PL/SQL programozás

Alkalmazásfejlesztés Oracle 10g-ben

Gábor, András

Juhász, István

PL/SQL programozás: Alkalmazásfejlesztés Oracle 10g-ben

Gábor, András
Juhász, István
Tóth, Attila

Ez a könyv az Oktatási Minisztérium támogatásával, a Felsőoktatási Pályázatok Irodája által lebonyolított felsőoktatási Tankönyv- és Szakkönyvtámogatási Pályázat keretében jelent meg.

Publication date 2007

A tananyag a TÁMOP-4.1.2-08/1/A-2009-0046 számú Kelet-magyarországi Informatika Tananyag Tárház projekt keretében készült. A tananyagfejlesztés az Európai Unió támogatásával és az Európai Szociális Alap társfinanszírozásával valósult meg.

Nemzeti Fejlesztési Ügynökség http://ujszechenyiterv.gov.hu/ *06 40 638-638*

Tartalom

Az ábrák listája

A táblázatok listája

Előszó

Jelen könyv a 2002-ben megjelent *PL/SQL-programozás – Alkalmazásfejlesztés Oracle9i-ben* könyv (*lásd* [2]) javított,átdolgozott, a 10*g* verzióba beépített új eszközöket is tartalmazó változata, amely a PL/SQL nyelv lehetőségeit tárgyalja, kitérve az alapvető nyelvi eszközök ismertetésére és azok használatára a gyakorlatban.

Ez a könyv elsősorban *tankönyv*, amely egy felsőoktatási kurzus anyagát tartalmazza. A megértést sok példa szolgálja. A könyv *haladó* ismereteket tartalmaz, amelyek elsajátításához az alábbi informatikai alapismeretek szükségesek:

- a relációs adatmodell fogalmai;
- a relációs adatbázis-kezelő rendszerek alapeszközei;
- a szabvány SQL:1999 nyelv ismerete;
- az Oracle SQL és SQL*Plus ismerete;
- az eljárásorientált és az objektumorientált programozási nyelvek alapeszközeinek fogalmi szintű ismerete;
- programozási gyakorlat egy eljárásorientált (például C) és egy objektumorinetált (például C++ vagy Java) programozási nyelven.

A PL/SQL nyelv elsajátításához ezen túlmenően sok segítséget adhat az Ada nyelv ismerete. Ezen alapismeretek megtalálhatók az [1], [3], [4], [5], [6], [8], [9], [10], [11], [12], [24] művekben.

A könyv 16 fejezetből áll, ezek ismeretanyaga fokozatosan egymásra épül, tehát feldolgozásukat ebben a sorrendben javasoljuk.

A könyv tankönyvként használható a felsőoktatási intézmények informatikai szakjainak haladó kurzusain. Feldolgozható önálló tanulással is, bár nem elsősorban ilyen céllal íródott. Egy azonban biztos, a tanulás csak akkor lesz eredményes, ha a megírt kódokat lefuttatjuk, teszteljük, átírjuk, egyszóval kipróbáljuk őket, hogyan is működnek a gyakorlatban.

A könyv megírásánál szempont volt az is, hogy részben *referenciakönyvként* is használni lehessen. A nyelvi eszközök ismertetésénél általában törekedtünk a teljességre, noha ez nem mindig sikerült. Ennek oka részben a terjedelmi korlátokban, részben didaktikai okokban keresendő. Egyes utasítások bizonyos opciói, utasításrészei, előírásai nem szerepelnek, különösen igaz ez az SQL utasításoknál. A teljes információt a [13]–[22] dokumentációkban lehet megtalálni.

Referencia jellege miatt a könyvet sikerrel forgathatják az Oracle-környezetben dolgozó gyakorló szakemberek is, ha most szeretnék elsajátítani a PL/SQL nyelvet vagy pedig az újabb verzióba beépített új eszközökkel szeretnének megismerkedni.

Debrecen, 2006. július

A szerzők

1. fejezet - Bevezetés

Az Oracle objektumrelációs adatbázis-kezelő rendszer, amely alapvetően relációs eszközöket tartalmaz és ezeket egészíti ki objektumorientált eszközökkel. Mint a relációs adatbázis-kezelő rendszerek általában, az Oracle is a szabványos SQL nyelvet használja az adatok kezelésére. Az Oracle az SQL:1999 szabványt [9] támogatja. Az SQL tipikus *negyedik generációs* nyelv, amely *deklaratív* jellemzőkkel rendelkezik. Ez azt jelenti, hogy a nyelv parancsai segítségével leírjuk, hogy *mit* kell csinálni, de azt nem adjuk meg, hogy *hogyan*. A parancs végrehajtásának részletei rejtettek maradnak.

A deklaratív SQL mellett *befogadó* nyelvként alkalmazhatók az olyan *eljárásorientált (procedurális)* nyelvek, mint a C vagy a Pascal, illetve az olyan *objektumorientált* nyelvek, mint a C++ vagy a Java. Ezek *algoritmikus* nyelvek, amelyeken a probléma megoldásának menetét a programozó adja meg.

A PL/SQL a kétféle paradigma egyesítését valósítja meg, ezáltal igen nagy kifejezőerejű nyelvet ad. A PL/SQL az SQL procedurális kiterjesztésének tekinthető, amely a következő konstrukciókat adja az SQL-hez:

- változók és típusok,
- vezérlési szerkezet,
- alprogramok és csomagok,
- kurzorok,
- kivételkezelés,
- objektumorientált eszközök.

A PL/SQL nem tartalmaz I/O utasításokat.

Könyvünkben az Oracle10*g* PL/SQL verzióját tárgyaljuk. Törekedtünk a teljességre, az adott verzió minden utasítása szerepel valamelyik fejezetben. Feltételezzük az SQL utasítások ismeretét, ezért azoknak gyakran csak azon utasításrészeit elemezzük, melyek a PL/SQL-hez kötődnek, vagy az SQL-ben nem is szerepelnek. Néhány bonyolultabb PL/SQL utasításnál egyes opciók, előírások hiányozhatnak. A részletes információt igénylők számára a [19], [21] dokumentációk állnak rendelkezésre.

1. A könyvben alkalmazott jelölések, konvenciók

Minden utasítás esetén formálisan megadjuk annak szintaxisát. A formális leírásnál az alábbi

jelöléseket használjuk:

- A *terminálisok* nagybetűs formában jelennek meg (például LOOP).
- A *nemterminálisok* dőlt kisbetűs formában szerepelnek (például *típusnév*). Ha a nemterminális megnevezése több szóból áll, a szavak közé aláhúzás (_) kerül (például *kurzorváltozó_név*).
- Az *egyéb szimbólumok* az írásképükkel jelennek meg (például :=).
- Az *alternatívákat* függőleges vonal (`|`) választja el (például `MAP|ORDER`).
- Az *opcionális elemek* szögletes zárójelben (`[ ]`) állnak (például `[NOT NULL]`).
- A *kötelezően megadandó alternatívákat* kapcsos zárójelek (`{ }`) zárják közre (például `{MAP|ORDER}`).
- Az *iteráció* jelölésére a három pont (…) szolgál (például `oszlop[,oszlop]`…).

A könyvben közölt kódrészletekben a jobb olvashatóság érdekében a következő jelölési

konvenciókat alkalmaztuk:

- Az alapszavak és a standard, beépített PL/SQL azonosítók nagybetűsek:

.

.

.

```
BEGIN

v_Datum := hozzaad(SYSDATE, 1, 'kiskutyafüle');

EXCEPTION

WHEN hibas_argumentum THEN

DBMS_OUTPUT.PUT_LINE('Blokk1 - hibás argumentum: '

|| SQLCODE || ', ' || SQLERRM);

END;
```

.

.

.

- A több szóból álló azonosítókban a szavakat aláhúzással (_) választjuk el.
- Az adatbázisbeli táblák nevei és az oszlopaik nevei a definíciójukban nagy kezdőbetűsek, de mindenütt másutt kisbetűsek:

```
SELECT id, cim FROM konyv;
```

- A programozói objektumok írására a következők vonatkoznak:

Csomagok és alprogramok nevei kisbetűsek: `konyvtar_csomag.lejart_konyvek`.

Formális paraméterek nevét `p_` prefix után nagy kezdőbetűvel írjuk: `p_Max_konyv`.

A változók nevét `v_` prefix után nagy kezdőbetűvel írjuk: `v_Max_konyv`. A rövid változók (például ciklusváltozó) nevénél ettől eltérően nem használunk prefixet: `i, rv`.

A nevesített konstansok nevét `c_` prefix után nagy kezdőbetűvel írjuk:

`c_Max_konyv_init`.

A kurzorok nevét `cur_` prefix után nagy kezdőbetűvel írjuk:

`cur_Lejart_kolcsonzesek`.

Az adatbázisban tárolt típusok nevét `T_` prefix után nagy kezdőbetűvel írjuk: `T_Szerzok`.

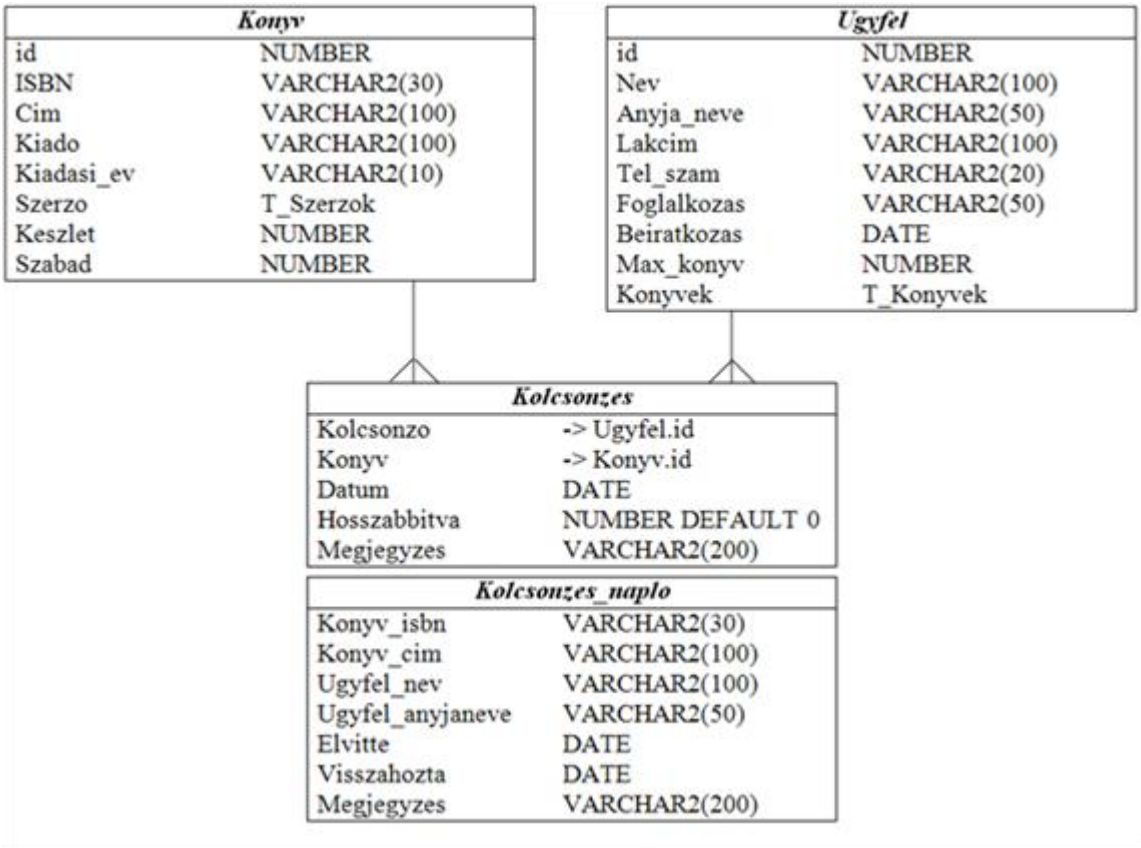
Az egyéb típusok nevét `t_` prefix után kis kezdőbetűvel írjuk: `t_nev`.

A kivételek nevét kisbetűvel írjuk (prefix nélkül): `hibas_argumentum`.

Az adatbázis triggereinek nevét `tr_` prefix után kisbetűvel írjuk: `tr_insert_kolcsonzes`.

2. A példákban használt adatbázistáblák

A könyv példáinak nagy része a következő adatbázissémára épül.



Tábláink egy egyszerű könyvtári nyilvántartórendszert modelleznek. Példáinkon keresztül megmutatjuk, hogyan oldhatók meg PL/SQL segítségével az ilyen rendszerben felmerülő problémák.

Az adatbázisséma a könyvtár működési szabályainak egy részét képes biztosítani. Így például nem kölcsönözhet könyvet az, aki nincs beíratkozva és nem lehet olyan könyvet kölcsönözni, amellyel a könyvtár nem rendelkezik. Lehetnek azonban olyan logikai szabályok, amelyeket a séma nem tud kezelni, ezért programozói eszközökkel kell azok teljesülését biztosítani. A mi könyvtárunkban csak néhány ilyen szabály van, amelyekre a példák során többször fogunk hivatkozni. Ezek a következők:

- Minden könyv kölcsönzési ideje egységesen 30 nap.
- Egy könyv kölcsönzése legfeljebb kétszer hosszabbítható, tehát egy könyv maximum 90 napig lehet valakinél.
- Minden kölcsönző csak a számára megszabott mennyiségű (kvótányi) könyvet kölcsönözheti egyszerre.

A séma tulajdonosa és az általunk használt felhasználó neve PLSQL, jelszava szintén PLSQL. Ha a példákban az adatbázis-objektumokat sémanévvel minősítjük, akkor ezt fogjuk használni.

A sémát létrehozó SQL utasítások a következők:

```
CREATE TYPE T_Szerzok IS

VARRAY (10) OF VARCHAR2(50)

/

CREATE TABLE Konyv (

id NUMBER,

ISBN VARCHAR2(30) NOT NULL,

Cim VARCHAR2(100) NOT NULL,
```

```
Kiado VARCHAR2(100) NOT NULL,
Kiadasi_ev VARCHAR2(10) NOT NULL,
Szerzo T_Szerzok NOT NULL,
Keszlet NUMBER NOT NULL,
Szabad NUMBER NOT NULL,
CONSTRAINT konyv_pk PRIMARY KEY (id),
CONSTRAINT konyv_szabad CHECK (Szabad >= 0)
)
/
CREATE SEQUENCE konyv_seq START WITH 100 INCREMENT BY 5
/
CREATE TYPE T_Tetel IS OBJECT(
Konyv_id NUMBER,
Datum DATE
)
/
CREATE TYPE T_Konyvek IS
TABLE OF T_Tetel
/
CREATE TABLE Ugyfel (
id NUMBER,
Nev VARCHAR2(100) NOT NULL,
Anyja_neve VARCHAR2(50) NOT NULL,
Lakcim VARCHAR2(100) NOT NULL,
Tel_szam VARCHAR2(20),
Foglalkozas VARCHAR2(50),
Beiratkozas DATE NOT NULL,
Max_konyv NUMBER DEFAULT 10 NOT NULL,
Konyvek T_Konyvek DEFAULT T_Konyvek(),
CONSTRAINT ugyfel_pk PRIMARY KEY (id)
) NESTED TABLE Konyvek STORE AS ugyfel_konyvek
/
CREATE SEQUENCE ugyfel_seq START WITH 100 INCREMENT BY 5
/
CREATE TABLE Kolcsonzes (
Kolcsonzo NUMBER NOT NULL,
```

```
Konyv NUMBER NOT NULL,
Datum DATE NOT NULL,
Hosszabbitva NUMBER DEFAULT 0 NOT NULL,
Megjegyzes VARCHAR2(200),
CONSTRAINT kolcsonzes_fk1 FOREIGN KEY (Kolcsonzo)
REFERENCES Ugyfel(Id),
CONSTRAINT kolcsonzes_fk2 FOREIGN KEY (Konyv)
REFERENCES Konyv(Id)
)
/
CREATE TABLE Kolcsonzes_naplo (
Konyv_isbn VARCHAR2(30) NOT NULL,
Konyv_cim VARCHAR2(100) NOT NULL,
Ugyfel_nev VARCHAR2(100) NOT NULL,
Ugyfel_anyjaneve VARCHAR2(50) NOT NULL,
Elvitte DATE NOT NULL,
Visszahozta DATE NOT NULL,
Megjegyzes VARCHAR2(200)
)
/
```

A sémát példaadatokkal feltöltő SQL utasítások:

```
/*
Konyv:
id NUMBER PRIMARY KEY,
ISBN VARCHAR2(30) NOT NULL,
Cim VARCHAR2(100) NOT NULL,
Kiado VARCHAR2(100) NOT NULL,
Kiadasi_ev VARCHAR2(10) NOT NULL,
Szerzo T_Szerzok NOT NULL,
Keszlet NUMBER NOT NULL,
Szabad NUMBER NOT NULL
*/
/* Az SQL*Plus escape karakterét \(backslash)-re állítjuk,
mert & karakter is van a sztringekben.
Ezt vegye figyelembe, ha nem SQL*Plus-t használ! */
```

```
SET ESCAPE \
INSERT INTO konyv VALUES (
5, 'ISBN 963 19 0297 8',
'A római jog története és institúciói',
'Nemzeti Tankönyvkiadó Rt.', 1996,
T_Szerzok('dr. Földi András', 'dr. Hamza Gábor'),
20, 19
)
/
INSERT INTO konyv VALUES (
10, 'ISBN 963 8453 09 5',
'A teljesség felé',
'Tericum Kiadó', 1995,
T_Szerzok('Weöres Sándor'),
5, 4
)
/
INSERT INTO konyv VALUES (
15, 'ISBN 963 9077 39 9',
'Piszkos Fred és a többiek',
'Könyvkuckó Kiadó', 2000,
T_Szerzok('P. Howard', 'Rejtő Jenő'),
5, 4
)
/
INSERT INTO konyv VALUES (
20, 'ISBN 3-540-42206-4',
'ECOOP 2001 - Object-Oriented Programming',
'Springer-Verlag', 2001,
T_Szerzok('Jorgen Lindskov Knudsen (Ed.)', 'Gerthard Goos', 'Juris
Hartmanis','Jan van Leeuwen'),
3, 2
)
/
INSERT INTO konyv VALUES (
25, 'ISBN 963 03 9005 1',
```

```
'Java - start!',
'Logos 2000 Bt.', 1999,
T_Szerzok('Vég Csaba', 'dr. Juhász István'),
10, 9
)
/
INSERT INTO konyv VALUES (
30, 'ISBN 1-55860-456-1',
'SQL:1999 Understanding Relational Language Components',
'Morgan Kaufmann Publishers', 2002,
T_Szerzok('Jim Melton', 'Alan R. Simon'),
3, 1
)
/
INSERT INTO konyv VALUES (
35, 'ISBN 0 521 27717 5',
'A critical introduction to twentieth-century American drama - Volume 2',
'Cambridge University Press', 1984,
T_Szerzok('C. W. E: Bigsby'),
2, 0
)
/
INSERT INTO konyv VALUES (
40, 'ISBN 0-393-95383-1',
'The Norton Anthology of American Literature - Second Edition - Volume 2',
'W. W. Norton \& Company, Inc.', 1985,
T_Szerzok('Nina Baym', 'Ronald Gottesman', 'Laurence B. Holland',
'Francis Murphy', 'Hershel Parker', 'William H. Pritchard',
'David Kalstone'),
2, 1
)
/
INSERT INTO konyv VALUES (
45, 'ISBN 963 05 6328 2',
'Matematikai zseblexikon',
'TypoTeX Kiadó', 1992,
```

```
T_Szerzok('Denkinger Géza', 'Scharnitzky Viktor', 'Takács Gábor',
'Takács Miklós'),
5, 3
)
/
INSERT INTO konyv VALUES (
50, 'ISBN 963-9132-59-4',
'Matematikai Kézikönyv',
'TypoTeX Kiadó', 2000,
T_Szerzok('I. N. Bronstejn', 'K. A. Szemangyajev', 'G. Musiol', 'H. Mühlig'),
5, 3
)
/
/*
Ugyfel:
id NUMBER PRIMARY KEY,
Nev VARCHAR2(100) NOT NULL,
Anyja_neve VARCHAR2(50) NOT NULL,
Lakcim VARCHAR2(100) NOT NULL,
Tel_szam VARCHAR2(20),
Foglalkozas VARCHAR2(50),
Beiratkozas DATE NOT NULL,
Max_konyv NUMBER DEFAULT 10 NOT NULL,
Konyvek T_Konyvek DEFAULT T_Konyvek()
A nevek és a címek kitalált adatok, így az irányítószámok,
városok, utcanevek a valóságot nem tükrözik, ám a célnak
tökéletesen megfelelnek.
*/
INSERT INTO ugyfel VALUES (
5, 'Kovács János', 'Szilágyi Anna',
'4242 Hajdúhadház, Jókai u. 3.', '06-52-123456',
'Középiskolai tanár', '00-MÁJ. -24', 10,
T_Konyvek()
)
/
INSERT INTO ugyfel VALUES (
```

```
10, 'Szabó Máté István', 'Szegedi Zsófia',
'1234 Budapest, IX. Kossuth u. 51/b.', '06-1-1111222',
NULL, '01-MÁJ. -23', 10,
T_Konyvek(T_Tetel(30, '02-ÁPR. -21'),
T_Tetel(45, '02-ÁPR. -21'),
T_Tetel(50, '02-ÁPR. -21'))
)
/
INSERT INTO ugyfel VALUES (
15, 'József István', 'Ábrók Katalin',
'4026 Debrecen, Bethlen u. 33. X./30.', '06-52-456654',
'Programozó', '01-SZEPT-11', 5,
T_Konyvek(T_Tetel(15, '02-JAN. -22'),
T_Tetel(20, '02-JAN. -22'),
T_Tetel(25, '02-ÁPR. -10'),
T_Tetel(45, '02-ÁPR. -10'),
T_Tetel(50, '02-ÁPR. -10'))
)
/
INSERT INTO ugyfel VALUES (
20, 'Tóth László', 'Nagy Mária',
'1122 Vác, Petőfi u. 15.', '06-42-154781',
'Informatikus', '1996-ÁPR. -01', 5,
T_Konyvek(T_Tetel(30, '02-FEBR. -24'))
)
/
INSERT INTO ugyfel VALUES (
25, 'Erdei Anita', 'Cserepes Erzsébet',
'2121 Hatvan, Széchenyi u. 4.', '06-12-447878',
'Angol tanár', '1997-DEC. -05', 5,
T_Konyvek(T_Tetel(35, '02-ÁPR. -15'))
)
/
INSERT INTO ugyfel VALUES (
30, 'Komor Ágnes', 'Tóth Eszter',
'1327 Budapest V., Kossuth tér 8.', NULL,
```

```
'Egyetemi hallgató', '00-JÚN. -11', 5,
T_Konyvek(T_Tetel(5, '02-ÁPR. -12'),
T_Tetel(10, '02-MÁRC. -12'))
)
/
INSERT INTO ugyfel VALUES (
35, 'Jaripekka Hämälainen', 'Pirkko Lehtovaara',
'00500 Helsinki, Lintulahdenaukio 6. as 15.', '+358-9-1234567',
NULL, '01-AUG. -24', 5,
T_Konyvek(T_Tetel(35, '02-MÁRC. -18'),
T_Tetel(40, '02-MÁRC. -18'))
)
/
/*
Konzisztenssé tesszük az adatbázist a megfelelő kolcsonzes
bejegyzésekkel.
*/
/* Kölcsönző: Szabó Máté István */
INSERT INTO kolcsonzes VALUES (
10, 30, '02-ÁPR. -21', 0, NULL
)
/
INSERT INTO kolcsonzes VALUES (
10, 45, '02-ÁPR. -21', 0, NULL
)
/
INSERT INTO kolcsonzes VALUES (
10, 50, '02-ÁPR. -21', 0, NULL
)
/
/* Kölcsönző: József István */
INSERT INTO kolcsonzes VALUES (
15, 15, '02-JAN. -22', 2, NULL
)
/
INSERT INTO kolcsonzes VALUES (
```

```
15, 20, '02-JAN. -22', 2, NULL
)
/
INSERT INTO kolcsonzes VALUES (
15, 25, '02-ÁPR. -10', 0, NULL
)
/
INSERT INTO kolcsonzes VALUES (
15, 45, '02-ÁPR. -10', 0, NULL
)
/
INSERT INTO kolcsonzes VALUES (
15, 50, '02-ÁPR. -10', 0, NULL
)
/
/* Kölcsönző: Tóth László */
INSERT INTO kolcsonzes VALUES (
20, 30, '02-FEBR. -24', 2, NULL
)
/
/* Kölcsönző: Erdei Anita */
INSERT INTO kolcsonzes VALUES (
25, 35, '02-ÁPR. -15', 0, NULL
)
/
/* Kölcsönző: Komor Ágnes */
INSERT INTO kolcsonzes VALUES (
30, 5, '02-ÁPR. -12', 0, NULL
)
/
INSERT INTO kolcsonzes VALUES (
30, 10, '02-MÁRC. -12', 1, NULL
)
/
/* Kölcsönző: Jaripekka Hämälainen */
INSERT INTO kolcsonzes VALUES (
```

```
35, 35, '02-MÁRC. -18', 0, NULL

)

/

INSERT INTO kolcsonzes VALUES (

35, 40, '02-MÁRC. -18', 0, NULL

)

/
```

