel < 15000 AND forskel >= 10000 then
    insert into uddata values('10-15000');
    update ordrer set fragt = fragt * 0.6 where ordrenr = 10865;
    commit;
  ELSE
    insert into uddata values('OVER 15000');
    update ordrer set fragt = fragt * 0.4 where ordrenr = 10865;
    commit;
  END IF;
END;
/
select * from uddata;
delete from uddata where 1=1;
```

```
-- OPGAVE 5.3

DECLARE
   samlet_salg NUMBER(10,2);
   g_hyre medarb.hyre%TYPE;
   n_hyre medarb.hyre%TYPE;
   salg_ok BOOLEAN;

BEGIN
   select sum(total) into samlet_salg from ordrer where mednr = '4';
   salg_ok := samlet_salg > 2000000;
   if salg_ok then
      select hyre into g_hyre from medarb where mednr = '4';
      n_hyre := g_hyre * 1.1;
      insert into uddata
       values('Gam: '||to_char(g_hyre)||' Ny: '||to_char(n_hyre));
    else
       insert into uddata values('HYRE IKKE ÆNDRET');
    end if;
END;
/
select * from uddata;
delete from uddata where 1=1;
commit;
```

```
-- OPGAVE 6.1

DECLARE
   counter NUMBER(5);
BEGIN
   counter := 100;

   LOOP
     exit when counter > 1000;
     insert into uddata values(to_char(counter));
     counter := counter + 100;
     commit;
   END LOOP;
END;
/
select * from uddata;
delete from uddata where 1=1;
commit;
```

```
-- OPGAVE 6.1 - 2

BEGIN

   FOR counter IN 100..200
   LOOP
     insert into uddata values(to_char(counter));
     commit;
   END LOOP;
END;
/
select * from uddata;
delete from uddata where 1=1;
commit;
```

```
-- OPGAVE 6.1 - 3

DECLARE
   counter NUMBER(5);
BEGIN
   counter := 100;

  while counter <= 1000
  LOOP
     insert into uddata values(to_char(counter));
     counter := counter + 100;
     commit;
   END LOOP;
END;
/
select * from uddata;
delete from uddata where 1=1;
commit;
```

```
-- OPGAVE 6.2

DECLARE
  denne_ref medarb.ref%TYPE;
  next_ref medarb.ref%TYPE;
BEGIN

  denne_ref:=11;
  loop
    select ref into next_ref from medarb where mednr = denne_ref;
    insert into uddata values (to_char(denne_ref));
    if next_ref is NULL then
       exit;
    end if;
    denne_ref := next_ref;
 end loop;




END;
/
select * from uddata order by tekst;
delete from uddata where 1=1;
commit;
```

```
-- OPGAVE 6.3

DECLARE
  start_p ordrer.ordrenr%TYPE;
  slut_p ordrer.ordrenr%TYPE;
  denne_ordre ordrer.ordrenr%TYPE;
  total_t ordrer.total%TYPE;
  kundeid_t ordrer.kundeid%type;

BEGIN
  select min(ordrenr) into start_p from ordrer;
  slut_p := start_p + 10;
  denne_ordre := start_p;
  LOOP
    exit when denne_ordre >  slut_p;
    select total, kundeid into total_t, kundeid_t from ordrer
      where ordrenr = denne_ordre;
    insert into uddata values(kundeid_t||'-'||to_char(total_t));
    commit;
    denne_ordre := denne_ordre + 1;
  END LOOP;
END;
/
select * from uddata;
delete from uddata where 1=1;
commit;
```

```
-- OPGAVE 7.1

DECLARE
  cursor med is select mednr, stilling, hyre from medarb;
  de_med med%ROWTYPE;
  nyhyre medarb.hyre%TYPE;
  total_hyre medarb.hyre%TYPE;

BEGIN
  total_hyre := 0;
  open med;
  loop
    fetch med into de_med;
    exit when med%NOTFOUND;
    total_hyre := total_hyre + (de_med.hyre * 0.15);
    nyhyre := de_med.hyre * 1.15;
    insert into uddata values (to_char(de_med.mednr)||' * '||de_med.stil   g|
  end loop;
    insert into uddata values ('Samlet stigning: '||to_char(total_hyre))
END;
/
select * from uddata;
delete from uddata where 1=1;
commit;
```