Mivel az azonosítókkal megadott kölcsönzési rekordok és a beágyazott táblák is nehezen olvashatók, létrehozhatunk egy nézetet, amely átláthatóbbá teszi adatainkat:

```
/*

A kölcsönzések ügyfél-könyv párjai.

*/

CREATE VIEW ugyfel_konyv AS

SELECT u.id AS ugyfel_id, u.nev AS Ugyfel,

k.id AS konyv_id, k.cim AS Konyv

FROM ugyfel u, konyv k

WHERE k.id IN (SELECT konyv_id FROM TABLE(u.konyvek))

ORDER BY UPPER(u.nev), UPPER(k.cim)

/
```

Ennek tartalma a következő:

```
SQL> SELECT * FROM ugyfel_konyv;

UGYFEL_ID UGYFEL

---------- ----------------------------------------------------------------

KONYV_ID KONYV

---------- ----------------------------------------------------------------

25 Erdei Anita

35 A critical introduction to twentieth-century American drama - Volume 2

35 Jaripekka Hämälainen

35 A critical introduction to twentieth-century American drama - Volume 2

35 Jaripekka Hämälainen

40 The Norton Anthology of American Literature - Second Edition - Volume 2

15 József István

20 ECOOP 2001 - Object-Oriented Programming

15 József István

25 Java - start!

15 József István
```

```
50 Matematikai Kézikönyv

15 József István

45 Matematikai zseblexikon

15 József István

15 Piszkos Fred és a többiek

30 Komor Ágnes

5 A római jog története és institúciói

30 Komor Ágnes

10 A teljesség felé

10 Szabó Máté

50 Matematikai Kézikönyv

10 Szabó Máté

45 Matematikai zseblexikon

10 Szabó Máté

30 SQL:1999 Understanding Relational Language Components

20 Tóth László

30 SQL:1999 Understanding Relational Language Components

14 sor kijelölve.
```

A példák futtatási környezete

A könyv példáit egy 32 bites processzoron futó Linux operációs rendszeren telepített Oracle 10*g* Release 2 adatbázis-kezelőben futtattuk.

Az Oracle tulajdonságai:

- Verzió: Oracle Database 10*g* Enterprise Edition Release 10.2.0.1.0.
- Adatbázis karakterészlete: AL32UTF8. Ez egy ASCII alapú Unicode karakterkészlet, amely több-bájtos karaktereket is tartalmaz.
- NLS_LANG környezeti változó értéke: Hungarian_Hungary.EE8ISO8859P2.

A példákat SQL*Plusban futtattuk. Fontos, hogy az SQL*Plus a példák által kiírt sorokat csak akkor jeleníti meg, ha a SERVEROUTPUT engedélyezve van. A mi futtató környezetünkben a következő parancs állította be ennek az értékét:

```
SET SERVEROUTPUT ON SIZE 10000 FORMAT WRAPPED
```

Fontos, hogy WRAPPED formátumot használjunk, mert más formátumoknál az SQL*Plus

átformázza a szerver pufferét.

2. fejezet - Alapelemek

Ebben a fejezetben a PL/SQL nyelv alapeszközeit, alapfogalmait ismerjük meg. Ezen alapeszközök, alapfogalmak ismerete elengedhetetlen a PL/SQL programok írásánál.

1. Karakterkészlet

Egy PL/SQL program forrásszövegének (mint minden forrásszövegnek) a legkisebb alkotóelemei a *karakterek*. A PL/SQL nyelv karakterkészletének elemei a következők:

- betűk, az angol kis- és nagybetűk: A–Z és a–z;
- számjegyek: 0–9;
- egyéb karakterek: ( ) + – * / < > = ! ~ ^ ; : . ’ @ % , ” # $ & _ | { } ? [ ];
- szóköz, tabulátor, kocsivissza.

A PL/SQL-ben a kis- és nagybetűk nem különböznek, a sztring literálok belsejét kivéve.

A PL/SQL karakterkészlete része az ASCII karakterkészletének, amely egybájtos karakterkódú. Az Oracle támogatja a több-bájtos karakterek kezelését is, ennek tárgyalása azonban túlmutat jelen könyv keretein.

2. Lexikális egységek

A PL/SQL program szövegében a következő *lexikális egységek* használhatók:

- elhatárolók,
- szimbolikus nevek,
- megjegyzések,
- literálok.

Ezeket részletezzük a következőkben.

2.1. Elhatárolók

Az *elhatároló* egy olyan karakter vagy karaktersorozat (többkarakteres szimbólum), amely speciális jelentéssel bír a PL/SQL program szövegében. Az egykarakteres elhatárolókat a 2.1. táblázat tartalmazza. A PL/SQL többkarakteres szimbólumait a 2.2. táblázatban láthatjuk.

2.1. táblázat - Egykarakteres elhatárolók

Szimbólum	Jelentés
+	Összeadás operátora
%	Attribútum kezdőszimbóluma
’	Sztring literál határoló
.	Komponens szelektor
/	Osztás operátora

(	Részkifejezés vagy lista kezdete
)	Részkifejezés vagy lista vége
:	Gazdaváltozó kezdőszimbóluma
,	Felsorolás elválasztójele
*	Szorzás operátora
”	Idézőjeles azonosító határolójele
=	Hasonlító operátor
<	Hasonlító operátor
>	Hasonlító operátor
@	Távoli hivatkozás szimbóluma
;	Utasítás végjele
–	Kivonás és negatív előjel operátora

2.2. táblázat - A PL/SQL többkarakteres szimbólumai

Szimbólum	Jelentés
: =	Értékadás operátora
=>	Hozzárendelés operátora
||	Összefűzés operátora
**	Hatványozás operátora
<<	Címke kezdete
>>	Címke vége
/*	Megjegyzés kezdete
*/	Megjegyzés vége
..	Intervallum operátora
< >	Hasonlító operátor
!=	Hasonlító operátor
~=	Hasonlító operátor
^=	Hasonlító operátor

<=	Hasonlító operátor
>=	Hasonlító operátor
--	Egysoros megjegyzés kezdete

2.2. Szimbolikus nevek

Azonosítók

Az *azonosító* a PL/SQL program bizonyos elemeinek megnevezésére szolgál azért, hogy a program szövegében az adott elemekre ezzel a névvel tudjunk hivatkozni. Ilyen programelemek például a változók, kurzorok, kivételek, alprogramok és csomagok.

Az azonosító olyan karaktersorozat, amely betűvel kezdődik és esetlegesen betűvel, számjeggyel, illetve a $, _, # karakterekkel folytatódhat. Egy azonosító maximális hossza 30 karakter és minden karakter szignifikáns.

Mivel a PL/SQL a kis- és nagybetűket nem tekinti különbözőnek, ezért a következő azonosítók megegyeznek:

```
Hallgato

HALLGATO

hallgato
```

Az alábbiak szabályos azonosítók:

```
x

h3#

hallgato_azonosito

SzemelyNev
```

A következő karaktersorozatok nem azonosítók:

```
x+y -- a + nem megengedett karakter

_hallgato -- betűvel kell kezdôdnie

Igen/Nem -- a / nem megengedett karakter
```

Foglalt szavak

A *foglalt szó (alapszó, kulcsszó, fenntartott szó)* olyan karaktersorozat, amelynek maga a PL/SQL tulajdonít speciális jelentést és ez a jelentés nem megváltoztatható. Egy foglalt szó soha nem használható azonosítóként. Ugyanakkor viszont egy azonosító *része* lehet foglalt szó. Szabálytalan tehát a következő:

```
DECLARE

mod BOOLEAN;
```

Szabályos viszont az alábbi, jóllehet a `modor` azonosító két foglalt szót is tartalmaz:

```
DECLARE

modor BOOLEAN;
```

A PL/SQL foglalt szavait az A) függelék tartalmazza.

Az Oracle-dokumentációk és egyes irodalmak általában a fenntartott szavakat nagybetűs alakban használják, a továbbiakban mi is ezt a konvenciót alkalmazzuk.

Előre definiált azonosítók

Az *előre definiált* (vagy *standard*) *azonosító* olyan azonosító, amelyet a STANDARD csomag definiál (például INVALID_NUMBER). Ezeket minden PL/SQL program használhatja a definiált jelentéssel, de bármikor felül is definiálhatja azt. Az újraértelmezés természetesen ideiglenes, csak lokálisan érvényes. Az előre definiált azonosítók felüldefiniálása hibaforrás lehet, inkonzisztenciához vezethet, ezért alkalmazását nem javasoljuk.

Idézőjeles azonosító

A PL/SQL lehetővé teszi, hogy egy azonosítót idézőjelek közé zárjunk. Ekkor az azonosítóban különböznek a kis- és nagybetűk, és használható benne a karakterkészlet összes karaktere. Így a következő idézőjeles azonosítók szabályosak:

```
"Igen/Nem"

"mod"

"Ez egy azonosito"

"x+y"
```

Az idézőjeles azonosítók használatával lehetővé válik fenntartott szavak azonosítóként való használata és „beszédesebb” azonosítók alkalmazása. Igazi jelentősége viszont ott van, hogy a PL/SQL olyan fenntartott szavai, amelyek az SQL-nek nem fenntartott szavai, használhatók SQL utasításban, például akkor, ha az SQL tábla egy oszlopának neve fenntartott PL/SQL szó. Ez persze elsősorban az angol nyelvű alkalmazásoknál jelent előnyt.

Itt jegyezzük meg, hogy az idézőjeles azonosítók az adatszótárban karakterhelyesen, az azonosítók viszont csupa nagybetűvel tárolódnak.

2.3. Megjegyzések

A *megjegyzés* az a programozási eszköz, amelyet a PL/SQL fordító nem vesz figyelembe. Ez egy olyan magyarázó szöveg, amely a program olvashatóságát, érthetőségét segíti elő. Egy jól megírt forrásszövegben általában kódszegmensenként vannak megjegyzések, amelyek az adott kódrészlet működését, használatát magyarázzák el. Egy megjegyzés tetszőleges karaktereket tartalmazhat.

A PL/SQL kétféle megjegyzést, úgymint e*gysoros* és *többsoros* megjegyzést ismer.

Egysoros megjegyzés

Az egysoros megjegyzés bármely sorban elhelyezhető, a -- jelek után áll és a sor végéig tart. A sorban a -- jelek előtt tényleges kódrészlet helyezhető el. Példa az egysoros megjegyzések alkalmazására:

```
-- a számítások kezdete

SELECT fiz INTO fizetes -- a fizetés lekérése

FROM dolg -- a dolgozó táblából

.

.

.
```

Az egysoros megjegyzések jól alkalmazhatók a program belövésénél. Ha a tesztelésnél egy adott sor tartalmát figyelmen kívül akarjuk hagyni, akkor rakjuk elé a -- jeleket, amint az alábbi példa mutatja:

```
-- SELECT * FROM dolg WHERE fiz>500000;
```

Többsoros megjegyzés

A többsoros megjegyzés a /* többkarakteres szimbólummal kezdődik és a */ jelkombináció jelzi a végét. A többsoros megjegyzés tetszőlegesen sok soron keresztül tarthat. Példa többsoros megjegyzésre:

```
BEGIN

.

.

.

/* A következő kód meghatározza

azon osztályok dolgozóinak átlagfizetését,

amelyek létszáma kevesebb mint 10. */

.

.

.

END;
```

A többsoros megjegyzések nem ágyazhatók egymásba.

Az olyan PL/SQL blokkban, amelyet egy előfordító dolgoz fel, nem alkalmazható egysoros megjegyzés, mivel az előfordító a sorvége karaktereket ignorálja. Ekkor használjunk többsoros megjegyzéseket.

2.4. Literálok

A *literál* egy explicit numerikus vagy sztring érték, amely a PL/SQL kód forrásszövegében önmagát definiálja.

Numerikus literálok

A numerikus literálnak két fajtája van: *egész* és *valós* literál.

Az egész literál egy opcionális + vagy – előjelből és egy számjegysorozatból áll:

```
011 5 -33 +139 0
```

A valós literál lehet *tizedestört* és *exponenciális* alakú. A tizedestörtnek opcionálisan lehet előjele (+ vagy –) és van egész és tört része. Az egész és tört rész között egy *tizedespont* áll:

```
6.666 3.14159 0.0 -1.0 +0.341
```

Az egész és tört rész valamelyike, amennyiben értéke nulla, elhagyható. Szabályos tizedestört valós literálok tehát a következők:

```
.5 +15.
```

Az exponenciális valós literál egy mantisszából (ami egész vagy tizedestört literál lehet) és egy karakterisztikából (ami egy egész) áll. A kettő között egy *E* vagy *e* betű szerepel:

```
2E5 -3.3e-11 .01e+28 3.14153e0
```

A nemzeti nyelvi támogatás beállításaitól függően a tizedespont helyett más karakter is állhat egy tizedestört literálban (például vessző).

A valós literál végén szerepelhet az *f* illetve a *d* betű, jelezve, hogy a literál BINARY_FLOAT, illetve BINARY_DOUBLE típusú. Például:

```
2.0f 2.0d
```

Sztring literálok

A sztring literálnak két különböző alakja lehet. Egyik esetben a sztring literál egy aposztrófok közé zárt tetszőleges látható karaktereket tartalmazó karaktersorozat. Példa ilyen sztring literálra:

```
'a' 'almafa' ' ''Barátaim!'' mondta'
```

Az ilyen sztring literálon belül a kis- és nagybetűk különböznek. Az aposztróf karaktert annak megkettőzésével lehet megjeleníteni. Például:

```
'Nem tudom ''jó'' helyen járok-e? '
```

A második alakú sztring literált speciális nyitó és záró szekvenciák határolják. Az ilyen sztring literál alakja:

```
q'nyitókarakter sztring zárókarakter'
```

Ha a *nyitókarakter* a [, {, <, (jelek egyike, akkor a *zárókarakter* kötelezően ezek természetes

párja, azaz rendre], }, >) jel. Egyéb *nyitókarakter* esetén a *zárókarakter* megegyezik

a nyitókarakterrel. Ennek az alaknak a következményeként az aposztrófokat nem kell a

sztring karaktersorozatán belül duplázni, azonban a *sztring* nem tartalmazhatja a *zárókarakter*

aposztróf jelkombinációt, mert az már a literál végét jelentené.

Példaként nézzük az *a'b* sztring literál néhány lehetséges felírási formáját:

Első („hagyományos") alak:

```
'a''b'
```

Második alak (speciális nyitó- és zárókarakterek):

```
q'[a'b]' q'{a'b}' q'<a'b>' q'(a'b)'
```

Második alak (tetszőleges nyitó- és zárókarakterek):

```
q'Ma'bM' q'za'bz' q'!a'b!' q'#a'b#'
```

Hibás literál:

```
q'Ma'bM'M'
```

Az üres sztring (''), amely egyetlen karaktert sem tartalmaz, a PL/SQL szerint azonos a NULL-lal. A sztring literálok adattípusa CHAR.

Egyes sztring literálok tartalmazhatnak olyan karaktersorozatokat, amelyek más típusok értékeit reprezentálhatják. Például az `'555'` literál számmá, a `'01-JAN-2002'` dátummá alakítható.

3. Címke

A PL/SQL programban bármely *végrehajtható* (*lásd* 5. fejezet) utasítás címkézhető. A címke egy azonosító, amelyet a << és >> szimbólumok határolnak és az utasítás előtt áll:

```
<<cimke>> A:= 3;
```

A PL/SQL-ben a címkének egyes vezérlőutasításoknál és a névhivatkozásoknál alapvető jelentősége van.

4. Nevesített konstans és változó

Értékek megnevezett kezelésére a PL/SQL programban nevesített konstansokat és változókat használhatunk. A program szövegében ezek mindig a nevükkel jelennek meg. A nevesített konstans értéke állandó, a változó értéke a futás folyamán tetszőlegesen megváltoztatható.

Nevesített konstans és változó egy blokk, alprogram vagy csomag deklarációs részében deklarálható. A deklarációs utasítás megadja a nevet, a típust, a kezdőértéket és esetlegesen a NOT NULL megszorítást.

A nevesített konstans deklarációjának szintaxisa:

```
név CONSTANT típus [NOT NULL] {:= | DEFAULT} kifejezés;
```

A változó deklarációjának szintaxisa:

```
név típus [[NOT NULL] {:= | DEFAULT} kifejezés];
```

A *típus* az alábbi lehet:

```
{ kollekciónév%TYPE
| kollekciótípus_név
| kurzornév%ROWTYPE
| kurzorváltozó_név%TYPE
| adatbázis_tábla_név{%ROWTYPE |.oszlopnév%TYPE}
| objektumnév%TYPE
| [REF] objektumtípus_név
| rekordnév%TYPE
| rekordtípus_név
| kurzorreferenciatípus_név
| skalár_adattípus_név
| változónév%TYPE}
```

A nevesített konstansnál kötelező a *kifejezés* megadása, ez határozza meg a fix értékét.

Változónál viszont ez opcionális, az explicit *kezdőértékadást* (vagy *inicializálást*) szolgálja.

A *név* egy azonosító.

Az *adatbázis_tábla_név.oszlopnév* egy adatbázisban létező tábla valamely oszlopát hivatkozza.

A *rekordnév* az adott ponton ismert, felhasználó által definiált rekord neve, a *mezőnév* ennek valamelyik mezője.

A *skalár_adattípus_név* valamelyik előre definiált skalár adattípus neve (*lásd* 3. fejezet).

A *változónév* egy programban deklarált változó neve.

A *kollekciónév* egy beágyazott tábla, egy asszociatív tömb vagy egy dinamikus tömb, a *kollekciótípus_név* egy ilyen felhasználói típus neve.

A *kurzornév* egy explicit kurzor neve.

A *kurzorváltozó_név* egy PL/SQL kurzorváltozó neve.

Az *objektumtípus_név* egy objektumtípus neve, az *objektumnév* egy ilyen típus egy példánya.

A *rekordtípus_név* egy felhasználó által létrehozott rekordtípus neve.

A *kurzorreferenciatípus_név* a felhasználó által létrehozott REF CURSOR típus neve vagy SYS_REFCURSOR.

A %TYPE egy korábban már deklarált kollekció, kurzorváltozó, mező, objektum, rekord, adatbázistábla oszlop vagy változó típusát veszi át és ezzel a típussal deklarálja a változót vagy nevesített konstansot. Használható még tábla oszlopának típusmegadásánál

is.

A %TYPE használatának két előnye van. Egyrészt nem kell pontosan ismerni az átvett típust, másrészt ha az adott eszköz típusa megváltozik, a változó típusa is automatikusan, futási időben követi ezt a változást.

1. példa

```
-- a v_Telszam deklarációja az ugyfel táblához kötődik a %TYPE miatt

v_Telszam ugyfel.tel_szam%TYPE DEFAULT '06-52-123456';

-- Az új változót felhasználhatjuk újabb deklarációban.

v_Telszam2 v_Telszam%TYPE;

-- A v_Telszam%TYPE típus azonban nem tartalmaz kezdőértékadást!

-- Így v_Telszam2 típusa megegyezik v_Telszam típusával,

-- kezdőértéke azonban NULL lesz.
```

A %ROWTYPE egy olyan rekordtípust szolgáltat, amely egy adatbázis-táblabeli sort, vagy egy kurzor által kezelt sort reprezentál. A rekordtípus mezői és a megfelelő oszlopok neve és típusa azonos.

2. példa

```
-- az ugyfel tábla szerkezetével megegyező rekordtípusú változók

v_Ugyfel ugyfel%ROWTYPE;

v_Ugyfel2 v_Ugyfel%TYPE;
```

A %ROWTYPE alkalmazása esetén a változónak nem lehet kezdőértéket adni. Ennek oka az, hogy tetszőleges rekordtípusú változónak sem lehet kezdőértéket adni.

A NOT NULL megszorítás megtiltja, hogy a változónak vagy nevesített konstansnak az értéke NULL legyen. Ha futás közben egy így deklarált változónak a NULL értéket próbálnánk adni, akkor a VALUE_ERROR kivétel váltódik ki. A NOT NULL megszorítás csak akkor adható meg egy változó deklarációjánál, ha az kezdőértéket kap.

3. példa

```
DECLARE

-- NOT NULL deklarációnál kötelező az értékadás

v_Szam1 NUMBER NOT NULL := 10;

-- v_Szam2 kezdőértéke NULL lesz

v_Szam2 NUMBER;

BEGIN

v_Szam1 := v_Szam2; -- VALUE_ERROR kivételt eredményez

END;

/
```

A NATURALN és POSITIVEN altípusok tartományának nem eleme a NULL (*lásd* 3.1. alfejezet). Ezen típusok használata esetén a kezdőértékadás kötelező. Ezért a következő deklarációk ekvivalensek:

```
szamlalo NATURALN NOT NULL := 0;

szamlalo NATURALN := 0;
```

Az alábbi deklaráció viszont szabálytalan:

```
szamlalo NATURALN;
```

A := vagy DEFAULT utasításrész a kezdőértékadást szolgálja, ahol a *kifejezés* egy tetszőleges, a megadott típussal kompatibilis típusú kifejezés.

Ha egy változónak nem adunk explicit kezdőértéket, akkor alapértelmezett kezdőértéke NULL lesz.

Hasonló módon kezdőérték adható még alprogram formális paramétereinek, kurzor

paramétereinek és rekord mezőinek.

4. példa

```
-- nevesített konstans deklarációja, az értékadó kifejezés egy

-- függvényhívás

c_Most CONSTANT DATE := SYSDATE;

-- változódeklaráció kezdőértékadás nélkül

v_Egeszszam PLS_INTEGER;

v_Logikai BOOLEAN;

-- változódeklaráció kezdőértékadással

v_Pozitiv POSITIVEN DEFAULT 1;

v_Idopecset TIMESTAMP := CURRENT_TIMESTAMP;

-- kezdőértékadás rekordtípus mezőjének

TYPE t_kiserlet IS RECORD (

leiras VARCHAR2(20),

probalkozas NUMBER := 0,

sikeres NUMBER := 0

);

-- rekord mezőinek csak a típusdeklarációban lehet kezdőértéket adni

v_Kiserlet t_kiserlet;
```

A nevesített konstansok és változók kezdőértékadása mindig megtörténik, valahányszor

aktivizálódik az a blokk vagy alprogram, amelyben deklaráltuk őket.

Csomagban deklarált nevesített konstans és változó kezdőértéket munkamenetenként

csak egyszer kap, kivéve ha a csomagra SERIALLY_REUSABLE pragma van érvényben. A

pragma leírását *lásd* a 10. fejezetben.

5. Pragmák

A *pragmák* a PL/SQL fordítónak szóló információkat tartalmaznak (gyakran hívják őket *fordítási direktíváknak* vagy *pszeudoutasításnak* is). A pragma feldolgozása mindig fordítási időben történik. A pragma a fordító működését befolyásolja, közvetlen szemantikai jelentése nincs. A pragma alakja a következő:

```
PRAGMA pragma_név[(argumentumok)];
```

A PL/SQL a következő pragmákat értelmezi:

- `AUTONOMOUS_TRANSACTION` (*lásd* 5.6.2. alfejezet),

- `EXCEPTION_INIT` (*lásd* 7. fejezet),
- `RESTRICT_REFERENCES` (*lásd* 9. fejezet),
- `SERIALLY_REUSABLE` (*lásd* 10. fejezet).

3. fejezet - Adattípusok

A literálok, nevesített konstansok, változók mindegyike rendelkezik *adattípus* komponenssel, amely meghatározza az általuk felvehető értékek tartományát, az értékeken végezhető műveleteket és az értékek tárbeli megjelenítési formáját (reprezentációját).

Az adattípusok lehetnek *előre definiált (beépített)* vagy a *felhasználó által definiált (felhasználói)* típusok. Az előre definiált adattípusokat a STANDARD csomag tartalmazza. Ezek minden PL/SQL programból elérhetők.

Egy tetszőleges adattípushoz (mint *alaptípushoz*) kapcsolódóan beszélhetünk *altípusról*. Az altípus ugyanazokkal a műveletekkel és reprezentációval rendelkezik, mint alaptípusa, de tartománya részhalmaza az alaptípus tartományának.

Az altípusok növelik a használhatóságot, az ISO/ANSI típusokkal való kompatibilitást és a kód olvashatóságát. A STANDARD csomag számos altípust definiál (*lásd* később ebben a fejezetben). A felhasználó maga is létrehozhat altípusokat.

Azt az altípust, amelynek tartománya megegyezik alaptípusának tartományával, *nemkorlátozott* altípusnak nevezzük. Ebben az esetben az alaptípus és az altípus neve szinonimáknak tekinthetők.

A PL/SQL ötféle adattípusról beszél. A *skalártípus* tartományának elemei nem rendelkeznek belső szerkezettel, az *összetett* típuséi viszont igen. A *referenciatípus* tartományának elemei más típusok elemeit címző *pointerek*. A *LOB típus* olyan értékeket tartalmaz (ezeket lokátoroknak nevezzük), amelyek egy nagyméretű objektum helyét határozzák meg. Az *objektumtípusok* az egységbe zárást szolgálják.

A PL/SQL előre definiált adattípusait a 3.1. táblázat tartalmazza.

Ebben a fejezetben a skalár-, a LOB és a rekordtípusokról lesz szó, a további típusokat a későbbi fejezetekben tárgyaljuk.

1. Skalártípusok

A skalártípusok, mint az a 3.1. táblázatban látható, családokra tagolódnak. A következőkben

ezeket a családokat tekintjük át.

3.1. táblázat - Előre definiált adattípusok

SKALÁRTÍPUSOK		
Numerikus család	Karakteres család	Dátum/intervallum család
BINARY_DOUBLE	CHAR	DATE
BINARY_FLOAT	CHARACTER	INTERVAL DAY TO SECOND
BINARY_INTEGER	LONG	INTERVAL YEAR TO MONTH
DEC	LONG RAW	TIMESTAMP
DECIMAL	NCHAR	TIMESTAMP WITH TIME
DOUBLE PRECISION	NVARCHAR2	ZONE

FLOAT	RAW	TIMESTAMP WITH LOCAL
INT	ROWID	TIME ZONE
INTEGER	STRING	
NATURAL	UROWID	
NATURALN	VARCHAR	
NUMBER	VARCHAR2	
NUMERIC		
PLS_INTEGER		
POSITIVE		
POSITIVEN	Logikai család	
REAL	BOOLEAN	
SIGNTYPE		
SMALLINT		
ÖSSZETETT TÍPUSOK	LOB TÍPUSOK	REFERENCIATÍPUSOK
RECORD	BFILE REF	CURSOR
TABLE	BLOB	SYS_REFCURSOR
VARRAY	CLOB	REF objektumtípus
NCLOB		

Numerikus család

A numerikus típusok tartományának értékei egészek vagy valósak. Négy alaptípus tartozik a családba, ezek: NUMBER, BINARY_INTEGER, BINARY_DOUBLE és BINARY_FLOAT. A többi típus valamelyikük altípusaként van definiálva.

NUMBER TÍPUS

Ez a típus egész és valós számokat egyaránt képes kezelni. Megegyezik a NUMBER adatbázistípussal. Tartománya: 1E–130..10E125. Belső ábrázolása fixpontos vagy lebegőpontos

decimális. Szintaxisa a következő:

```
NUMBER [(p[,s])]
```

A teljes alakban *p* a *pontosság*, *s* a *skála* megadását szolgálja. Értékük csak egész literál lehet. A pontosság az összes számjegy számát, a skála a tizedes jegyek számát határozza meg. Az *s*-nek a –84..127 tartományba kell esnie, *p* alapértelmezett értéke 38, lehetséges értékeinek tartománya 1..38. Ha mindkettőt megadtuk, akkor a belső ábrázolás fixpontos. Ha sem *p*, sem *s* nem szerepel, akkor *s* értéke tetszőleges 1 és 38 között. Ha *p*

szerepel és *s*-et nem adjuk meg, akkor egész értéket kezelünk (tehát ekkor *s* = 0 és az ábrázolás fixpontos). Negatív skála esetén a kezelt érték a tizedesponttól balra, az *s*-edik jegynél kerekítésre kerül. Ha a kezelendő érték törtjegyeinek száma meghaladja a skála értékét, akkor *s* tizedesre kerekítés történik. A 3.2. táblázatban láthatunk példákat.

3.2. táblázat - Pontosság és skála a NUMBER típusnál

Típus	Kezelendő érték	Tárolt érték
NUMBER	1111.2222	1111.2222
NUMBER(3)	321	321
NUMBER(3)	3210	Túlcsordulás
NUMBER(4,3)	33.222	Túlcsordulás
NUMBER(4,3)	3.23455	3.235
NUMBER(3,–3)	1234	1000
NUMBER(3,–1)	1234	1230

A `NUMBER` nemkorlátozott altípusai:

```
DEC, DECIMAL, DOUBLE PRECISION, FLOAT, NUMERIC.
```

A `NUMBER` korlátozott altípusai:

```
REAL (p maximuma 18)

INT, INTEGER, SMALLINT (s = 0, tehát egészek).
```

A DEC, DECIMAL, NUMERIC, INT, INTEGER, SMALLINT típusok belső ábrázolása fixpontos, a REAL, DOUBLE PRECISION, FLOAT típusoké pedig lebegőpontos.

BINARY_INTEGER TÍPUS

A NUMBER típus belső ábrázolása hatékony tárolást tesz lehetővé, de az aritmetikai műveletek

közvetlenül nem végezhetők el rajta. A PL/SQL motor műveletek végzésénél a NUMBER típust automatikusan binárissá konvertálja, majd a művelet elvégzése után szintén automatikusan végrehajtja a fordított irányú átalakítást is.

Ha egy egész értéket nem akarunk tárolni, csak műveletet akarunk vele végezni, akkor használjuk a BINARY_INTEGER adattípust. Ez a típus tehát egész értékeket kezel a –2147483647..2147483647 tartományban. Ezeket az értékeket fixpontosan tárolja, így a műveletvégzés gyorsabb.

A PLS_INTEGER típus a BINARY_INTEGER nemkorlátozott altípusa, de nem teljesen kompatibilisek. Tartományuk azonos, de a PLS_INTEGER típus műveletei gépi aritmetikát használnak, így gyorsabbak, mint a könyvtári aritmetikát használó BINARY_INTEGER műveletek.

A BINARY_INTEGER korlátozott altípusai a következők:

NATURAL	0..2147483647
NATURALN	0..2147483647 és NOT NULL

POSITIVE	1..2147483647
POSITIVEN	1..2147483647 és NOT NULL
SIGNTYPE	–1,0,1

BINARY_DOUBLE, BINARY_FLOAT TÍPUSOK

A BINARY_FLOAT típus belső ábrázolása egyszeres pontosságú (32 bites), a BINARY_DOUBLE típusé dupla pontosságú (64 bites) bináris lebegőpontos. A számításintenzív alkalmazásokban játszanak elsődleges szerepet, mert velük az aritmetikai műveletek gyorsabban végezhetők, mert a gépi aritmetikát használják közvetlenül. Ezen típusok értékeivel végzett műveletek speciális értékeket eredményezhetnek, amelyeket ellenőrizhetünk beépített nevesített konstansokkal való hasonlítással, és ekkor nem váltódik ki kivétel (részletekért *lásd* [18], [19], [21]).

Karakteres család

A karakteres típusok tartományának elemei tetszőleges karaktersorozatok. Reprezentációjuk az alkalmazott karakterkódtól függ.

CHAR TÍPUS

Fix hosszúságú karakterláncok kezelésére alkalmas. Szintaxisa:

```
CHAR [(h [CHAR|BYTE])]
```

ahol *h* az 1..32767 intervallumba eső egész literál, alapértelmezése 1. A *h* értéke bájtokban (BYTE) vagy karakterekben (CHAR – ez az alapértelmezés) értendő és a hosszat adja meg. A karakterláncok számára mindig ennyi bájt foglalódik le, ha a tárolandó karakterlánc ennél rövidebb, akkor jobbról kiegészül szóközökkel.

A CHARACTER a CHAR nemkorlátozott altípusa, így annak szinonimájaként használható.

VARCHAR2 TÍPUS

Változó hosszúságú karakterláncok kezelésére alkalmas. Szintaxisa a következő:

```
VARCHAR2(h [CHAR|BYTE])
```

ahol *h* az 1..32767 intervallumba eső egész literál és a maximális hosszat adja meg. Értéke CHAR megadása esetén karakterekben, BYTE esetén bájtokban, ezek hiánya esetén karakterekben értendő. A maximális hossz legfeljebb 32767 bájt lehet.

A megadott maximális hosszon belül a kezelendő karakterláncok csak a szükséges mennyiségű

bájtot foglalják el.

A VARCHAR2 típus nemkorlátozott altípusai: STRING és VARCHAR.

NCHAR ÉS NVARCHAR2 TÍPUSOK

Az Oracle NLS (National Language Support – nemzeti nyelvi támogatás) lehetővé teszi egybájtos és több-bájtos karakterkódok használatát és az ezek közötti konverziót. Így az alkalmazások különböző nyelvi környezetekben futtathatók.

A PL/SQL két karakterkészletet támogat, az egyik az *adatbázis karakterkészlet*, amely az azonosítók és egyáltalán a forráskód kialakításánál használható, a másik a *nemzeti karakterkészlet*, amely az NLS-adatok kezelését teszi lehetővé.

Az NCHAR és NVARCHAR2 típusok megfelelnek a CHAR és VARCHAR2 típusoknak, de ezekkel a nemzeti karakterkészlet karaktereiből alkotott karakterláncok kezelhetők. További különbség, hogy a maximális hossz itt mindig karakterekben értendő.

LONG TÍPUS

A VARCHAR2-höz hasonló, változó hosszúságú karakterláncokat kezelő típus, ahol a maximális hossz 32760 bájt lehet.

LONG RAW TÍPUS

Változó hosszúságú bináris adatok (bájtsorozatok) kezelésére való. Maximális hossza 32760 bájt.

RAW TÍPUS

Nem strukturált és nem interpretált bináris adatok (például grafikus adat, kép, video) kezelésére való. Szintaxisa:

```
RAW(h)
```

ahol *h* az 1..32767 intervallumba eső egész literál, amely a hosszat határozza meg bájtokban.

ROWID, UROWID TÍPUSOK

Minden adatbázistábla rendelkezik egy ROWID nevű pszeudooszloppal, amely egy bináris értéket, a *sorazonosítót* tárolja. Minden sorazonosító a sor tárolási címén alapul. A *fizikai sorazonosító* egy „normális” tábla sorát azonosítja, a *logikai sorazonosító* pedig egy asszociatív tömbét. A ROWID adattípus tartományában fizikai sorazonosítók vannak. Az UROWID adattípus viszont fizikai, logikai és idegen (nem Oracle) sorazonosítókat egyaránt tud kezelni.

Dátum/intervallum család

Ezen családon belül három alaptípus létezik: DATE, TIMESTAMP és INTERVAL.

DATE TÍPUS

Ez a típus a dátum és idő információinak kezelését teszi lehetővé. Minden értéke 7 bájton tárolódik, amelyek rendre az évszázad, év, hónap, nap, óra, perc, másodperc adatait tartalmazzák.

A típus tartományába az időszámítás előtti 4712. január 1. és időszámítás szerinti 9999. december 31. közötti dátumok tartoznak. A Julianus naptár alkalmazásával az időszámítás előtti 4712. január 1-jétől eltelt napok számát tartalmazza a dátum.

Az aktuális dátum és idő lekérdezhető a SYSDATE függvény visszatérési értékeként.

TIMESTAMP TÍPUS

Ezen típus tartományának értékei az évet, hónapot, napot, órát, percet, másodpercet és a másodperc törtrészét tartalmazzák. Időbélyeg kezelésére alkalmas. Szintaxisa:

```
TIMESTAMP[(p)] [WITH [LOCAL] TIME ZONE]
```

ahol *p* a másodperc törtrészének kifejezésére használt számjegyek száma. Alapértelmezésben 6. A WITH TIME ZONE megadása esetén az értékek még a felhasználó időzónájának adatát is tartalmazzák. A LOCAL megadása esetén pedig az adatbázisban tárolt (és nem a felhasználói) időzóna adata kerül kezelésre.

INTERVAL TÍPUS

A INTERVAL típus segítségével két időbélyeg közötti időtartam értékeit kezelhetjük. Szintaxisa:

```
INTERVAL {YEAR[(p)] TO MONTH|DAY[(np)] TO SECOND[(mp)]}
```

A YEAR[(*p*)] TO MONTH az időintervallumot években és hónapokban adja. A *p* az évek értékének tárolására használt számjegyek száma. Alapértelmezett értéke 2.

A DAY[(*np*)] TO SECOND [(*mp*)] az időintervallumot napokban és másodpercekben adja; *np* a napok, *mp* a másodpercek értékének tárolására használt számjegyek száma; *np* alapértelmezett értéke 2, *mp*-é 6.

Logikai család

Egyetlen típus tartozik ide, a BOOLEAN, amelynek tartománya a logikai igaz, hamis és a NULL értéket tartalmazza. Logikai típusú literál nincs, de az Oracle három beépített nevesített konstanst értelmez. TRUE értéke a logikai igaz, FALSE értéke a logikai hamis és NULL értéke NULL.

2. LOB típusok

A LOB típusok lehetővé teszik nemstrukturált adatok (például szöveg, kép, video, hang) blokkjainak tárolását 4G méretig. A kezelésnél pedig az adatok részeinek gyors, közvetlen elérését biztosítják. A LOB típusok ún. LOB lokátorokat tartalmaznak, ezek nem mások, mint pointerek. Maguk a LOB értékek vagy az adatbázis tábláiban, vagy azokon kívül, de az adatbázisban vagy egy külső állományban tárolódnak. A BLOB, CLOB, NCLOB típusú adatok az adatbázisban, a BFILE típusú értékek az operációs rendszer állományaiban jelennek meg.

A PL/SQL a LOB értékeket a lokátorokon keresztül kezeli. Például egy BLOB típusú oszlop lekérdezése a lokátort adja vissza. A lokátorok nem adhatók át más tranzakcióknak és más munkamenetnek.

A LOB értékek kezeléséhez a PL/SQL beépített DBMS_LOB csomagjának eszközeit használhatjuk. A következőkben röviden áttekintjük a LOB típusokat, részletes információkat [8] és [13] tartalmaz.

BFILE

A BFILE arra való, hogy nagyméretű bináris objektumokat tároljon operációsrendszer-állományokban. Egy BFILE lokátor tartalmazza az állomány teljes elérési útját. A BFILE csak olvasható, tehát nem lehet módosítani. Mérete az operációs rendszertől függ, de nem lehet nagyobb 232 – 1 bájtnál. A BFILE adatintegritását nem az Oracle, hanem az operációs rendszer biztosítja.

BLOB, CLOB, NCLOB

A BLOB típus nagyméretű bináris objektumokat, a CLOB nagyméretű karaktersorozatokat, az NCLOB nagyméretű NCHAR karaktersorozatokat tárol az adatbázisban. Az adott típusú változó tartalma egy lokátor, amely a megfelelő LOB értéket címzi. Maximális méretük 4GB lehet. Mindhárom típus értékei visszaállíthatók és többszöröthetők. Kezelésük tranzakción belül történik, a változások véglegesíthetők és visszavonhatók.

3. A rekordtípus

A skalártípusok mindegyike beépített típus, a STANDARD csomag tartalmazza őket. Ezek tartományának elemei atomiak. Ezzel szemben az összetett típusok tartományának elemei mindig egy-egy adatcsoportot reprezentálnak. Ezeket a típusokat mindig felhasználói típusként kell létrehozni. Két összetett típuscsoport van. Az egyikbe tartozik a *rekord*, amely heterogén, összetett típus, a másikba tartozik az *asszociatív tömb*, a *beágyazott tábla* és a *dinamikus tömb*. Ez utóbbiakat összefoglaló néven *kollekciótípusoknak* hívjuk. A kollekciótípusokkal a 12. fejezet foglalkozik.

A rekord logikailag egybetartozó adatok heterogén csoportja, ahol minden adatot egy-egy mező tárol. A mezőnek saját *neve* és *típusa* van. A rekordtípus teszi lehetővé számunkra, hogy különböző adatok együttesét egyetlen logikai egységként kezeljünk. A rekord adattípus segítségével olyan programeszközöket tudunk deklarálni, amelyek egy adatbázistábla sorait közvetlenül tudják kezelni. Egy rekordtípus deklarációja a következőképpen történik:

```
TYPE név IS RECORD(

mezőnév típus [[NOT NULL] {:=|DEFAULT} kifejezés]

[,mezőnév típus [[NOT NULL] {:=|DEFAULT} kifejezés]]…);
```

A *név* a létrehozott rekordtípus neve, a továbbiakban deklarációkban a *rekord* típusának

megadására szolgál.

A *mezőnév* a rekord mezőinek, elemeinek neve.

A *típus* a REF CURSOR kivételével bármely PL/SQL típus lehet.

A NOT NULL megadása esetén az adott mező nem veheti fel a NULL értéket. Ha futási időben mégis ilyen értékadás következne be, akkor kiváltódik a VALUE_ERROR kivétel. NOT NULL megadása esetén kötelező az inicializálás.

A :=|DEFAULT utasításrész a mező inicializálására szolgál. A *kifejezés* a mező kezdőértékét

határozza meg.

Egy *rekord* deklarációjának alakja:

```
rekordnév rekordtípus_név;
```

1. példa

```
DECLARE

/* Rekorddefiníciók */

-- NOT NULL megszorítás és értékadás

TYPE t_aru_tetel IS RECORD (

kod NUMBER NOT NULL := 0,

nev VARCHAR2(20),

afa_kulcs NUMBER := 0.25

);

-- Az előzővel azonos szerkezetű rekord

TYPE t_aru_bejegyzes IS RECORD (

kod NUMBER NOT NULL := 0,

nev VARCHAR2(20),

afa_kulcs NUMBER := 0.25

);

-- %ROWTYPE és rekordban rekord

TYPE t_kolcsonzes_rec IS RECORD (

bejegyzes plsql.kolcsonzes%ROWTYPE,

kolcsonzo plsql.ugyfel%ROWTYPE,

konyv plsql.konyv%ROWTYPE

);

-- %ROWTYPE

SUBTYPE t_ugyfel_rec IS plsql.ugyfel%ROWTYPE;

v_Tetel1 t_aru_tetel; -- Itt nem lehet inicializálás és NOT NULL!

v_Tetel2 t_aru_tetel;

.

.

.
```

Egy rekord mezőire minősítéssel a következő formában tudunk hivatkozni:

```
rekordnév.mezőnév
```

2. példa

```
v_Tetel1.kod := 1;
```

Rekordot nem lehet összehasonlítani egyenlőségre vagy egyenlőtlenségre és nem tesztelhető a NULL értéke sem.