```
-- OPGAVE 7.2

DECLARE
  cursor nye_ordrer is select kundeid, total from ordrer
     where total > 70000;
  kunde ordrer.kundeid%TYPE;
  ialt ordrer.total%TYPE;
  counter NUMBER;

BEGIN
  open nye_ordrer;
  counter := 1;
  loop
    fetch nye_ordrer into kunde, ialt;
    exit when nye_ordrer%NOTFOUND OR counter > 3;
    insert into uddata values(kunde||'  '||to_char(ialt));
    counter := counter + 1;
  end loop;
  close nye_ordrer;
  commit;
END;
/
select * from uddata;
delete from uddata where 1=1;
commit;
```

```
-- OPGAVE 73


DECLARE
  cursor ord is select kundeid from ordrer
    where kundeid not in (select kundeid from kunder)
    group by kundeid;
   counter number;
   denne_kunde kunder.kundeid%TYPE;

BEGIN

  open ord;
  loop
    fetch ord into denne_kunde;
    exit when ord%NOTFOUND;
    insert into kunder values(denne_kunde,'ukendt firma','ukendt person'    ,'
    counter := ord%ROWCOUNT;
    insert into uddata values(to_char(counter)||' '||denne_kunde);
  end loop;
  commit;
  close ord;

END;
/
select * from uddata;
delete from uddata where 1=1;
commit;
```

```
-- OPGAVE 7.4


DECLARE
   cursor ansatte is select mednr, efternavn, hyre from medarb
         order by hyre for update of hyre;
   pulje medarb.hyre%TYPE;
   d_a ansatte%ROWTYPE;
BEGIN
   pulje := 30000;
   open ansatte;
   loop
     fetch ansatte into d_a;
     exit when ansatte%NOTFOUND;
     exit when (pulje - (d_a.hyre * 0.1) <= 0);
     update medarb set hyre = hyre * 1.1 where current of ansatte;
     pulje := pulje - (d_a.hyre * 0.1);
     insert into uddata values (d_a.efternavn||' fik '||to_char(d_a.hyre   1)
   end loop;
   commit;
   close ansatte;
END;
/
select * from uddata;
delete from uddata where 1=1;
commit;
```

```
-- OPGAVE 81

DECLARE
   cursor alfki is select kundeid, ordrenr, total, fragt
   from ordrer where kundeid = 'ALFKI';
   d_o alfki%ROWTYPE;
   fragt_rate number(5,2);

BEGIN
  open alfki;
  loop
    begin
      fetch alfki into d_o;
      exit when alfki %NOTFOUND;
      fragt_rate := (d_o.fragt / d_o.total)*100;
      insert into uddata values(to_char(d_o.ordrenr)||' '||to_char(fragt   te
    exception
      when zero_divide then
       insert into uddata values('Ordre '||to_char(d_o.ordrenr)||' fejl'
    end;
  end loop;
  close alfki;
END;
/
select * from uddata;
delete from uddata where 1=1;
commit;
```

```
-- OPGAVE 8.2

DECLARE
   fejl_i_data EXCEPTION;
 cursor o is select ordrenr from ordrer order by ordrenr;
   d_o ordrer.ordrenr%TYPE;
   forrige ordrer.ordrenr%TYPE;
   fejl_id ordrer.ordrenr%TYPE;

BEGIN
  forrige := 9999;
  open o;
  loop
    fetch o into d_o;
    exit when o%notfound;
    if d_o <> forrige + 1 then
      fejl_id := d_o;
      raise fejl_i_data;
    end if;
    forrige := d_o;
  end loop;
  close o;
EXCEPTION
  when fejl_i_data then
    insert into uddata values ('FEJL ved ordrer '||to_char(fejl_id));
END;
/
select * from uddata;
delete from uddata where 1=1;
commit;
```

```
-- OPGAVE 8.3


DECLARE
msg char(200);
begin
   for num in 2000 .. 2010
   loop
     msg := sqlerrm(-num);
     insert into uddata values(msg);
   end loop;
end;
/
select * from uddata;
delete from uddata where 1=1;
commit;
```

Dansk Data Elektronik A/S
Herley Hovedgade 199
DK-2730 Herlev
Tel: +45 42 84 90 11
Fax: +45 42 84 52 27