Komplex példa

```
DECLARE

/* Rekorddefiníciók */

-- NOT NULL megszorítás és értékadás

TYPE t_aru_tetel IS RECORD (

kod NUMBER NOT NULL := 0,

nev VARCHAR2(20),

afa_kulcs NUMBER := 0.25

);

-- Az előzővel azonos szerkezetű rekord

TYPE t_aru_bejegyzes IS RECORD (

kod NUMBER NOT NULL := 0,

nev VARCHAR2(20),

afa_kulcs NUMBER := 0.25

);

-- %ROWTYPE és rekordban rekord

TYPE t_kolcsonzes_rec IS RECORD (

bejegyzes plsql.kolcsonzes%ROWTYPE,

kolcsonzo plsql.ugyfel%ROWTYPE,

konyv plsql.konyv%ROWTYPE

);

-- %ROWTYPE

SUBTYPE t_ugyfel_rec IS plsql.ugyfel%ROWTYPE;

v_Tetel1 t_aru_tetel; -- Itt nem lehet inicializálás és NOT NULL!

v_Tetel2 t_aru_tetel;

v_Bejegyzes t_aru_bejegyzes;

-- %TYPE

v_Kod v_Tetel1.kod%TYPE := 10;

v_Kolcsonzes t_kolcsonzes_rec;

/* Függvény, ami rekordot ad vissza */

FUNCTION fv(
```

```
p_Kolcsonzo v_Kolcsonzes.kolcsonzo.id%TYPE,
p_Konyv v_Kolcsonzes.konyv.id%TYPE
) RETURN t_kolcsonzes_rec IS
v_Vissza t_kolcsonzes_rec;
BEGIN
v_Vissza.kolcsonzo.id := p_Kolcsonzo;
v_Vissza.konyv.id := p_Konyv;
RETURN v_Vissza;
END;
BEGIN
/* Hivatkozás rekord mezőjére minősítéssel */
v_Tetel1.kod := 1;
v_Tetel1.nev := 'alma';
v_Tetel1.nev := INITCAP(v_Tetel1.nev);
v_Kolcsonzes.konyv.id := fv(10, 15).konyv.id;
/* Megengedett az értékadás azonos típusok esetén. */
v_Tetel2 := v_Tetel1;
/* A szerkezeti egyezőség nem elég értékadásnál */
-- v_Bejegyzes := v_Tetel1; -- Hibás értékadás
/* Rekordok egyenlőségének összehasonlítása és NULL tesztelése
nem lehetséges */
/* A következő IF feltétele hibás
IF v_Tetel1 IS NULL
OR v_Tetel1 = v_Tetel2
OR v_Tetel1 <> v_Tetel2 THEN
NULL;
END IF;
*/
END;
/
```

4. Felhasználói altípusok

Ahogy azt már említettük, az altípus ugyanazokkal a műveletekkel és reprezentációval rendelkezik, mint alaptípusa, legfeljebb annak tartományát szűkítheti. Tehát az altípus nem *új* típus. Az előzőekben találkoztunk beépített altípusokkal. Most megnézzük, hogyan hozhat létre a felhasználó saját altípust. Erre szolgál a SUBTYPE deklarációs utasítás, amelyet egy blokk, alprogram vagy csomag deklarációs részében helyezhetünk el. Szintaxisa:

```
SUBTYPE altípus_név IS alaptípus_név[(korlát)] [NOT NULL];
```

ahol az *alaptípus_név* azon típusnak a neve, amelyből az altípust származtatjuk, *altípus_név* pedig az új altípus neve. A NOT NULL opció megadása esetén az altípus tartománya nem tartalmazza a NULL értéket.

A *korlát* hossz, pontosság és skála információkat adhat meg. Ugyanakkor a felhasználói altípus (a NULL-tól eltekintve) nemkorlátozott altípusként jön létre. Tehát az *altípus_név* deklarációban való szerepeltetésénél a *korlát* értékét meghaladó érték is megadható. BINARY_INTEGER alaptípus esetén létrehozhatunk korlátozott altípust is, a tartomány leszűkítésének alakja:

```
RANGE alsó_határ..felső_határ
```

A RANGE után egy érvényes intervallumnak kell állnia, ez lesz az altípus tartománya. Példák altípusok használatára:

```
SUBTYPE t_szam IS NUMBER;

SUBTYPE t_evszam IS NUMBER(4,0);

SUBTYPE t_tiz IS NUMBER(10);

SUBTYPE t_egy IS t_tiz(1); -- NUMBER(1)

SUBTYPE t_ezres IS t_tiz(10, -3); -- NUMBER(10, -3)

SUBTYPE t_nev IS VARCHAR2(40);

-- felüldefiniáljuk az előző hosszmegszorítást

SUBTYPE t_nev2 IS t_nev(50);

v_Tiz t_tiz := 12345678901; -- túl nagy az érték pontossága

v_Egy t_egy := 12345; -- túl nagy az érték pontossága

v_Ezres t_ezres := 12345; -- 12000 kerül tárolásra

SUBTYPE t_szo IS VARCHAR2;

v_Szo t_szo(10);

v_Szo2 t_szo; -- Hiba, nincs megadva a hossz
```

1. példa (Szabályos altípus deklarációk)

```
CREATE TABLE tab1 (

id NUMBER NOT NULL,

nev VARCHAR2(40)

)

/

DECLARE

-- rekordtípusú altípus

SUBTYPE t_tabsor IS tab1%ROWTYPE;

-- t_nev örökli a tab1.nev hosszmegszorítását

SUBTYPE t_nev IS tab1.nev%TYPE NOT NULL;

-- t_id nem örökli a tab1.id NOT NULL megszorítását,

-- mivel ez táblamegszorítás, nem a típushoz tartozik.

SUBTYPE t_id IS tab1.id%TYPE;
```

```
-- t_nev2 örökli a tab1.nev hosszmegszorítását és NOT NULL
-- megszorítását is, mert PL/SQL típus esetén a NOT NULL
-- is a típushoz tartozik.
SUBTYPE t_nev2 IS t_nev;
v_Id t_id; -- NUMBER típusú, NULL kezdőértékkel
v_Nev1 t_nev; -- hibás, fordítási hiba, NOT NULL típusnál
-- mindig kötelező az értékadás
v_Nev2 t_nev := 'XY'; -- így már jó
BEGIN
v_Id := NULL; -- megengedett, lehet NULL
v_Nev2 := NULL; -- hibás, NULL-t nem lehet értékül adni
END;
/
DECLARE
-- RANGE megszorítás altípusban
SUBTYPE t_szamjegy10 IS BINARY_INTEGER RANGE 0..9;
v_Szamjegy10 t_szamjegy10;
v_Szamjegy16 BINARY_INTEGER RANGE 0..15;
BEGIN
-- megengedett értékadások
v_Szamjegy10 := 9;
v_Szamjegy16 := 15;
-- hibás értékadások, mindkettő futási idejű hibát váltana ki
-- v_Szamjegy10 := 10;
-- v_Szamjegy16 := 16;
END;
/
```

2. példa (Szabálytalan altípus deklaráció)

```
-- nem lehet az altípusban megszorítás
SUBTYPE t_skala(skala) IS NUMBER(10,skala);
```

5. Adattípusok konverziója

A PL/SQL a skalártípusok családjai között értelmezi a konverziót. A családok között a PL/SQL automatikusan *implicit* konverziót alkalmaz, amennyiben az lehetséges. Természetesen ekkor a konvertálandó értéknek mindkét típusnál értelmezhetőnek kell lenni. Például a ‘123’ CHAR típusú érték gond nélkül konvertálható a 123 NUMBER típusú értékre.

A 3.3. táblázat a skalár alaptípusok közötti lehetséges implicit konverziókat szemlélteti.

PL/SQL alkalmazások írásánál javallott az explicit konverziós függvények használata. Az implicit konverzió ugyanis esetleges, verziófüggő, a szövegkörnyezet által meghatározott, tehát a kód nehezebben áttekinthető, kevésbé hordozható és karbantartható.

A 3.4. táblázat a leggyakoribb konverziós függvények rövid leírását adja.

3.3. táblázat - Implicit konverziók

	BIN_DOU	BIN_FLO	BIN_INT	BLOB	CHAR	CLOB	DATE	LONG	NUMB	PLS_INT	RAW	UROWID	VARCHAR2
BIN_DOU		X	X		X			X	X				X
BIN_FLO	X		X		X			X	X				X
BIN_INT	X	X			X			X	X	X			X
BLOB											X		
CHAR	X	X	X			X	X	X	X	X	X	X	X
CLOB					X								X
DATE					X			X					X
LONG	X	X			X						X		X
NUMBER	X	X	X		X			X		X			X
PLS_INT	X	X			X			X	X				X
RAW				X	X			X					X
UROWID					X								X
VARCHAR2	X	X	X		X	X	X	X	X	X	X	X	

3.4. táblázat - Konverziós függvények

Függvény	**Leírás**	**Konvertálható családok**
TO_CHAR	A megadott paramétert VARCHAR2 típusúra konvertálja, opcionálisan megadható a formátum.	Numerikus, dátum
TO_DATE	A megadott paramétert DATE típusúra	Karakteres

	konvertálja, opcionálisan megadható a formátum.	
TO_TIMESTAMP	A megadott paramétert TIMESTAMP típusúra konvertálja, opcionálisan megadható a formátum.	Karakteres
TO_TIMESTAMP_TZ	A megadott paramétert TIMESTAMP WITH TIMEZONE típusúra konvertálja, opcionálisan megadható a formátum.	Karakteres
TO_DSINTERVAL	A megadott paramétert INTERVAL DAY TO SECOND típusúra konvertálja, opcionálisan megadható a formátum.	Karakteres
TO_YMINTERVAL	A megadott paramétert INTERVAL YEAR TO MONTH típusúra konvertálja, opcionálisan megadható a formátum.	Karakteres
TO_NUMBER	A megadott paramétert NUMBER típusúra konvertálja, opcionálisan megadható a formátum.	Karakteres
TO_BINARY_DOUBLE	A megadott paramétert BINARY_DOUBLE típusúra konvertálja, opcionálisan megadható a formátum.	Karakteres, Numerikus
TO_BINARY_FLOAT	A megadott paramétert BINARY_FLOAT típusúra konvertálja, opcionálisan megadható a formátum.	Karakteres, Numerikus
RAWTOHEX	A paraméterként megadott bináris érték hexadecimális reprezentációját adja meg.	Raw
HEXTORAW	A paraméterként megadott hexadecimális reprezentációjú értéket a bináris formájában adja meg.	Karakteres (hexadecimális reprezentációt kell tartalmaznia).
CHARTOROWID	A karakteres reprezentációjával adott ROWID belső bináris alakját adja meg.	Karakteres (18-karakteres rowid formátumot kell tartalmaznia).
ROWIDTOCHAR	A paraméterként megadott belső bináris reprezentációjú ROWID külső karakteres reprezentációját adja meg.	Rowid

4. fejezet - Kifejezések

A kifejezés *operátorokból* és *operandusokból* áll. Az operandus lehet literál, nevesített konstans, változó és függvényhívás. Az operátorok *unárisok* (egyoperandusúak) vagy *binárisak* (kétoperandusúak) lehetnek. A PL/SQL-ben nincs ternáris operátor. A legegyszerűbb kifejezés egyetlen operandusból áll. A PL/SQL a kifejezések *infix* alakját használja.

A kifejezés *kiértékelése* az a folyamat, amikor az operandusok értékeit felhasználva az operátorok által meghatározott műveletek egy adott sorrendben végrehajtódnak és előáll egy adott típusú érték. A PL/SQL a procedurális nyelveknél megszokott módon a kiértékelést a *balról jobbra* szabály alapján a *precedenciatáblázat* felhasználásával hajtja végre. A PL/SQL precedenciatáblázata a 4.1. táblázatban látható. Itt hívjuk fel a figyelmet, hogy a precedenciatáblázatban nincs kötési irány! A PL/SQL azt mondja, hogy az azonos precedenciájú műveletek végrehajtási sorrendje tetszőleges (ennek következményeit az optimalizáló működésére *lásd* a 16. fejezetben).

4.1. táblázat - A PL/SQL precedenciatáblázata

Operátor
**, NOT
+, –
*, /
+, –, ||
=, !=, ~=, ^=, <>, <, >, <=, >=, IS NULL, LIKE, IN, BETWEEN
AND
OR

A kifejezés tetszőlegesen zárójelezhető. A zárójel az operátorok precedenciájának felülbírálására való. A zárójelezett részkifejezést előbb kell kiértékelni, mint az egyéb operandusokat.

Aritmetikai operátorok

Az aritmetikai operátorokkal a szokásos aritmetikai műveletek hajthatók végre, numerikus értékeken.

Operátor	Jelentése
**	hatványozás
+, –	unáris előjelek
*	szorzás
/	osztás
+	összeadás
–	kivonás

Karakteres operátorok

A PL/SQL-nek egyetlen karakteres operátora van, ez az összefűzés (konkatenáció) operátor: ||. Például:

```
'A' || 'PL/SQL' || 'alapján'
```

eredménye

```
'A PL/SQL alapján'
```

Ha az összefűzés mindkét operandusa CHAR típusú, az eredmény is az lesz. Ha valamelyik VARCHAR2 típusú, az eredmény is VARCHAR2 típusú lesz.

Hasonlító operátorok

Ezek az operátorok binárisak, két karakteres, numerikus vagy dátum típusú operandust hasonlítanak össze és mindig logikai értéket eredményeznek.

Operátor	Jelentése
=	egyenlő
!=, ~=, ^=, <>	nem egyenlő
<	kisebb
>	nagyobb
<=	kisebb egyenlő
>=	nagyobb egyenlő

Két numerikus operandus közül az a kisebb, amelynek az értéke kisebb. Két dátum közül a korábbi a kisebb. A karakteres típusú operandusok összehasonlításánál viszont az Oracle kétféle szemantikát használ, ezek: *szóköz-hozzáadásos, nem-szóköz-hozzáadásos*.

A szóköz-hozzáadásos szemantika algoritmusa a következő:

a. Ha a két karakterlánc különböző hosszúságú, akkor a rövidebbet jobbról kiegészítjük annyi szóközzel, hogy azonos hosszúságúak legyenek.

b. Összehasonlítjuk rendre a két karakterlánc elemeit, a legelső karaktertől indulva.

c. Ha azonosak, akkor tovább lépünk a következő karakterre. Ha nem azonosak, akkor az a kisebb, amelyiknek az adott karaktere kisebb (természetesen az alkalmazott kódtábla értelmében). Így a hasonlító operátornak megfelelő igaz vagy hamis érték megállapítható.

d. Ha a karakterláncok végére érünk, akkor azok egyenlők.

Ezen szemantika alapján a következő kifejezések értéke igaz lesz:

```
'abc' = 'abc'

'xy ' = 'xy'

'xy' < 'xyz'

'xyz' > 'xya'
```

A nem-szóköz-hozzáadásos szemantika algoritmusa a következő:

a. Összehasonlítjuk rendre a két karakterlánc elemeit, a legelső karaktertől indulva.

b. Ha azonosak, akkor továbblépünk a következő karakterre. Ha nem azonosak, akkor az a kisebb, amelyiknek az adott karaktere kisebb (az NLS_COMP és az NLS_SORT inicializációs paraméter beállításától függően). Így a hasonlító operátornak megfelelő igaz vagy hamis érték megállapítható.

c. Ha elérjük valamelyik karakterlánc végét úgy, hogy a másiknak még nincs vége, akkor az a kisebb.

d. Ha egyszerre érjük el mindkét karakterlánc végét, akkor azok egyenlők.

Ezen szemantika alapján a fenti kifejezések közül az alábbiak értéke most is igaz:

```
'abc' = 'abc'

'xy' < 'xyz'

'xyz' > 'xya'
```

de az

```
'xy ' = 'xy'
```

kifejezés értéke hamis.

A PL/SQL a szóköz-hozzáadásos szemantikát akkor használja, ha mindkét operandus fix hosszúságú, azaz CHAR vagy NCHAR típusú, egyébként a nem-szóköz-hozzáadásos algoritmus alapján jár el.

Példa

```
nev1 VARCHAR2(10) := 'Gábor';

nev2 CHAR(10) := 'Gábor'; -- a PL/SQL szóközöket ad hozzá
```

A következő IF feltétel hamis, mert nev2 a szóközöket is tartalmazza:

```
IF nev1 = nev2 THEN …
```

Ha ezen operátorok valamelyik operandusa NULL értékű, akkor az eredmény értéke is NULL lesz. Ismételten felhívjuk a figyelmet, hogy az üres sztring azonos a NULL értékkel a karakteres típusok esetén.

Ha egy adott értékről azt akarjuk eldönteni, hogy az NULL érték-e, akkor az IS NULL unáris postfix operátort kell használnunk. Ez igaz értéket szolgáltat, ha operandusa NULL, különben pedig hamisat.

A LIKE bináris operátor, amely egy karakterláncot hasonlít egy mintához. A hasonlításnál a kis- és nagybetűk különböznek. Igaz értéket ad, ha a minta illeszkedik a karakterláncra, hamisat, ha nem.

A mintában használhatunk két speciális, helyettesítő karaktert. Az aláhúzás (_) pontosan egy tetszőleges karakterrel helyettesíthető, a százalékjel (%) pedig akármennyivel (akár nulla darabbal is). Az alábbi kifejezések igaz értékűek:

```
'Almafa' LIKE 'A%a'

'Almafa' LIKE '%'

'Almafa' LIKE '_lm_f_'
```

A BETWEEN bináris operátor egyesíti a <= és >= operátorok hatását, második operandusa egy intervallumot ad meg

```
alsó_határ AND felső_határ
```

formában. Az operátor igaz értéket ad, ha az első operandus értéke eleme az intervallumnak, egyébként pedig hamisat.

A következő kifejezés értéke igaz:

```
11 BETWEEN 10 AND 20
```

Az IN bináris operátor második operandusa egy halmaz. Igaz értéket ad, ha az első operandus értéke eleme a halmaznak, egyébként pedig hamisat. A következő kifejezés értéke hamis:

```
'Piros' IN ('Fekete','Lila')
```

Logikai operátorok

Operátor	Jelentése
NOT	tagadás
AND	logikai és
OR	logikai vagy

A PL/SQL-ben háromértékű logika van (mivel a NULL a logikai típus tartományának is eleme). Az igazságtáblákat a 4.2. táblázat tartalmazza.

4.2. táblázat - Logikai igazságtáblák

NOT	igaz	hamis	NULL
	hamis	igaz	NULL

AND	igaz	hamis	NULL
igaz	igaz	hamis	NULL
hamis	hamis	hamis	hamis
NULL	NULL	hamis	NULL

OR	igaz	hamis	NULL
igaz	igaz	igaz	igaz
hamis	igaz	hamis	NULL
NULL	igaz	NULL	NULL

A PL/SQL a logikai operátorokat tartalmazó kifejezések kiértékelését *rövidzár* módon végzi, azaz csak addig értékeli ki a kifejezést, ameddig annak értéke el nem dől, és a hátralevő részkifejezésekkel nem foglalkozik tovább. Tekintsük a következő kifejezést:

```
(a=0) OR ((b/a)<5)
```

A rövidzár kiértékelés annyit jelent, hogy ha *a* értéke 0, akkor a kifejezés értéke már eldőlt (az OR miatt) és az igaz. Ha viszont nem rövidzár módon történne a kiértékelés, akkor az OR második operandusában *nullával való osztás* hiba lépne fel.

Feltételes kifejezések

Egy feltételes kifejezés *szelektort* használ arra, hogy a kifejezés lehetséges értékei közül egyet kiválasszon. Alakja:

```
CASE szelektor

WHEN kifejezés THEN eredmény

[WHEN kifejezés THEN eredmény]…

[ELSE eredmény]

END
```

A *szelektornak* (amely maga is kifejezés) a *kifejezés* típusával kompatibilis értéket kell szolgáltatnia.

A feltételes kifejezés kiértékelésénél kiértékelődik a *szelektor*, és az értéke a felsorolás sorrendjében rendre a WHEN utáni *kifejezések* értékéhez hasonlítódik. Ha van egyezés, akkor a feltételes kifejezés értéke a megfelelő THEN utáni *eredmény* lesz. Ha egyetlen WHEN utáni kifejezés értékével sincs egyezés és van ELSE ág, akkor az ELSE utáni *eredmény* lesz a feltételes kifejezés értéke. Ha pedig nincs ELSE, akkor NULL.

1. példa

```
DECLARE

v_Osztalyzat NUMBER(1);

v_Minosites VARCHAR2(10);

BEGIN

.

.

.

v_Minosites :=

CASE v_Osztalyzat

WHEN 5 THEN 'Jeles'

WHEN 4 THEN 'Jó'

WHEN 3 THEN 'Közepes'

WHEN 2 THEN 'Elégséges'

WHEN 1 THEN 'Elégtelen'

ELSE 'Nincs ilyen osztályzat'

END;

.

.

.

END;

/
```

A feltételes kifejezésnek létezik olyan alakja, ahol nincs szelektor és a WHEN után feltételek (logikai kifejezések) állnak. Ekkor az első olyan WHEN ág *eredménye* lesz a feltételes kifejezés értéke, amelyben a feltétel igaz. Az ELSE az előzőhöz hasonlóan működik.

2. példa

```
DECLARE

v_Osztalyzat NUMBER(1);

v_Minosites VARCHAR2(10);

BEGIN

.

.

.

v_Minosites :=

CASE

WHEN v_Osztalyzat = 5 THEN 'Jeles'

WHEN v_Osztalyzat = 4 THEN 'Jó'

WHEN v_Osztalyzat = 3 THEN 'Közepes'

WHEN v_Osztalyzat = 2 THEN 'Elégséges'

WHEN v_Osztalyzat = 1 THEN 'Elégtelen'

ELSE 'Nincs ilyen osztályzat'

END;

.

.

.

END;

/
```

5. fejezet - Végrehajtható utasítások

A PL/SQL végrehajtható utasításai arra valók, hogy velük a program tevékenységét (mint algoritmust) leírjuk.

1. Az üres utasítás

Az üres utasítás nem csinál semmi mást, mint átadja a vezérlést a következő utasításnak. A PL/SQL-ben az üres utasítás használható:

- elágaztató utasításokban (*lásd* 5.4. alfejezet);
- kivételkezelőben (*lásd* 7. fejezet);
- üres eljárástörzsben, top-down fejlesztésnél;
- általában mindenütt, ahol végrehajtható utasításnak kell állnia, de a szemantika az, hogy ne történjen semmi.

Alakja a következő:

```
NULL;
```

2. Az értékadó utasítás

Az értékadó utasítás egy változó, mező, kollekcióelem és objektumattribútum értékét állítja be. Az értéket mindig egy kifejezés segítségével adjuk meg. Alakja:

```
{kollekció_név[(index)]

|objektumnév[.attribútumnév]

|rekordnév[.mezőnév]

|változónév } := kifejezés;
```

Példák

```
DECLARE

-- kollekciótípus

TYPE t_szamok IS TABLE OF NUMBER

INDEX BY BINARY_INTEGER;

v_Szam NUMBER;

-- rekord típusú változó

v_Konyv konyv%ROWTYPE;

v_Konyv2 v_Konyv%TYPE;

v_KonyvId konyv.id%TYPE;

v_Szamok t_szamok;

BEGIN

v_Szam := 10.4;

v_Konyv.id := 15;

v_Konyv2 := v_Konyv; -- Rekordok értékadása
```

```
v_KonyvId := v_Konyv2.id; -- értéke 15

v_Szamok(4) := v_Szam;

END;

/
```

3. Az ugró utasítás

Az ugró utasítás segítségével a vezérlést átadhatjuk egy megcímkézett végrehajtható utasításra vagy blokkra. Alakja:

```
GOTO címke;
```

A GOTO utasítással nem lehet a vezérlést átadni egy feltételes utasítás valamelyik ágára, ciklus magjába vagy tartalmazott blokkba.

Egy blokk esetén GOTO segítségével átadható a vezérlés bármely végrehajtható utasításra vagy egy *tartalmazó* blokkba. Kivételkezelőből (*lásd* 7. fejezet) soha nem ugorhatunk vissza a kivételt kiváltó blokk utasításaira, csak tartalmazó blokkba.

Ha kurzor FOR ciklusban (*lásd* 8.5. alfejezet) alkalmazzuk a GOTO utasítást, akkor a kurzor automatikusan lezáródik.

A GOTO utasítás használata a PL/SQL programok írásánál nem szükséges. Használatát nem javasoljuk, mivel áttekinthetetlen, strukturálatlan, nehezen karbantartható, bonyolult szemantikájú kód létrehozását eredményezheti.

4. Elágaztató utasítások

Az elágaztató utasítások segítségével a programba olyan szerkezetek építhetők be, amelyek bizonyos tevékenységek alternatív végrehajtását teszik lehetővé.

4.1. A feltételes utasítás

A feltételes utasítás egymást kölcsönösen kizáró tevékenységek közül egy feltételsorozat alapján választ ki egyet végrehajtásra, vagy esetleg egyet sem választ. Alakja:

```
IF feltétel THEN utasítás [utasítás]…

[ELSIF feltétel THEN utasítás [utasítás]…]…

[ELSE utasítás [utasítás]…]

END IF;
```

A feltételes utasításnak három formája van: IF-THEN, IF-THEN-ELSE és IF-THEN-ELSIF.

A legegyszerűbb alak esetén a tevékenységet a THEN és az END IF alapszavak közé zárt utasítássorozat írja le. Ezek akkor hajtódnak végre, ha a feltétel értéke igaz. Hamis és NULL feltételértékek mellett az IF utasítás nem csinál semmit, tehát hatása megegyezik egy üres utasításéval.

Az IF-THEN-ELSE alak esetén az egyik tevékenységet a THEN és ELSE közötti, a másikat az ELSE és END IF közötti utasítássorozat adja. Ha a feltétel igaz, akkor a THEN utáni, ha hamis vagy NULL, akkor az ELSE utáni utasítássorozat hajtódik végre.

A harmadik alak egy feltételsorozatot tartalmaz. Ez a feltételsorozat a felírás sorrendjében értékelődik ki. Ha valamelyik igaz értékű, akkor az utána következő THEN-t követő utasítássorozat hajtódik végre. Ha minden feltétel hamis vagy NULL értékű, akkor az ELSE alapszót követő utasítássorozatra kerül a vezérlés, ha nincs ELSE rész, akkor ez egy üres utasítás.

Az IF utasítás esetén bármely tevékenység végrehajtása után (ha nem volt az utasítások között GOTO) a program az IF-et követő utasításon folytatódik.

A THEN és ELSE után álló utasítások között lehet újabb IF utasítás, az egymásba skatulyázás mélysége tetszőleges.

Példák

```
DECLARE

v_Nagyobb NUMBER;

x NUMBER;

y NUMBER;

z NUMBER;

BEGIN

.

.

.

v_Nagyobb := x;

IF x < y THEN

v_Nagyobb := y;

END IF;

.

.

.

v_Nagyobb := x;

IF x < y THEN

v_Nagyobb := y;

ELSE

DBMS_OUTPUT.PUT_LINE('x tényleg nem kisebb y-nál.');

END IF;

.

.

.

IF x < y THEN

v_Nagyobb := y;

ELSE

v_Nagyobb := x;

END IF;

.
```

```
.

.

IF x > y THEN

IF x > z THEN

v_Nagyobb := x;

ELSE

v_Nagyobb := z;

ELSE

IF y > z THEN

v_Nagyobb := y;

ELSE

v_Nagyobb := z;

END IF;

.

.

.

IF x < y THEN

DBMS_OUTPUT.PUT_LINE('x kisebb, mint y');

v_Nagyobb = x;

ELSIF x > y THEN

DBMS_OUTPUT.PUT_LINE('x nagyobb, mint y');

v_Nagyobb = y;

ELSE

DBMS_OUTPUT.PUT_LINE('x és y egyenlők');

v_Nagyobb = x; -- lehetne y is

END IF;

.

.

.

END;
```

4.2. A CASE utasítás

A CASE egy olyan elágaztató utasítás, ahol az egymást kölcsönösen kizáró tevékenységek közül egy kifejezés értékei, vagy feltételek teljesülése szerint lehet választani. Az utasítás alakja:

```
CASE [szelektor_kifejezés]

WHEN {kifejezés | feltétel} THEN utasítás [utasítás]…

[WHEN {kifejezés | feltétel} THEN utasítás [utasítás]…]…
```

```
[ELSE utasítás [utasítás]…]

END CASE;
```

Ha a CASE utasítás címkézett, az adott címke az END CASE után feltüntethető.

Tehát egy CASE utasítás tetszőleges számú WHEN ágból és egy opcionális ELSE ágból áll. Ha a *szelektor_kifejezés* szerepel, akkor a WHEN ágakban *kifejezés* áll, ha nem szerepel, akkor *feltétel*.

Működése a következő:

Ha szerepel *szelektor_kifejezés*, akkor ez kiértékelődik, majd az értéke a felírás sorrendjében hasonlításra kerül a WHEN ágak kifejezéseinek értékeivel. Ha megegyezik valamelyikkel, akkor az adott ágban a THEN után megadott utasítássorozat hajtódik végre, és ha nincs GOTO, akkor a működés folytatódik a CASE utasítást követő utasításon.

Ha a *szelektor_kifejezés* értéke nem egyezik meg egyetlen kifejezés értékével sem és van ELSE ág, akkor végrehajtódnak az abban megadott utasítások, és ha nincs GOTO, akkor a működés folytatódik a CASE utasítást követő utasításon. Ha viszont nincs ELSE ág, akkor a CASE_NOT_FOUND kivétel váltódik ki.

Ha a CASE alapszó után nincs megadva *szelektor_kifejezés*, akkor a felírás sorrendjében sorra kiértékelődnek a *feltételek* és amelyik igaz értéket vesz fel, annak a WHEN ága kerül kiválasztásra. A szemantika a továbbiakban azonos a fent leírtakkal.

Példák

```
/* Case 1 - szelektor_kifejezés van, az első egyező értékű ág fut le,

az ágakban tetszőleges értékű kifejezés szerepelhet.

*/

DECLARE

v_Allat VARCHAR2(10);

BEGIN

v_Allat := 'hal';

CASE v_Allat || 'maz'

WHEN 'halló' THEN

DBMS_OUTPUT.PUT_LINE('A halló nem is állat.');

WHEN SUBSTR('halmazelmélet', 1, 6) THEN

DBMS_OUTPUT.PUT_LINE('A halmaz sem állat.');

WHEN 'halmaz' THEN

DBMS_OUTPUT.PUT_LINE('Ez már nem fut le.');

ELSE

DBMS_OUTPUT.PUT_LINE('Most ez sem fut le.');

END CASE;

END;

/

/* Case 2 - szelektor_kifejezés van, nincs egyező ág, nincs ELSE */

BEGIN
```

```
CASE 2
WHEN 1 THEN
DBMS_OUTPUT.PUT_LINE('2 = 1');
WHEN 1+2 THEN
DBMS_OUTPUT.PUT_LINE('2 = 1 + 2 = ' || (1+2));
END CASE;
-- kivétel: ORA-06592 , azaz CASE_NOT_FOUND
END;
/
/* Case 3 - A case utasítás cimkézhető. */
BEGIN
-- A case ágai cimkézhetők
<<elso_elagazas>>
CASE 1
WHEN 1 THEN
<<masodik_elagazas>>
CASE 2
WHEN 2 THEN
DBMS_OUTPUT.PUT_LINE('Megtaláltuk.');
END CASE masodik_elagazas;
END CASE elso_elagazas;
END;
/
/* Case 4 - Nincs szelektor_kifejezés, az ágakban feltétel szerepel.*/
DECLARE
v_Szam NUMBER;
BEGIN
v_Szam := 10;
CASE
WHEN v_Szam MOD 2 = 0 THEN
DBMS_OUTPUT.PUT_LINE('Páros.');
WHEN v_Szam < 5 THEN
DBMS_OUTPUT.PUT_LINE('Kisebb 5-nél.');
WHEN v_Szam > 5 THEN
DBMS_OUTPUT.PUT_LINE('Nagyobb 5-nél.');
ELSE
```

```
DBMS_OUTPUT.PUT_LINE('Ez csak az 5 lehet.');

END CASE;

END;

/
```

5. Ciklusok

A ciklusok olyan programeszközök, amelyek egy adott tevékenység tetszés szerinti (adott esetben akár nullaszoros) ismétlését teszik lehetővé. Az ismétlődő tevékenységet egy végrehajtható utasítássorozat írja le, ezt az utasítássorozatot a ciklus *magjának* nevezzük.

A PL/SQL négyfajta ciklust ismer, ezek a következők:

- alapciklus (vagy végtelen ciklus);
- WHILE ciklus (vagy előfeltételes ciklus);
- FOR ciklus (vagy előírt lépésszámú ciklus);
- kurzor FOR ciklus.

A negyedik fajta ciklust a 8. fejezetben tárgyaljuk.

A ciklusmag ismétlődésére vonatkozó információkat (amennyiben vannak) a mag előtt, a ciklus *fejében* kell megadni. Ezek az információk az adott ciklusfajtára nézve egyediek. Egy ciklus a működését mindig csak a mag első utasításának végrehajtásával kezdheti meg. Egy ciklus befejeződhet, ha

- az ismétlődésre vonatkozó információk ezt kényszerítik ki;
- a GOTO utasítással kilépünk a magból;
- az EXIT utasítással befejeztetjük a ciklust;
- kivétel (*lásd* 7. fejezet) váltódik ki.

5.1. Alapciklus

Az alapciklus alakja a következő:

```
[címke] LOOP utasítás [utasítás]…

END LOOP [címke];
```

Az alapciklusnál nem adunk információt az ismétlődésre vonatkozóan, tehát ha a magban nem fejeztetjük be a másik három utasítás valamelyikével, akkor végtelenszer ismétel.

Példa

```
SET SERVEROUTPUT ON SIZE 10000;

/* Loop 1 - egy végtelen ciklus */

BEGIN

LOOP

DBMS_OUTPUT.PUT_LINE('Ez egy végtelen ciklus.');

-- A DBMS_OUTPUT puffere szab csak határt ennek a ciklusnak.

END LOOP;
```

```
END;

/

/* Loop 2 - egy látszólag végtelen ciklus */

DECLARE

v_Faktorialis NUMBER(5);

i PLS_INTEGER;

BEGIN

i := 1;

v_Faktorialis := 1;

LOOP

-- Előbb-utóbb nem fér el a szorzat 5 számjegyen.

v_Faktorialis := v_Faktorialis * i;

i := i + 1;

END LOOP;

EXCEPTION

WHEN VALUE_ERROR THEN

DBMS_OUTPUT.PUT_LINE(v_Faktorialis

|| ' a legnagyobb, legfeljebb 5-jegyű faktoriális.');

END;

/
```

5.2. WHILE ciklus

A WHILE ciklus alakja a következő:

```
[címke] WHILE feltétel

LOOP utasítás [utasítás]…

END LOOP [címke];
```

Ennél a ciklusfajtánál az ismétlődést egy *feltétel* szabályozza. A ciklus működése azzal kezdődik, hogy kiértékelődik a feltétel. Ha értéke igaz, akkor lefut a mag, majd újra kiértékelődik a feltétel. Ha a feltétel értéke hamis vagy NULL lesz, akkor az ismétlődés befejeződik, és a program folytatódik a ciklust követő utasításon.

A WHILE ciklus működésének két szélsőséges esete van. Ha a feltétel a legelső esetben hamis vagy NULL, akkor a ciklusmag egyszer sem fut le *(üres ciklus)*. Ha a feltétel a legelső esetben igaz és a magban nem történik valami olyan, amely ezt az értéket megváltoztatná, akkor az ismétlődés nem fejeződik be *(végtelen ciklus)*.

Példák

```
/* While 1 - eredménye megegyezik Loop 2 eredményével. Nincs kivétel. */

DECLARE

v_Faktorialis NUMBER(5);

i PLS_INTEGER;
```

```
BEGIN
i := 1;
v_Faktorialis := 1;
WHILE v_Faktorialis * i < 10**5 LOOP
v_Faktorialis := v_Faktorialis * i;
i := i + 1;
END LOOP;
DBMS_OUTPUT.PUT_LINE(v_Faktorialis
|| ' a legnagyobb, legfeljebb 5-jegyű faktoriális.');
END;
/
/* While 2 - üres és végtelen ciklus. */
DECLARE
i PLS_INTEGER;
BEGIN
i := 1;
/* Üres ciklus */
WHILE i < 1 LOOP
i := i + 1;
END LOOP;
/* Végtelen ciklus */
WHILE i < 60 LOOP
i := (i + 1) MOD 60;
END LOOP;
END;
/
```

Feltételezve, hogy a magjuk azonos, a következő két ciklus működése ekvivalens:

WHILE TRUE LOOP		LOOP
⋮		⋮
END LOOP;		END LOOP;

5.3. FOR ciklus

Ezen ciklusfajta egy egész tartomány minden egyes értékére lefut egyszer. Alakja:

```
[címke] FOR ciklusváltozó IN [REVERSE] alsó_határ..felső_határ
```

```
LOOP utasítás [utasítás]...

END LOOP [címke];
```

A *ciklusváltozó (ciklusindex, ciklusszámláló)* implicit módon PLS_INTEGER típusúnak deklarált változó, amelynek hatásköre a ciklusmag. Ez a változó rendre felveszi az *alsó_határ* és *felső_határ* által meghatározott egész tartomány minden értékét és minden egyes értékére egyszer lefut a mag. Az *alsó_határ* és *felső_határ* egész értékű kifejezés lehet. A kifejezések egyszer, a ciklus működésének megkezdése előtt értékelődnek ki.

A REVERSE kulcsszó megadása esetén a ciklusváltozó a tartomány értékeit *csökkenően*, annak hiányában *növekvően* veszi fel. Megjegyzendő, hogy REVERSE megadása esetén is a tartománynak az alsó határát kell először megadni:

```
-- Ciklus 10-től 1-ig

FOR i IN REVERSE 1..10 LOOP …
```

A ciklusváltozónak a ciklus magjában nem lehet értéket adni, csak az aktuális értékét lehet felhasználni kifejezésben.

Ha az *alsó_határ* nagyobb, mint a *felső_határ,* a ciklus egyszer sem fut le (üres ciklus). A FOR ciklus nem lehet végtelen ciklus.

Példák

```
/* For 1 - Egy egyszerű FOR ciklus */

DECLARE

v_Osszeg PLS_INTEGER;

BEGIN

v_Osszeg := 0;

FOR i IN 1..100 LOOP

v_Osszeg := v_Osszeg + i;

END LOOP;

DBMS_OUTPUT.PUT_LINE('Gauss már gyerekkorában könnyen kiszámolta, hogy');

DBMS_OUTPUT.PUT_LINE(' 1 + 2 + ... + 100 = ' || v_Osszeg || '.');

END;

/

/* For 2 - REVERSE használata

BEGIN

DBMS_OUTPUT.PUT_LINE('Gauss egymás alá írta a számokat (most csak 1-10):');

FOR i IN 1..10 LOOP

DBMS_OUTPUT.PUT(RPAD(i, 4));

END LOOP;

DBMS_OUTPUT.NEW_LINE;

FOR i IN REVERSE 1..10 LOOP

DBMS_OUTPUT.PUT(RPAD(i, 4));

END LOOP;
```

```
DBMS_OUTPUT.NEW_LINE;
FOR i IN REVERSE 1..10 LOOP
DBMS_OUTPUT.PUT(RPAD('--', 4));
END LOOP;
DBMS_OUTPUT.NEW_LINE;
FOR i IN REVERSE 1..10 LOOP
DBMS_OUTPUT.PUT(RPAD(11, 4));
END LOOP;
DBMS_OUTPUT.NEW_LINE;
END;
/
/* For 3 - "FOR Elise" - Egy kis esettanulmány */
<<cimke>>
DECLARE
i VARCHAR2(10) := 'Almafa';
j INTEGER;
BEGIN
/* A példát úgy próbálja ki, hogy j-re nézve mindig csak egy értékadás
történjen. Ekkor igaz lesz az értékadás előtti megjegyzés.
*/
-- A határok egyike sem lehet NULL, a ciklusfejben j kiértékelése
-- VALUE_ERROR kivételt eredményez.
-- Az alsó határ nagyobb a felső határnál, üres a tartomány,
-- a ciklusmag nem fut le egyszer sem.
j := 0;
-- Az alsó es a felső határ megegyezik, a ciklusmag egyszer fut le.
j := 1;
-- Az alsó határ kisebb a felső határnál,
-- a ciklusmag jelen esetben j-1+1 = 10-szer fut le.
j := 10;
-- Túl nagy szám. A PLS_INTEGER tartományába nem fér bele,
-- így kivételt kapunk a ciklusfejben
j := 2**32;
-- Az i ciklusváltozó implicit deklarációja elfedi
-- az i explicit deklarációját
FOR i IN 1..j
```

```
LOOP
-- a ciklusváltozó szerepelhet kifejezésben
DBMS_OUTPUT.PUT_LINE(i*2);
-- hibás, a ciklusváltozó nevesített konstansként viselkedik
i := i + 1;
-- Az elfedett változóra minősítéssel hivatkozhatunk.
DBMS_OUTPUT.PUT_LINE(cimke.i);
cimke.i := 'Körtefa'; -- szabályos, ez a blokk elején deklarált i
END LOOP;
-- Az explicit módon deklarált i újra elérhető minősítés nélkül
DBMS_OUTPUT.PUT_LINE(i);
END;
/
```

5.4. Az EXIT utasítás

Az EXIT utasítás bármely ciklus magjában kiadható, de ciklusmagon kívül nem használható. Hatására a ciklus befejezi a működését. Alakja:

```
EXIT [címke] [WHEN feltétel];
```

Az EXIT hatására a ciklus működése befejeződik és a program a követő utasításon folytatódik.

1. példa

```
/* Exit 1 - az EXIT hatására a ciklus befejeződik. */
DECLARE
v_Osszeg PLS_INTEGER;
i PLS_INTEGER;
BEGIN
i := 1;
LOOP
v_Osszeg := v_Osszeg + i;
IF v_Osszeg >= 100 THEN
EXIT; -- elértük a célt.
END IF;
i := i+1;
END LOOP;
DBMS_OUTPUT.PUT_LINE(i
|| ' az első olyan n szám, melyre 1 + 2 + … + n >= 100.');
END;
```

```
/
```

Az EXIT *címke* utasítással az egymásba skatulyázott ciklusok sorozatát fejeztetjük be a megadott címkéjű ciklussal bezárólag.

2. példa

```
/* Exit 2 - A címke segítségével nem csak a belső ciklus fejezhető be */

DECLARE

v_Osszeg PLS_INTEGER := 0;

BEGIN

<<kulso>>

FOR i IN 1..100 LOOP

FOR j IN 1..i LOOP

v_Osszeg := v_Osszeg + i;

-- Ha elértük a keresett összeget, mindkét ciklusból kilépünk.

EXIT kulso WHEN v_Osszeg > 100;

END LOOP;

END LOOP;

DBMS_OUTPUT.PUT_LINE(v_Osszeg);

END;

/
```

A WHEN utasításrész a ciklus feltételes befejeződését eredményezi. Az ismétlődés csak a *feltétel* igaz értéke mellett nem folytatódik tovább.

3. példa

```
/* Exit 3 - EXIT WHEN - IF - GOTO */

DECLARE

v_Osszeg PLS_INTEGER;

i PLS_INTEGER;

BEGIN

i := 1;

v_Osszeg := 0;

LOOP

v_Osszeg := v_Osszeg + i;

-- 1.

EXIT WHEN v_Osszeg >= 100;

-- 2.

IF v_Osszeg >= 100 THEN

EXIT;
```

```
END IF;

-- 3.

IF v_Osszeg >= 100 THEN

GOTO ki;

i := i+1;

END LOOP;

<<ki>>

DBMS_OUTPUT.PUT_LINE(i

|| ' az első olyan n szám, melyre 1 + 2 + … + n >= 100.');

END;

/
```

6. SQL utasítások a PL/SQL-ben

Egy PL/SQL program szövegébe *közvetlenül* csak az SQL DML és tranzakcióvezérlő utasításai építhetők be, a DDL utasítások nem. A DML és tranzakcióvezérlő utasítások bárhol használhatók, ahol végrehajtható utasítások állhatnak. A PL/SQL egy külön eszközt használ arra, hogy *minden* SQL utasítást alkalmazhassunk egy PL/SQL programban, ez a *natív dinamikus SQL*, amelyet a 15. fejezetben ismertetünk. Most a *statikus* SQL lehetőségeit tárgyaljuk. Ezen utasítások szövege a fordításkor ismert, a PL/SQL fordító ugyanúgy kezeli, fordítja őket, mint a procedurális utasításokat.

6.1. DML utasítások

A PL/SQL-ben a SELECT, DELETE, INSERT, UPDATE, MERGE utasítások speciális változatai használhatók. Mi most a hangsúlyt a speciális jellemzőkre helyezzük, az utasítások részletei megtalálhatók a referenciákban (*lásd* [8] és [21]).

SELECT INTO

A SELECT INTO utasítás egy vagy több adatbázistáblát kérdez le és a származtatott értékeket változókba vagy egy rekordba helyezi el. Alakja:

```
SELECT [{ALL|{DISTINCT|UNIQUE}}]

{*|select_kifejezés[,select_kifejezés]…}

{INTO {változónév[,változónév]…|rekordnév}|

BULK COLLECT INTO kollekciónév[,kollekciónév]…}

FROM {táblahivatkozás|

(alkérdés)|

TABLE(alkérdés1)} [másodlagos_név]

[,{táblahivatkozás|

(alkérdés)|

TABLE(alkérdés1)} [másodlagos_név]}]…

további_utasításrészek;

select_kifejezés:

{NULL|literál|függvényhívás |
```

```
szekvencianév.{CURRVAL|NEXTVAL}|

[sémanév.]{táblanév|nézetnév}.]oszlopnév}

[ [AS] másodlagos_név]

táblahivatkozás:

[sémanév.]{táblanév|nézetnév][@ab_kapcsoló]
```

Az INTO utasításrészben minden *select_kifejezéshez* meg kell adni egy típuskompatibilis változót, vagy pedig a *rekordnak* rendelkeznie kell ilyen mezővel. A BULK COLLECT INTO utasításrészt a 12. fejezetben tárgyaljuk.

A FROM utasításrészben az *alkérdés* egy INTO utasításrészt nem tartalmazó SELECT. A TABLE egy operátor (*lásd* 12. fejezet), amelynél az *alkérdés1* egy olyan SELECT, amely egyetlen oszlopértéket szolgáltat, de az beágyazott tábla vagy dinamikus tömb típusú. Tehát itt egy kollekciót és nem egy skalár értéket kell kezelnünk.

A *további_utasításrészek* a SELECT utasítás FROM utasításrésze után szabályosan elhelyezhető utasításrészek lehetnek (WHERE, GROUP BY, ORDER BY).

A BULK COLLECT utasításrészt nem tartalmazó SELECT utasításnak pontosan egy sort kell leválogatnia. Ha az eredmény egynél több sorból áll, a TOO_MANY_ROWS kivétel váltódik ki. Ha viszont egyetlen sort sem eredményez, akkor a NO_DATA_FOUND kivétel következik be.

1. példa

```
DECLARE

v_Ugyfel ugyfel%ROWTYPE;

v_Id ugyfel.id%TYPE;

v_Info VARCHAR2(100);

BEGIN

SELECT *

INTO v_Ugyfel

FROM ugyfel

WHERE id = 15;

SELECT id, SUBSTR(nev, 1, 40) || ' - ' || tel_szam

INTO v_Id, v_Info

FROM ugyfel

WHERE id = 15;

END;

/
```

2. példa

```
DECLARE

v_Ugyfel ugyfel%ROWTYPE;

BEGIN

SELECT *

INTO v_Ugyfel
```

```
FROM ugyfel
WHERE id = -1;
END;
/
/*
Eredmény:
DECLARE
*
Hiba a(z) 1. sorban:
ORA-01403: nem talált adatot
ORA-06512: a(z) helyen a(z) 4. sornál
*/
DECLARE
v_Ugyfel ugyfel%ROWTYPE;
BEGIN
SELECT *
INTO v_Ugyfel
FROM ugyfel;
END;
/
/*
Eredmény:
DECLARE
*
Hiba a(z) 1. sorban:
ORA-01422: a pontos lehívás (FETCH) a kívántnál több sorral tér vissza
ORA-06512: a(z) helyen a(z) 4. sornál
*/
```

DELETE

A DELETE utasítás egy adott tábla vagy nézet sorait törli. Alakja:

```
DELETE [FROM] {táblahivatkozás|
(alkérdés)|
TABLE(alkérdés1)}
[másodlagos_név]
[WHERE {feltétel|CURRENT OF kurzornév}]
```

```
[returning_utasításrész];

returning_utasításrész:

RETURNING {egysoros_select_kifejezés[,egysoros_select_kifejezés]…

INTO változó[,változó]…|

többsoros_select_kifejezés[,többsoros_select_kifejezés]…

[BULK COLLECT INTO kollekciónév[,kollekciónév]…
```

A FROM utasításrész elemei megegyeznek a SELECT utasítás FROM utasításrészének elemeivel. A táblából vagy nézetből csak azok a sorok törlődnek, amelyekre igaz a *feltétel*. A CURRENT OF utasításrész esetén a megadott kurzorhoz rendelt lekérdezésnek rendelkeznie kell a FOR UPDATE utasításrésszel és ekkor a legutolsó FETCH által betöltött sor törlődik.

A *returning_utasításrész* segítségével a törölt sorok alapján számított értékek kaphatók vissza. Ekkor nem kell ezeket az értékeket a törlés előtt egy SELECT segítségével származtatni. Az értékek változókban, rekordban vagy kollekciókban tárolhatók. A BULK COLLECT utasításrészt a 12. fejezet tárgyalja.

Példa

```
DECLARE

v_Datum kolcsonzes.datum%TYPE;

v_Hosszabbitva kolcsonzes.hosszabbitva%TYPE;

BEGIN

DELETE FROM kolcsonzes

WHERE kolcsonzo = 15

AND konyv = 20

RETURNING datum, hosszabbitva

INTO v_Datum, v_Hosszabbitva;

END;

/
```

INSERT

Az INSERT utasítás új sorokkal bővít egy megadott táblát vagy nézetet. Alakja:

```
INSERT INTO {táblahivatkozás|

(alkérdés)|

TABLE(alkérdés1)}

[másodlagos_név]

[(oszlop[,oszlop]…)]

{{VALUES(sql_kifejezés[,sql_kifejezés]…)|rekord}

[returning_utasításrész]|

alkérdés2};
```

A *returning_utasításrész* magyarázatát lásd a DELETE utasításnál.

Az *oszlop* egy adatbázisbeli tábla vagy nézet oszlopneve. Az oszlopok sorrendje tetszőleges, nem kell hogy a CREATE TABLE vagy CREATE VIEW által definiált sorrendet kövesse, viszont minden oszlopnév csak egyszer fordulhat elő. Ha a tábla vagy nézet nem minden oszlopa szerepel, akkor a hiányzók NULL értéket kapnak, vagy a megfelelő CREATE utasításban megadott alapértelmezett értéket veszik fel.

A VALUES utasításrész a megadott oszlopokhoz rendel értékeket SQL kifejezések segítségével. Ha az oszloplista hiányzik, akkor a CREATE utasításban megadott oszlopsorrendben történik az értékhozzárendelés. A kifejezések típusának kompatibilisnek kell lenni a megfelelő oszlopok típusával. Minden oszlophoz pontosan egy értéket kell rendelni, ha az oszloplista szerepel. Teljes sornak tudunk értéket adni egy típuskompatibilis *rekord* segítségével.

Az *alkérdés2* egy olyan SELECT, amelynek eredményeképpen az oszlopok értéket kapnak. Annyi értéket kell szolgáltatnia, ahány oszlop meg van adva, vagy pedig az oszloplista hiányában annyit, ahány oszlopa van a táblának vagy nézetnek. A tábla vagy nézet annyi sorral bővül, ahány sort a lekérdezés visszaad.

Példa

```
DECLARE

v_Kolcsonzes kolcsonzes%ROWTYPE;

BEGIN

INSERT INTO kolcsonzes (kolcsonzo, konyv, datum)

VALUES (15, 20, SYSDATE)

RETURNING kolcsonzo, konyv, datum, hosszabbitva, megjegyzes

INTO v_Kolcsonzes;

-- rekord használata

v_Kolcsonzes.kolcsonzo := 20;

v_Kolcsonzes.konyv := 25;

v_Kolcsonzes.datum := SYSDATE;

v_Kolcsonzes.hosszabbitva := 0;

INSERT INTO kolcsonzes VALUES v_Kolcsonzes;

END;

/
```

UPDATE

Az UPDATE utasítás megváltoztatja egy megadott tábla vagy nézet megadott oszlopainak értékét. Alakja:

```
UPDATE {táblahivatkozás|

(alkérdés)|

TABLE(alkérdés1)}

[másodlagos_név]

SET{{oszlop={sql_kifejezés|(alkérdés2)}|

(oszlop[,oszlop]…)=(alkérdés3)}

[,{oszlop={sql_kifejezés|(alkérdés2)}|

(oszlop[,oszlop]…)=(alkérdés3)}]…

|ROW=rekord}
```

```
[WHERE {feltétel|CURRENT OF kurzornév}]

[returning_utasításrész];
```

A SET *oszlop=sql_kifejezés* az oszlopok új értékét egyenként adja meg SQL kifejezések segítségével. A SET *oszlop=(alkérdés2)* esetén az *alkérdés2* egy olyan SELECT, amely pontosan egyetlen sort és egyetlen oszlopot ad vissza, ez határozza meg az *oszlop* új értékét. Ha oszloplistát adunk meg, akkor az *alkérdés3* egy olyan SELECT, amely pontosan egy sort és annyi oszlopot eredményez, ahány elemű az oszloplista. A lekérdezés eredménye az oszloplista sorrendjében írja fölül az oszlopok értékét. A ROW megadása esetén a sornak egy típuskompatibilis *rekord* segítségével tudunk új értéket adni.

A többi utasításrész leírását lásd a DELETE utasításnál.

Példa

```
DECLARE

TYPE t_id_lista IS TABLE OF NUMBER;

v_Kolcsonzes kolcsonzes%ROWTYPE;

BEGIN

UPDATE TABLE(SELECT konyvek

FROM ugyfel

WHERE id = 20)

SET datum = datum + 1

RETURNING konyv_id BULK COLLECT INTO v_Id_lista;

SELECT *

INTO v_Kolcsonzes

FROM kolcsonzes

WHERE kolcsonzo = 25

AND konyv = 35;

v_Kolcsonzes.datum := v_Kolcsonzes.datum+1;

v_Kolcsonzes.hosszabbitva := v_Kolcsonzes.hosszabbitva+1;

-- rekord használata

UPDATE kolcsonzes

SET ROW = v_Kolcsonzes

WHERE kolcsonzo = v_Kolcsonzes.kolcsonzo

AND konyv = v_Kolcsonzes.konyv;

END;

/
```

MERGE

A MERGE utasítás a többszörös INSERT és DELETE utasítások elkerülésére való. Alakja:

```
MERGE INTO tábla [másodlagos_név]

USING {tábla|nézet|alkérdés} [másodlagos_név] ON (feltétel)
```

```
WHEN MATCHED THEN UPDATE SET oszlop={kifejezés|DEFAULT}

[,oszlop={kifejezés|DEFAULT}]...

WHEN NOT MATCHED THEN INSERT(oszlop[,oszlop]...)

VALUES ({DEFAULT|kifejezés[,kifejezés]...});
```

Az INTO határozza meg a céltáblát, amelyet bővíteni vagy módosítani akarunk.

A USING adja meg az adatok forrását, amely tábla, nézet vagy egy alkérdés lehet.

Az ON utasításrészben megadott *feltétel* szolgál a beszúrás és módosítás vezérlésére. Minden olyan céltáblasor, amelyre igaz a feltétel, a forrásadatoknak megfelelően módosul. Ha valamelyik sorra a feltétel nem teljesül, az Oracle beszúrást végez a forrásadatok alapján.

A WHEN MATCHED utasításrész a céltábla új oszlopértékét határozza meg. Ez a rész akkor hajtódik végre, ha a *feltétel* igaz. A WHEN NOT MATCHED utasításrész megadja a beszúrandó sor oszlopértékét, ha a *feltétel* hamis.

Példa

```
/*

Szinkronizáljuk a kolcsonzes táblát a 20-as azonosítójú ügyfél esetén

az ugyfel konyvek oszlopához.

*/

DECLARE

c_Uid CONSTANT NUMBER := 20;

BEGIN

MERGE INTO kolcsonzes k

USING (SELECT u.id, uk.konyv_id, uk.datum

FROM ugyfel u, TABLE(u.konyvek) uk

WHERE id = c_Uid) s

ON (k.kolcsonzo = c_Uid AND k.konyv = s.konyv_id)

WHEN MATCHED THEN UPDATE SET k.datum = s.datum

WHEN NOT MATCHED THEN INSERT (kolcsonzo, konyv, datum)

VALUES (c_Uid, s.konyv_id, s.datum);

END;

/
```

6.2. Tranzakcióvezérlés

Az Oracle működése közben *munkameneteket* kezel. Egy *felhasználói munkamenet* egy alkalmazás vagy egy Oracle-eszköz elindításával, az Oracle-hez való *kapcsolódással* indul. A munkamenetek egyidejűleg, egymással párhuzamosan, az erőforrásokat megosztva működnek. Az *adatintegritás* megőrzéséhez (ahhoz, hogy az adatok változásának sorrendje érvényes legyen) az Oracle megfelelő *konkurenciavezérlést* alkalmaz.

Az Oracle *zárakat* használ az adatok konkurens elérésének biztosítására. A zár átmeneti tulajdonosi jogkört biztosít a felhasználó számára olyan adatbázisbeli objektumok fölött, mint például egy teljes tábla vagy egy tábla bizonyos sorai. Más felhasználó nem módosíthatja az adatokat mindaddig, amíg a zárolás fennáll. Az Oracle automatikus zárolási mechanizmussal rendelkezik, de a felhasználó explicit módon is zárolhat.

Ha egy táblát az egyik felhasználó éppen lekérdez, egy másik pedig módosít, akkor a módosítás előtti adatokat *visszagörgető szegmensekben* tárolja, ezzel biztosítva az olvasási konzisztenciát.

A *tranzakció* nem más, mint DML-utasítások sorozata, amelyek a munka egyik logikai egységét alkotják. A tranzakció utasításainak hatása együtt jelentkezik. A tranzakció sikeres végrehajtása esetén a módosított adatok *véglegesítődnek*, a tranzakcióhoz tartozó visszagörgetési szegmensek újra felhasználhatóvá válnak. Ha viszont valamilyen hiba folytán a tranzakció sikertelen (bármelyik utasítása nem hajtható végre), akkor *visszagörgetődik*, és az adatbázis tranzakció előtti állapota nem változik meg. Az Oracle lehetőséget biztosít egy tranzakció részleges visszagörgetésére is.

Minden SQL utasítás egy tranzakció része. A tranzakciót az első SQL utasítás indítja el. Ha egy tranzakció befejeződött, a következő SQL utasítás új tranzakciót indít el.

A tranzakció explicit véglegesítésére a COMMIT utasítás szolgál, amelynek alakja:

```
COMMIT [WORK];
```

A WORK alapszó csak az olvashatóságot szolgálja, nincs szemantikai jelentése.

A COMMIT a tranzakció által okozott módosításokat átvezeti az adatbázisba és láthatóvá teszi azokat más munkamenetek számára, felold minden – a tranzakció működése közben elhelyezett – zárat és törli a mentési pontokat.

A SAVEPOINT utasítással egy tranzakcióban *mentési pontokat* helyezhetünk el. Ezek a tranzakció részleges visszagörgetését szolgálják. Az utasítás alakja:

```
SAVEPOINT név;
```

A *név* nemdeklarált azonosító, amely a tranzakció adott pontját jelöli meg. A *név* egy másik SAVEPOINT utasításban felhasználható. Ekkor a később kiadott utasítás hatása lesz érvényes.

A visszagörgetést a ROLLBACK utasítás végzi, alakja:

```
ROLLBACK [WORK] [TO [SAVEPOINT] mentési_pont];
```

A WORK és a SAVEPOINT nem bír szemantikai jelentéssel.

Az egyszerű ROLLBACK utasítás érvényteleníti a teljes tranzakció hatását (az adatbázis változatlan marad), oldja a zárakat és törli a mentési pontokat. A tranzakció befejeződik.

A TO utasításrésszel rendelkező ROLLBACK a megadott mentési pontig görgeti vissza a tranzakciót, a megadott mentési pont érvényben marad, az azt követők törlődnek, a mentési pont után elhelyezett zárak feloldásra kerülnek és a tranzakció a megadott mentési ponttól folytatódik.

Az Oracle minden INSERT, UPDATE, DELETE utasítás elé elhelyez egy implicit (a felhasználó számára nem elérhető) mentési pontot. Ha az adott utasítás sikertelen, akkor azt az Oracle automatikusan visszagörgeti. Ha azonban az utasítás egy nem kezelt kivételt vált ki, akkor a gazdakörnyezet dönt a visszagörgetésről.

Azt javasoljuk, hogy minden PL/SQL programban explicit módon véglegesítsük vagy görgessük vissza a tranzakciókat. Ha ezt nem tesszük meg, a program lefutása után a tranzakció befejeződése attól függ, hogy milyen tevékenység következik. Egy DDL, DCL vagy COMMIT utasítás kiadása, vagy az EXIT, DISCONNECT, QUIT parancs végrehajtása véglegesíti a tranzakciót, a ROLLBACK utasítás vagy az SQL*Plus munkamenet abortálása pedig visszagörgeti azt.

Nagyon fontos megjegyezni, hogy egy PL/SQL blokk és a tranzakció nem azonos fogalmak. A blokk kezdete nem indít tranzakciót, mint ahogy a záró END sem jelenti a tranzakció befejeződését. Egy blokkban akárhány tranzakció elhelyezhető és egy tranzakció akárhány blokkon átívelhet.

A tranzakció tulajdonságai

Egy tranzakció tulajdonságait a SET TRANSACTION utasítással állíthatjuk be. Ennek az utasításnak

mindig a tranzakció első utasításának kell lenni. Alakja:

```
SET TRANSACTION{READ ONLY|

READ WRITE|

ISOLATION LEVEL {SERIALIZABLE|

READ COMMITTED}|

USE ROLLBACK SEGMENT visszagörgető_szegmens}

[NAME sztring];
```

A READ ONLY egy csak olvasható tranzakciót indít el. A csak olvasható tranzakció lekérdezései az adatbázisnak azt a pillanatfelvételét látják, amely a tranzakció előtt keletkezik. Más felhasználók és tranzakciók által okozott változtatásokat a tranzakció nem látja. Egy csak olvasható tranzakcióban SELECT INTO (FOR UPDATE nélkül), OPEN, FETCH, CLOSE, LOCK TABLE, COMMIT és ROLLBACK utasítások helyezhetők el. A csak olvasható tranzakció nem generál visszagörgetési információt.

A READ WRITE az aktuális tranzakciót olvasható-írható tranzakcióvá nyilvánítja. A tranzakcióban minden DML-utasítás használható, ezek ugyanazon visszagörgető szegmensen dolgoznak.

Az ISOLATION LEVEL az adatbázist módosító tranzakciók kezelését adja meg. A READ COMMITTED az alapértelmezett izolációs szint, ennél szigorúbb izolációs szintet a SERIALIZABLE explicit megadásával írhatunk elő. Az izolációs szintekről bővebben *lásd* [15].

A USE ROLLBACK SEGMENT az aktuális tranzakcióhoz a megadott visszagörgető szegmenst rendeli és azt olvasható-írható tranzakcióvá teszi.

A NAME által a tranzakcióhoz hozzárendelt *sztring* hozzáférhető a tranzakció futása alatt, lehetővé téve hosszú tranzakciók monitorozását vagy elosztott tranzakcióknál a vitás tranzakciók kezelését.

Explicit zárolás

Az Oracle automatikusan zárolja az adatokat a feldolgozásnál. A felhasználó azonban explicit módon is zárolhat egy teljes táblát vagy egy tábla bizonyos sorait.

A SELECT FOR UPDATE utasítással a kérdés által leválogatott sorok egy későbbi UPDATE vagy DELETE utasítás számára a lekérdezés végrehajtása után lefoglalásra kerülnek. Ezt akkor tegyük meg, ha biztosak akarunk lenni abban, hogy az adott sorokat más felhasználó nem módosítja addig, míg mi a módosítást végre nem hajtjuk.

Ha egy olyan UPDATE vagy DELETE utasítást használunk, amelyikben van CURRENT OF utasításrész, akkor a kurzor lekérdezésében (*lásd* 8. fejezet) kötelező a FOR UPDATE utasításrész szerepeltetése. Az utasításrész alakja:

```
FOR UPDATE [NOWAIT]
```

A NOWAIT hiánya esetén az Oracle vár addig, amíg a sorok elérhetők lesznek, ha viszont megadjuk, akkor a sorok más felhasználó által történt zárolása esetén egy kivétel kiváltása mellett a vezérlés azonnal visszatér a programhoz, hogy az más tevékenységet hajthasson végre, mielőtt újra próbálkozna a zárolással.

A LOCK TABLE utasítás segítségével egy vagy több teljes táblát tudunk zárolni a megadott módon. Alakja:

```
LOCK TABLE tábla[,tábla]… IN mód MODE [NOWAIT];
```

A *mód* a következők valamelyike lehet: ROW SHARE, ROW EXCLUSIVE, SHARE, SHARE UPDATE, SHARE ROW EXCLUSIVE, EXCLUSIVE.

A NOWAIT jelentése az előbbi.

Az Oracle zárolási módjairól bővebben *lásd* [15] és [21].

Autonóm tranzakciók

Egy tranzakció működése közben elindíthat egy másik tranzakciót is. Gyakran előfordul,

hogy a másodikként indult tranzakció az indító tranzakció hatáskörén kívül működik. Az ilyen szituációk kezelésére vezeti be az Oracle az autonóm tranzakció fogalmát. Az *autonóm tranzakció* egy másik tranzakció (a *fő tranzakció*) által elindított, de tőle független tranzakció. Az autonóm tranzakció teljesen független, nincs megosztott zár, erőforrás vagy egyéb függőség a fő tranzakcióval. Segítségével moduláris, az újrafelhasználást jobban segítő szoftverkomponensek állíthatók elő.

Az autonóm tranzakció létrehozására egy pragma szolgál, melynek alakja:

```
PRAGMA AUTONOMOUS_TRANSACTION;
```

Ez a pragma egy *rutin* deklarációs részében helyezhető el és a rutint autonóm tranzakciónak deklarálja. A *rutin* itt a következők valamelyikét jelenti:

- külső szinten álló név nélküli blokk (*lásd* 6.1. alfejezet);
- lokális, tárolt vagy csomagbeli alprogram (*lásd* 6.2. alfejezet és 9., 10. fejezet);
- objektumtípus metódusa (*lásd* 14. fejezet).

Javasoljuk, hogy az olvashatóság kedvéért a pragma a deklarációs rész elején legyen elhelyezve.

Egy csomag összes alprogramját nem tudjuk egyszerre autonómmá tenni (a pragma nem állhat csomagszinten).

Beágyazott blokk nem lehet autonóm. Trigger törzse viszont deklarálható autonómnak, illetve lehet autonómnak deklarált eljárás hívása.

Az Oracle nem támogatja a beágyazott tranzakciókat. Az autonóm tranzakció fogalma nem ekvivalens a beágyazott tranzakció fogalmával, ugyanis a beágyazott tranzakciók teljes mértékben a fő tranzakció részei, tőle függnek minden értelemben.

Az autonóm és beágyazott tranzakciók közötti különbségek a következők:

- Az autonóm tranzakció és a fő tranzakció erőforrásai különböznek, a beágyazott és fő tranzakcióé közösek.
- Az autonóm tranzakció visszagörgetése független a fő tranzakció visszagörgetésétől. A beágyazott tranzakció viszont visszagörgetésre kerül, ha a fő tranzakció visszagörgetésre került.
- Az autonóm tranzakció véglegesítése után az általa okozott változások azonnal láthatók minden más tranzakció számára. A beágyazott tranzakció véglegesítése utáni változások csak a fő tranzakcióban láthatók, más tranzakciók azokat csak a fő tranzakció véglegesítése után látják.

Egy autonómnak deklarált rutin egymás után több autonóm tranzakciót is végrehajthat, azaz a kiadott COMMIT vagy ROLLBACK nem jelenti a rutin befejezését.

Példa

```
CREATE TABLE a_tabla (

oszlop NUMBER

DECLARE

PROCEDURE autonom(p NUMBER) IS

PRAGMA AUTONOMOUS_TRANSACTION;

BEGIN

/* Első autonóm tranzakció kezdete - A1 */

INSERT INTO a_tabla VALUES(p);

/* Második autonóm tranzakció kezdete - A2*/
```

```
INSERT INTO a_tabla VALUES(p+1);

COMMIT;

END;

BEGIN

/* Itt még a fő tranzakció fut - F */

SAVEPOINT kezdet;

/* Az eljáráshívás autonóm tranzakciót indít */

autonom(10);

/* A fő tranzakcó visszagörgetése */

ROLLBACK TO kezdet;

END;

/

SELECT * FROM a_tabla;

/*

OSZLOP

-----------

10

11

*/
```

Meg kell jegyezni, hogy mind az A1, mind az A2 tranzakció számára F a fő tranzakció.

Mielőtt a vezérlés kilép az autonóm rutinból, az utolsó tranzakciót is be kell fejezni, annak változtatásait vagy véglegesíteni kell, vagy vissza kell görgetni. Ellenkező esetben a be nem fejezett autonóm tranzakciót az Oracle implicit módon visszagörgeti és a fő tranzakcióban a futását indító helyen (ez lehet egy alprogramhívás, lehet a futtató környezet névtelen blokk esetén, de lehet egy egyszerű DML utasítás hívása is, amelynek hatására egy trigger fut le, elindítva egy autonóm tranzakciót) kivételt vált ki.

Egy autonóm tranzakcióból új, tőle független autonóm tranzakció indítható, amely az indító tranzakciótól és az azt indító fő tranzakciótól is független. Vagyis ha FT fő tranzakcióból indul egy AT1 autonóm tranzakció, akkor AT1-ből is indulhat egy AT2 autonóm tranzakció, amely számára AT1 lesz a fő tranzakció. Ekkor mindhárom tranzakció egymástól független lesz. Az adatbázisban az összes munkamenetben együttvéve egy időben megengedett befejezetlen tranzakciók maximális számát az Oracle TRANSACTIONS inicializációs paramétere határozza meg.

Felhasználó által definiált zárak

A DBMS_LOCK csomag lehetőséget ad felhasználói zárak létrehozására. Lefoglalhatunk egy új zárat megadott zárolási móddal, elnevezhetjük azt, hogy más munkamenetekben és más adatbázis-példányokban is látható legyen, megváltoztathatjuk a zárolás módját, és végül feloldhatjuk. A DBMS_LOCK az Oracle Lock Managementet használja, ezért egy felhasználói zár teljesen megegyezik egy Oracle-zárral, így képes például holtpont detektálására.

A felhasználói zárak segítségével

- kizárólagos hozzáférést biztosíthatunk egy adott eszközhöz, például a terminálhoz;
- alkalmazás szintű olvasási zárakat hozhatunk létre;

- érzékelhetjük egy zár feloldását, így lehetőségünk nyílik egy alkalmazás végrehajtása utáni speciális tevékenységek végrehajtására;
- szinkronizációs eszközként kikényszeríthetjük az alkalmazások szekvenciális végrehajtását.

6. fejezet - Programegységek

A PL/SQL nyelv a *blokkszerkezetű* nyelvek közé tartozik. A procedurális nyelveknél szokásos programegységek közül a PL/SQL a következő hármat ismeri:

- blokk,
- alprogram,
- csomag.

A blokkot és alprogramot ebben a fejezetben, a csomagot a 10. fejezetben tárgyaljuk.

1. A blokk

A blokk a PL/SQL alapvető programegysége. Kezelhető önállóan és beágyazható más programegységbe, ugyanis a blokk megjelenhet bárhol a programban, ahol végrehajtható utasítás állhat. A blokk felépítése a következő:

```
[címke] [DECLARE deklarációk]

BEGIN utasítás [utasítás]…

[EXCEPTION kivételkezelő]

END [címke];
```

Egy blokk lehet címkézett vagy címkézetlen (anonim). Egy címkézetlen blokk akkor kezd el működni, ha szekvenciálisan rákerül a vezérlés. A címkézett blokkra viszont ezen túlmenően GOTO utasítással is át lehet adni a vezérlést.

A blokknak három alapvető része van: *deklarációs, végrehajtható* és *kivételkezelő* rész. Ezek közül az első és utolsó opcionális.

A DECLARE alapszóval kezdődő deklarációs részben a blokk lokális eszközeit deklaráljuk, ezek az alábbiak lehetnek:

- típus,
- nevesített konstans,
- változó,
- kivétel,
- kurzor,
- alprogram.

Az alprogramok deklarációja csak az összes többi eszköz deklarációja után helyezhető el.

A BEGIN alapszó után tetszőleges végrehajtható utasítássorozat áll. Ezek egy adott algoritmus leírását teszik lehetővé.

Az EXCEPTION alapszó után kivételkezelő áll, ennek részleteit a 7. fejezet tárgyalja.

A blokkot záró END nem jelenti egy tranzakció végét. Egy blokk működése szétoszthatő több tranzakcióba és egy tranzakció akárhány blokkot tartalmazhat.

A blokk működése befejeződik, ha

- elfogynak a végrehajtható utasításai, ekkor beágyazott blokknál a blokkot követő utasításra, egyébként a hívó környezetbe kerül a vezérlés;
- kivétel következik be (*lásd* 7. fejezet);
- ha GOTO-val kiugrunk belőle (ez csak beágyazott blokknál lehetséges);
- a RETURN utasítás hatására (ekkor az összes tartalmazó blokk is befejeződik).

Példa

```
/* Blokk */

<<kulso>>

DECLARE

s VARCHAR2(30) := 'hal';

BEGIN

/* beágyazott blokk */

DECLARE

s2 VARCHAR2(20) := 'El_lgató fej_lgató';

BEGIN

-- s2 is és s is látható

DBMS_OUTPUT.PUT_LINE(s); -- 'hal'

s := REPLACE(s2, '_', s);

-- blokk vége

END;

DBMS_OUTPUT.PUT_LINE(s); -- 'Elhallgató fejhallgató'

/* A következő sor fordítási hibát eredményezne, mert s2 már nem látható.

DBMS_OUTPUT.PUT_LINE(s2);

*/

<<belso>>

DECLARE

sNUMBER; -- elfedi a külső blokkbeli deklarációt

BEGIN

s := 5;

belso.s := 7;

kulso.s := 'Almafa';

GOTO ki;

-- Ide soha nem kerülhet a vezérlés.

DBMS_OUTPUT.PUT_LINE('Sikertelen GOTO ?!');

END;

DBMS_OUTPUT.PUT_LINE(s); -- 'Almafa'
```

```
<<ki>>
BEGIN
/* A következő érték nem fér bele a változóba,
VALUE_ERROR kivétel váltódik ki.*/
s := '1234567890ABCDEFGHIJ1234567890ABCDEFGHHIJ';
-- A kivétel miatt ide nem kerülhet a vezérlés.
DBMS_OUTPUT.PUT_LINE('Mégiscsak elfért az a sztring a változóban.');
DBMS_OUTPUT.PUT_LINE(s);
EXCEPTION
WHEN VALUE_ERROR THEN
DBMS_OUTPUT.PUT_LINE('VALUE_ERROR volt.');
DBMS_OUTPUT.PUT_LINE(s); -- 'Almafa'
END;
-- A RETURN utasítás hatására is befejeződik a blokk működése.
-- Ilyenkor az összes tartalmazó blokk működése is befejeződik!
BEGIN
RETURN; -- A kulso címkéjű blokk működése is befejeződik.
-- Ide soha nem kerülhet a vezérlés.
DBMS_OUTPUT.PUT_LINE('Sikertelen RETURN 1. ?!');
END;
-- És ide sem kerülhet a vezérlés.
DBMS_OUTPUT.PUT_LINE('Sikertelen RETURN 2. ?!');
END;
/
```

Az eredmény:

```
SQL> @blokk
hal
Elhallgató fejhallgató
Almafa
VALUE_ERROR volt.
Almafa
A PL/SQL eljárás sikeresen befejeződött.
```

2. Alprogramok

Az alprogramok (mint minden procedurális nyelvben) a PL/SQL nyelvben is a *procedurális absztrakció*, az *újrafelhasználhatóság*, a *modularitás* és a *karbantarthatóság* eszközei. Az alprogramok paraméterezhetők és az

alprogramok *hívhatók (aktivizálhatók)* a felhasználás helyén. A PL/SQL alprogramjai alapértelmezés szerint *rekurzívan* hívhatók.

Az alprogramoknak két fő része van: a *specifikáció* és a *törzs*. Az alprogram felépítése ennek megfelelően a következő:

```
specifikáció

{IS|AS}

[deklarációk]

BEGIN utasítás [utasítás]…

[EXCEPTION kivételkezelő]

END [név];
```

Láthatjuk, hogy az alprogram törzse a DECLARE alapszótól eltekintve megegyezik egy címke nélküli blokkal. Az ott elmondottak érvényesek itt is.

Kétféle alprogram van: *eljárás* és *függvény*. Az eljárás valamilyen tevékenységet hajt végre, a függvény pedig egy adott értéket határoz meg.

Eljárás

Az eljárás specifikációjának alakja az alábbi:

```
PROCEDURE név[(formális_paraméter[,formális_paraméter]…)]
```

A *név* az eljárás neve, amelyre a hívásnál hivatkozunk. A név után az opcionális *formálisparaméter-*

lista áll. Egy *formális_paraméter* formája:

```
név [{IN|{OUT|IN OUT} [NOCOPY]}] típus [{:=|DEFAULT}kifejezés]
```

A formális paraméter neve után a paraméterátadás *módját* lehet megadni: IN esetén érték szerinti, OUT esetén eredmény szerinti, IN OUT esetén érték-eredmény szerinti a paraméterátadás. Ha nem adjuk meg, akkor az IN alapértelmezett.

Az alprogram törzsében az IN módú paraméter nevesített konstansként, az OUT módú változóként, az IN OUT módú inicializált változóként kezelhető. Tehát az IN módú paraméternek nem adható érték.

Az OUT módú formális paraméter automatikus kezdőértéke NULL, tehát típusa nem lehet eleve NOT NULL megszorítással rendelkező altípus (például NATURALN).

IN mód esetén az aktuális paraméter kifejezés, OUT és IN OUT esetén változó lehet.

Az Oracle alaphelyzetben az OUT és IN OUT esetén a paraméterátadást értékmásolással oldja meg, azonban IN esetén nincs értékmásolás, itt a formális paraméter *referencia* típusú és csak az aktuális paraméter *értékének* a *címét* kapja meg. Ez azonban nem cím szerinti paraméterátadás, mert az aktuális paraméter értéke nem írható felül.

Ha azt szeretnénk, hogy az OUT és az IN OUT esetben se legyen értékmásolás, akkor adjuk meg a NOCOPY opciót. Ez egy fordítói ajánlás *(hint),* amelyet a fordító vagy figyelembe vesz, vagy nem. Bővebb információt ezzel kapcsolatban a [19] dokumentációban talál az Olvasó.

A *típus* a változó deklarációjánál megismert típusmegadás valamelyike lehet. A típus megadása nem tartalmazhat korlátozásokat, azaz hossz-, pontosság- és skálainformációkat, valamint nem szerepelhet a NOT NULL előírás sem. Korlátozott típus így csak programozói altípus vagy %TYPE segítségével adható meg.

1. példa

```
DECLARE
```

```
-- Hibás paramétertípus, nem lehet korlátozást megadni

PROCEDURE proc1(p_nev VARCHAR2(100) NOT NULL) …

-- Viszont mégis van rá mód programozói altípus segítségével

SUBTYPE t_nev IS VARCHAR2(100) NOT NULL;

PROCEDURE proc2(p_nev t_nev) …
```

A := vagy DEFAULT IN módú formális paramétereknek kezdőértéket ad a *kifejezés* segítségével.

Az eljárást *utasításszerűen* tudjuk meghívni. Tehát eljáráshívás a programban mindenütt szerepelhet, ahol végrehajtható utasítás állhat. A hívás formája: név és aktuális paraméterlista.

Az eljárás befejeződik, ha elfogynak a végrehajtható utasításai, vagy pedig a RETURN utasítás hajtódik végre, amelynek alakja:

```
RETURN;
```

Ekkor a vezérlés visszatér a hívást követő utasításra. A RETURN utasítás ezen alakja használható blokk befejeztetésére is.

Eljárás nem fejeztethető be GOTO utasítással.

2. példa (Különböző paraméterátadási módok)

```
DECLARE

v_Szam Number;

PROCEDURE inp(p_In IN NUMBER)

IS

BEGIN

DBMS_OUTPUT.PUT_LINE('in:' || p_In);

/* Az értékadás nem megengedett,

az IN módú paraméter nevesített konstansként viselkedik a törzsben,

ezért a következő utasítás fordítási hibát eredményezne: */

-- p_In := 0;

END inp;

PROCEDURE outp(p_Out OUT NUMBER)

IS

BEGIN

/* Az OUT módú paraméter értékére lehet hivatkozni.

Kezdeti értéke azonban NULL. */

DBMS_OUTPUT.PUT_LINE('out:' || NVL(p_Out, -1));

p_Out := 20;

PROCEDURE inoutp(p_Inout IN OUT NUMBER)

IS

BEGIN
```

```
/* Az IN OUT módú paraméter értékére lehet hivatkozni.
Kezdeti értéke az aktuális paraméter értéke lesz. */
DBMS_OUTPUT.PUT_LINE('inout:' || p_Inout);
p_Inout := 30;
END inoutp;
PROCEDURE outp_kivetel(p_Out IN OUT NUMBER)
IS
BEGIN
/* Az OUT és az IN OUT módú paraméter értéke csak az alprogram
sikeres lefutása esetén kerül vissza az aktuális paraméterbe.
Kezeletlen kivétel esetén nem.
*/
p_Out := 40;
DBMS_OUTPUT.PUT_LINE('kivétel előtt:' || p_Out);
RAISE VALUE_ERROR;
END outp_kivetel;
BEGIN
v_Szam := 10;
DBMS_OUTPUT.PUT_LINE('1:' || v_Szam);
inp(v_Szam);
inp(v_Szam + 1000); -- tetszőleges kifejezés lehet IN módú paraméter
DBMS_OUTPUT.PUT_LINE('2:' || v_Szam);
outp(v_Szam);
/* outp és inoutp paramétere csak változó lehet, ezért
a következő utasítás fordítási hibát eredményezne: */
-- outp(v_Szam + 1000);
DBMS_OUTPUT.PUT_LINE('3:' || v_Szam);
inoutp(v_Szam);
/* A következő utasítás is fordítási hibát eredményezne: */
-- inoutp(v_Szam + 1000);
DBMS_OUTPUT.PUT_LINE('4:' || v_Szam);
outp_kivetel(v_Szam);
EXCEPTION
WHEN OTHERS THEN
DBMS_OUTPUT.PUT_LINE('kivételkezelőben:' || v_Szam);
END;
```

```
/
/*
Eredmény:
1:10
in:10
in:1010
81
2:10
out:-1
3:20
inout:20
4:30
kivétel előtt:40
kivételkezelőben:30
A PL/SQL eljárás sikeresen befejeződött.
*/
```

3. példa (Eljárás használata)

```
/*
A következőkben megadunk egy eljárást, amely
adminisztrálja egy könyv kölcsönzését. A kölcsönzés sikeres,
ha van szabad példány a könyvből és a felhasználó még kölcsönözhet
könyvet. Egyébként a kölcsönzés nem sikeres és a hiba okát
kiírjuk.
Az eljárás használatát a tartalmazó blokk demonstrálja.
DECLARE
PROCEDURE kolcsonoz(
p_Ugyfel_id ugyfel.id%TYPE,
p_Konyv_id konyv.id%TYPE,
p_Megjegyzes kolcsonzes.megjegyzes%TYPE
) IS
-- A könyvből ennyi szabad példány van.
v_Szabad konyv.szabad%TYPE;
-- Az ügyfélnél levő könyvek száma
v_Kolcsonzott PLS_INTEGER;
-- Az ügyfél által maximum kölcsönözhető könyvek száma
```

```
v_Max_konyv ugyfel.max_konyv%TYPE;
BEGIN
DBMS_OUTPUT.NEW_LINE;
DBMS_OUTPUT.PUT_LINE('Kölcsönzés - ügyfél id: ' || p_Ugyfel_id
|| ', könyv id: ' || p_Konyv_id || '.');
-- Van-e szabad példány a könyvből?
SELECT szabad
INTO v_Szabad
FROM konyv
WHERE id = p_Konyv_id;
IF v_Szabad = 0 THEN
DBMS_OUTPUT.PUT_LINE('Hiba! A könyv minden példánya ki van kölcsönözve.');
RETURN;
END IF;
-- Kölcsönözhet még az ügyfél könyvet?
SELECT COUNT(1)
INTO v_Kolcsonzott
FROM TABLE(SELECT konyvek
FROM ugyfel
WHERE id = p_Ugyfel_id);
SELECT max_konyv
INTO v_Max_konyv
FROM ugyfel
WHERE id = p_Ugyfel_id;
IF v_Max_konyv <= v_Kolcsonzott THEN
DBMS_OUTPUT.PUT_LINE('Hiba! Az ügyfél nem kölcsönözhet több könyvet.');
RETURN;
END IF;
-- A kölcsönzésnek nincs akadálya
v_Datum := SYSDATE;
UPDATE konyv
SET szabad = szabad - 1
WHERE id = p_Konyv_id;
INSERT INTO kolcsonzes
VALUES (p_Ugyfel_id, p_Konyv_id, v_Datum, 0, p_Megjegyzes);
INSERT INTO TABLE(SELECT konyvek
```

```
FROM ugyfel
WHERE id = p_Ugyfel_id)
VALUES (p_Konyv_id, v_Datum);
DBMS_OUTPUT.PUT_LINE('Sikeres kölcsönzés.');
END kolcsonoz;
BEGIN
/*
Tóth László (20) kölcsönzi a 'Java - start!' című könyvet (25).
*/
kolcsonoz(20, 25, NULL);
/*
József István (15) kölcsönzi a 'A teljesség felé' című könyvet (10).
Nem sikerül, mert az ügyfél már a maximális számú könyvet kölcsönzi.
*/
kolcsonoz(15, 10, NULL);
/*
Komor Ágnes (30) kölcsönzi a 'A critical introduction...' című könyvet
(35).
Nem sikerül, mert a könyvből már nincs szabad példány.
*/
kolcsonoz(30, 35, NULL);
END;
/
/*
Eredmény:
Kölcsönzés - ügyfél id: 20, könyv id: 25.
Sikeres kölcsönzés.
Kölcsönzés - ügyfél id: 15, könyv id: 10.
Hiba! Az ügyfél nem kölcsönözhet több könyvet.
Kölcsönzés - ügyfél id: 30, könyv id: 35.
Hiba! A könyv minden példánya ki van kölcsönözve.
A PL/SQL eljárás sikeresen befejeződött.
*/
```

4. példa (Példa a NOCOPY használatára)

```
DECLARE
```

```
v_lista T_Konyvek;
t0 TIMESTAMP;
t1 TIMESTAMP;
t2 TIMESTAMP;
t3 TIMESTAMP;
t4 TIMESTAMP;
PROCEDURE ido(t TIMESTAMP)
IS
BEGIN
DBMS_OUTPUT.PUT_LINE(SUBSTR(TO_CHAR(t, 'SS.FF'), 1, 6));
/* Az alapértelmezett átadási mód IN esetén */
PROCEDURE inp(p T_Konyvek) IS
BEGIN
NULL;
END;
/* IN OUT NOCOPY nélkül */
PROCEDURE inoutp(p IN OUT T_Konyvek) IS
BEGIN
NULL;
END;
/* IN OUT NOCOPY mellett */
PROCEDURE inoutp_nocopy(p IN OUT NOCOPY T_Konyvek) IS
BEGIN
NULL;
END;
BEGIN
/* Feltöltjük a nagyméretű változót adatokkal. */
t0 := SYSTIMESTAMP;
v_lista := T_Konyvek();
FOR i IN 1..10000
LOOP
v_lista.EXTEND;
v_lista(i) := T_Tetel(1, '00-JAN. -01');
END LOOP;
/* Rendre átadjuk a nagyméretű változót. */
t1 := SYSTIMESTAMP;
```

```
inp(v_lista);
t2 := SYSTIMESTAMP;
inoutp(v_lista);
t3 := SYSTIMESTAMP;
inoutp_nocopy(v_lista);
t4 := SYSTIMESTAMP;
ido(t0);
DBMS_OUTPUT.PUT_LINE('inicializálás');
ido(t1);
DBMS_OUTPUT.PUT_LINE('inp');
ido(t2);
DBMS_OUTPUT.PUT_LINE('inoutp');
ido(t3);
DBMS_OUTPUT.PUT_LINE('inoutp_nocopy');
ido(t4);
END;
/
/*
Egy eredmény: (feltéve, hogy a NOCOPY érvényben van)
16.422
inicializálás
16.458
inp
16.458
inoutp
16.482
inoutp_nocopy
16.482
A PL/SQL eljárás sikeresen befejeződött.
*/
```

5. példa (Példa a NOCOPY használatára)

```
DECLARE
v_Szam NUMBER;
/* OUT NOCOPY nélkül */
PROCEDURE outp(p OUT NUMBER) IS
```

```
BEGIN
p := 10;
RAISE VALUE_ERROR; -- kivétel
END outp;
/* OUT NOCOPY mellett */
PROCEDURE outp_nocopy(p OUT NOCOPY NUMBER) IS
BEGIN
p := 10;
RAISE VALUE_ERROR; -- kivétel
END outp_nocopy;
BEGIN
v_Szam := 0;
BEGIN
outp(v_Szam);
EXCEPTION
WHEN OTHERS THEN
DBMS_OUTPUT.PUT_LINE('1: ' || v_Szam);
END;
v_Szam := 0;
BEGIN
outp_nocopy(v_Szam);
EXCEPTION
WHEN OTHERS THEN
DBMS_OUTPUT.PUT_LINE('2: ' || v_Szam);
END;
END;
/
/*
Eredmény: (feltéve, hogy a NOCOPY érvényben van)
1: 0
2: 10
A PL/SQL eljárás sikeresen befejeződött.
Magyarázat:
Az eljáráshívásokban kivétel következik be. Így egyik eljárás sem sikeres.
outp hívása esetén így elmarad a formális paraméternek az aktuálisba
történő másolása.
```

```
outp_nocopy hívásakor azonban az aktuális paraméter megegyezik a formálissal,
így az értékadás eredményes lesz az aktuális paraméterre nézve.
*/
```

6. példa Példa a NOCOPY használatára)

```
DECLARE
v_Szam NUMBER;
/*
Eredmény szerinti paraméterátadás.
Az aktuális paraméter az alprogram befejeztével
értékül kapja a formális paraméter értékét.
*/
PROCEDURE valtoztat1(p OUT NUMBER)
IS
BEGIN
p := 10;
v_Szam := 5;
END valtoztat1;
/*
Referencia szerinti paraméterátadás (feltéve, hogy a NOCOPY érvényben van).
Az aktuális paraméter megegyezik a formálissal.
Nincs értékadás az alprogram befejezésekor.
*/
PROCEDURE valtoztat2(p OUT NOCOPY NUMBER)
IS
BEGIN
p := 10;
v_Szam := 5;
BEGIN
v_Szam := NULL;
valtoztat1(v_Szam); -- Az alprogram végén v_Szam p értékét kapja meg
DBMS_OUTPUT.PUT_LINE('eredmény szerint: ' || v_Szam);
v_Szam := NULL;
valtoztat2(v_Szam); -- Az alprogramban v_Szam és p megegyezik
DBMS_OUTPUT.PUT_LINE('referencia szerint: ' || v_Szam);
END;
```

```
/
/*
Eredmény:
eredmény szerint: 10
referencia szerint: 5
*/
```

7. példa (Példa a NOCOPY használatára)

```
DECLARE
v_Szam NUMBER;
/*
A három paraméter átadásának módja (feltéve, hogy a NOCOPY érvényben van):
p1 - referencia szerinti
p2 – érték-eredmény szerinti
p3 - referencia szerinti
*/
PROCEDURE elj(
p1 IN NUMBER,
p2 IN OUT NUMBER,
p3 IN OUT NOCOPY NUMBER
) IS
BEGIN
DBMS_OUTPUT.PUT_LINE('1: '||v_Szam||' '||p1||' '||p2||' '||p3);
p2 := 20;
DBMS_OUTPUT.PUT_LINE('2: '||v_Szam||' '||p1||' '||p2||' '||p3);
p3 := 30;
DBMS_OUTPUT.PUT_LINE('3: '||v_Szam||' '||p1||' '||p2||' '||p3);
END elj;
BEGIN
DBMS_OUTPUT.PUT_LINE(' v_Szam p1 p2 p3');
DBMS_OUTPUT.PUT_LINE('-- ------ -- -- --');
v_Szam := 10;
elj(v_Szam, v_Szam, v_Szam);
DBMS_OUTPUT.PUT_LINE('4: ' || v_Szam);
END;
/
```

```
/*
Eredmény:
v_Szam p1 p2 p3
-- ------ -- -- --
1: 10 10 10 10
2: 10 10 20 10
3: 30 30 20 30
4: 20
Magyarázat:
Az eljárás hívásakor p1, p3 és v_Szam ugyanazt a szám objektumot jelölik,
p2 azonban nem. Ezért p2 := 20; értékadás nem változtatja meg p1-et, így
p3-at és v_Szam-ot sem közvetlenül. A p3 := 30; értékadás viszont
megváltoztatja p1-et, azaz p3-at, azaz v_Szam-ot is 30-ra.
Az eljárás befejeztével v_Szam aktuális paraméter megkapja p2 formális
paraméter értékét, az értékeredmény átadási mód miatt. Így v_Szam értéke
20 lesz.
A példa azt is igazolja, hogy p1 is referencia szerint kerül átadásra.
*/
```

Függvény

A függvény specifikációjának alakja:

```
FUNCTION név [(formális_paraméter[,formális_paraméter]…)]
RETURN típus
```

A formálisparaméter-lista szintaktikailag és szemantikailag megegyezik az eljárás formálisparaméter-listájával.

A RETURN alapszó után a *típus* a függvény visszatérési értékének a típusát határozza meg.

Egy függvényt meghívni kifejezésben lehet. A hívás formája: név és aktuálisparaméterlista.

A függvény törzsében legalább egy RETURN utasításnak szerepelnie kell, különben a PROGRAM_ERROR kivétel váltódik ki működése közben. A függvényben használt RETURN utasítás a visszatérési értéket is meghatározza. Alakja:

```
RETURN [()kifejezés[)];
```

A visszatérési érték a *kifejezés* értéke lesz, az érték a függvény nevéhez rendelődik, az közvetíti vissza a hívás helyére.

A függvény *mellékhatásának* hívjuk azt a jelenséget, amikor a függvény megváltoztatja a paramétereit vagy a környezetét (a globális változóit). A függvény feladata a visszatérési érték meghatározása, ezért a mellékhatás akár káros is lehet. Éppen ezért kerüljük az OUT és IN OUT módú paraméterek és a globális változók használatát.

8. példa

```
DECLARE
```

```
v_Datum DATE;
/*
Megnöveli p_Datum-ot a második p_Ido-vel, aminek
a mértékegységét p_Egyseg tartalmazza. Ezek értéke
'perc', 'óra', 'nap', 'hónap' egyike lehet.
Ha hibás a mértékegység, akkor az eredeti dátumot kapjuk vissza.
*/
FUNCTION hozzaad(
p_Datum DATE,
p_Ido NUMBER,
p_Egyseg VARCHAR2
) RETURN DATE IS
rv DATE;
BEGIN
CASE p_Egyseg
WHEN 'perc' THEN
rv := p_Datum + p_Ido/(24*60);
WHEN 'óra' THEN
rv := p_Datum + p_Ido/24;
WHEN 'nap' THEN
rv := p_Datum + p_Ido;
WHEN 'hónap' THEN
rv := ADD_MONTHS(p_Datum, p_Ido);
ELSE
rv := p_Datum;
END CASE;
RETURN rv;
END hozzaad;
PROCEDURE kiir(p_Datum DATE) IS
BEGIN
DBMS_OUTPUT.PUT_LINE(i || ': ' || TO_CHAR(p_Datum, 'YYYY-MON-DD HH24:MI:SS'));
i := i+1;
END kiir;
BEGIN
v_Datum := TO_DATE('2006-MÁJ. -01 20:00:00', 'YYYY-MON-DD HH24:MI:SS');
kiir(v_Datum); -- 1
```

```
DBMS_OUTPUT.NEW_LINE;
kiir(hozzaad(v_Datum, 5, 'perc')); -- 2
kiir(hozzaad(v_Datum, 1.25, 'óra')); -- 3
kiir(hozzaad(v_Datum, -7, 'nap')); -- 4
kiir(hozzaad(v_Datum, -8, 'hónap')); -- 5
END;
/
/*
Eredmény:
1: 2006-MÁJ. -01 20:00:00
2: 2006-MÁJ. -01 20:05:00
3: 2006-MÁJ. -01 21:15:00
4: 2006-ÁPR. -24 20:00:00
5: 2005-SZEPT.-01 20:00:00
A PL/SQL eljárás sikeresen befejeződött.
*/
```

9. példa

```
DECLARE
i NUMBER;
j NUMBER;
k NUMBER;
/*
Ennek a függvénynek van pár mellékhatása.
*/
FUNCTION mellekhatas(
p1 NUMBER,
/* OUT és IN OUT módú paraméterek nem lehetnek
egy tiszta függvényben.
Minden értéket a visszatérési értékben kellene
visszaadni. Ez rekorddal megtehető, amennyiben
tényleg szükséges. */
p2 OUT NUMBER,
p3 IN OUT NUMBER,
p4 IN OUT NUMBER
) RETURN NUMBER IS
```

```
BEGIN

i := 10; -- globális változó megváltoztatása

DBMS_OUTPUT.PUT_LINE('Egy tiszta függvény nem ír a kimenetére sem.');

p2 := 20;

p3 := 30;

p4 := p3;

RETURN 50;

BEGIN

i := 0;

j := mellekhatas(j, j, j, k);

/* Mennyi most i, j és k értéke? */

DBMS_OUTPUT.PUT_LINE('i= ' || i);

DBMS_OUTPUT.PUT_LINE('j= ' || j);

DBMS_OUTPUT.PUT_LINE('k= ' || k);

END;

/

/*

Eredmény:

Egy tiszta függvény nem ír a kimenetére sem.

i= 10

j= 50

k= 30

A PL/SQL eljárás sikeresen befejeződött.

*/
```

Formális és aktuális paraméterek

Az alprogramok közötti kommunikáció, információcsere *paraméterek* segítségével történik. Egy adott formálisparaméter-listával rendelkező alprogram akárhány aktuális paraméterlistával meghívható.

Az aktuális paraméter típusának a formális paraméter típusával kompatibilisnek kell lennie.

A formális paraméterek és az aktuális paraméterek egymáshoz rendelése történhet *pozíció* vagy *név* szerint. A pozíció szerinti egymáshoz rendelés azt jelenti, hogy a formális és aktuális paraméterek sorrendje a döntő *(sorrendi kötés)*. Az első formális paraméterhez az első aktuális paraméter, a másodikhoz a második stb. rendelődik hozzá.

A név szerinti egymáshoz rendelésnél *(név szerinti kötés)* az aktuálisparaméter-listán a formális paraméterek sorrendjétől függetlenül, tetszőleges sorrendben felsoroljuk a formális paraméterek *nevét*, majd a => jelkombináció után megadjuk a megfelelő aktuális paramétert.

A kétfajta megfeleltetés keverhető is, de lényeges, hogy elöl mindig a pozicionális paraméterek állnak, és őket követik a név szerinti kötéssel rendelkezők.

A PL/SQL-ben a programozó csak *fix* paraméterű alprogramokat tud írni (szemben például a C nyelvvel, ahol *változó* paraméterszámú függvények is létrehozhatók). Az alprogramok hívásánál az aktuális paraméterek száma kisebb vagy egyenlő lehet, mint a formális paraméterek száma. Ez az IN módú paramétereknek adott kezdőértékek kezelésétől függ. Egy olyan formális paraméterhez, amelynek van kezdőértéke, nem szükséges megadnunk aktuális paramétert. Ha megadunk, akkor az inicializálás az aktuális paraméter értékével, ha nem adunk meg, akkor a formálisparaméter-listán szereplő kezdőértékkel történik.

A PL/SQL megengedi lokális és csomagbeli alprogramnevek *túlterhelését*. Az ilyen alprogramoknál a név azonos, de a formális paraméterek száma, típusa vagy sorrendje eltérő kell legyen. Ekkor a fordító az aktuálisparaméter-lista alapján választja ki a hívásnál a megfelelő törzset.

10. példa

```
DECLARE

v_Datum DATE;

/* Visszaadja p_Datumot a p_Format formátumban,

ha hiányzik valamelyik, akkor az alapértelmezett

kezdőértékkel dolgozik a függvény */

FUNCTION to_char2(

p_Datum DATE DEFAULT SYSDATE,

p_Format VARCHAR2 DEFAULT 'YYYY-MON-DD HH24:MI:SS'

) RETURN VARCHAR2 IS

BEGIN

return TO_CHAR(p_Datum, p_Format);

END to_char2;

BEGIN

v_Datum := TO_DATE('2006-ÁPR. -10 20:00:00', 'YYYY-MON-DD HH24:MI:SS');

/* sorrendi kötés */

DBMS_OUTPUT.PUT_LINE('1: ' || to_char2(v_Datum, 'YYYY-MON-DD'));

/* név szerinti kötés */

DBMS_OUTPUT.PUT_LINE('2: ' || to_char2(p_Format => 'YYYY-MON-DD',

p_Datum => v_Datum));

/* név szerinti kötés és paraméter hiánya */

DBMS_OUTPUT.PUT_LINE('3: ' || to_char2(p_Format => 'YYYY-MON-DD'));

/* mindkét kötés keverve */

DBMS_OUTPUT.PUT_LINE('4: ' || to_char2(v_Datum,

p_Format => 'YYYY-MON-DD'));

/* mindkét paraméter hiánya */

DBMS_OUTPUT.PUT_LINE('5: ' || to_char2);

END;

/
```

```
/*
Eredmény (az aktuális dátum ma 2006-JÚN. -19):
1: 2006-ÁPR. -10
2: 2006-ÁPR. -10
3: 2006-JÚN. -19
4: 2006-ÁPR. -10
5: 2006-JÚN. -19 17:39:16
A PL/SQL eljárás sikeresen befejeződött.
*/
```

11. példa (Alprogram nevének túlterhelése)

```
DECLARE
v_Marad NUMBER;
/*
Megadja, hogy a p_Ugyfel_id azonosítójú ügyfél még
még hány könyvet kölcsönözhet.
*/
PROCEDURE marad(
p_Ugyfel_id ugyfel.id%TYPE,
p_Kolcsonozhet OUT NUMBER
) IS
BEGIN
SELECT max_konyv - db
INTO p_Kolcsonozhet
FROM
(SELECT max_konyv FROM ugyfel WHERE id = p_Ugyfel_id),
(SELECT COUNT(1) AS db FROM kolcsonzes WHERE kolcsonzo = p_Ugyfel_id);
END marad;
/*
Megadja, hogy a p_Ugyfel_nev nevű ügyfél
még hány könyvet kölcsönözhet.
*/
PROCEDURE marad(
p_Ugyfel_nev ugyfel.nev%TYPE,
p_Kolcsonozhet OUT NUMBER
) IS
```

```
v_Id ugyfel.id%TYPE;

BEGIN

SELECT id

INTO v_Id

FROM ugyfel

WHERE nev = p_Ugyfel_nev;

marad(v_Id, p_Kolcsonozhet);

END marad;

BEGIN

/* József István (15) */

marad(15, v_Marad);

DBMS_OUTPUT.PUT_LINE('1: ' || v_Marad);

/* Komor Ágnes (30) */

marad('Komor Ágnes', v_Marad);

DBMS_OUTPUT.PUT_LINE('2: ' || v_Marad);

END;

/

/*

Eredmény:

1: 0

2: 3

A PL/SQL eljárás sikeresen befejeződött.

*/
```

3. Beépített függvények

A PL/SQL a STANDARD csomagban deklarálja a beépített függvényeket, ezek minden

programban használhatók. Az alábbiakban felsoroljuk ezeket.

Kivételkezelő függvények: SQLCODE, SQLERRM

Numerikus függvények: ABS, ACOS, ASIN, ATAN, ATAN2, BITAND, CEIL, COS, COSH, EXP, FLOOR, LN, LOG, MOD, POWER, REMAIND, ER, ROUND, SIGN, SIN, SINH, SQRT, TAN, TANH, TRUNC

Karakteres függvények: ASCII, ASCIISTR, CHR, COMPOSE, CONCAT, DECOMPOSE, INITCAP, INSTR, INSTR2, INSTR4, INSTRB, INSTRC, LENGTH, LENGTH2, LENGTH4, LENGTHB, LENGTHC, LOWER, LPAD, LTRIM, NCHR, NLS_INITCAP, NLS_LOWER, NLSSORT, NLS_UPPER, REGEXP_INSTR, REGEXP_LIKE, REGEXP_REPLACE, REGEXP_SUBSTR, REPLACE, RPAD, RTRIM, SOUNDEX, SUBSTR, SUBSTR2, SUBSTR4, SUBSTRB, SUBSTRC, TRANSLATE, TRIM, UNISTR, UPPER

Konverziós függvények: CHARTOROWID, CONVERT, HEXTORAW, RAWTOHEX, RAWTONHEX, ROWIDTOCHAR, TO_BINARY_DOUBLE, TO_BLOB, TO_BINARY_FLOAT, TO_CHAR, TO_CLOB, TO_DATE, TO_MULTI_BYTE, TO_NCHAR, TO_NCLOB, TO_NUMBER, TO_SINGLE_BYTE

Dátum- és időfüggvények: ADD_MONTHS, CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP, DBTIMEZONE, EXTRACT, FROM_TZ, LAST_DAY, LOCALTIMESTAMP, MONTHS_BETWEEN, NEW_TIME, NEXT_DAY, NUMTODSINTERVAL, NUMTOYMINTERVAL, ROUND, SESSIONTIMEZONE, SYS_EXTRACT_UTC, SYSDATE, SYSTIMESTAMP, TO_DSINTERVAL, TO_TIME, TO_TIME_TZ, TO_TIMESTAMP, TO_TIMESTAMP_TZ, TO_YMINTERVAL, TRUNC, TZ_OFFSET

Objektumreferencia függvények: DEREF, REF, TREAT, VALUE

Egyéb függvények: BFILENAME, COALESCE, DECODE, DUMP, EMPTY_BLOB, EMPTY_CLOB, GREATEST, LEAST, NANVL, NLS_CHARSET_DECL_LEN, NLS_CHARSET_ID, NLS_CHARSET_NAME, NULLIF, NVL, SYS_CONTEXT, SYS_GUID, UID, USER, USERENV, VSIZE

A részletesebb ismeretekre vágyóknak a [8], [19] és [21] műveket ajánljuk.

4. Hatáskör és élettartam

A PL/SQL a statikus hatáskörkezelést alkalmazza, de egy lokális név csak a deklarációjától kezdve látszik. Egy adott programegységben deklarált név a tartalmazott programegységekben globális névként jelenik meg és újradeklarálás esetén *minősítéssel* hivatkozható. A minősítő az alprogram vagy csomag neve, illetve a blokk címkéje lehet. Ha tartalmazott programegységben nem deklarálunk újra egy globális nevet, akkor az minősítés nélkül hivatkozható.

A PL/SQL-ben tehát minden nevet az előtt kell deklarálni, mielőtt hivatkoznánk rá. Alprogramok esetén viszont előfordulhat a kölcsönös rekurzív hívás. Ennek kezelésére a PL/SQL az alprogramoknál ismeri az *előrehivatkozás* lehetőségét. Ahhoz, hogy alprogramot hívni tudjunk, elég a specifikációját ismernünk. Az előrehivatkozás úgy néz ki, hogy csak az alprogram specifikációját adjuk meg egy pontosvesszővel lezárva, az alprogram teljes deklarációja ugyanezen deklarációs részben csak később következik.

A PL/SQL a *dinamikus* élettartam-kezelést alkalmazza blokkokra és alprogramokra. Egy változó akkor kap címkomponenst, ha aktivizálódik az a programegység, amelynek ő lokális változója. Ekkor történik meg az automatikus vagy explicit kezdőértékadás. A címkomponens megszűnik, ha a programegység befejezi a működését.

Nevesített konstansok inicializálása is mindig megtörténik, ha aktivizálódik az a blokk vagy alprogram, amelyben deklaráltuk őket.

1. példa

```
<<blokk>>

DECLARE

i NUMBER := 1;

p VARCHAR2(10) := 'Hello';

/* Az alprogramban deklarált i elfedi a globális változót

p paraméter is elfedi a külső p változót */

PROCEDURE alprg1(p NUMBER DEFAULT 22) IS

i NUMBER := 2;

/* A lokális alprogram további névütközéseket tartalmaz. */

PROCEDURE alprg2(p NUMBER DEFAULT 555) IS

i NUMBER := 3;

BEGIN

/* A következő utasítások szemléltetik a minősített nevek
```

```
használatát.
A minősítések közül egyedül a 3. utasítás alprg2.p
minősítése felesleges (de megengedett). */
FOR i IN 4..4 LOOP
DBMS_OUTPUT.PUT_LINE(blokk.i || ', ' || blokk.p);
DBMS_OUTPUT.PUT_LINE(alprg1.i || ', ' || alprg1.p);
DBMS_OUTPUT.PUT_LINE(alprg2.i || ', ' || alprg2.p);
DBMS_OUTPUT.PUT_LINE(i || ', ' || p);
END LOOP;
END alprg2;
BEGIN
alprg2;
END alprg1;
BEGIN
alprg1;
END blokk;
/
/*
Eredmény:
1, Hello
2, 22
3, 555
4, 555
A PL/SQL eljárás sikeresen befejeződött.
*/
```

2. példa

```
/*
Kölcsönös hivatkozás
*/
DECLARE
PROCEDURE elj1(p NUMBER);
PROCEDURE elj2(p NUMBER)
IS
BEGIN
DBMS_OUTPUT.PUT_LINE('elj2: ' || p);
```

```
elj1(p+1);
END elj2;
PROCEDURE elj1(p NUMBER)
IS
BEGIN
DBMS_OUTPUT.PUT_LINE('elj1: ' || p);
IF p = 0 THEN
elj2(p+1);
END IF;
END elj1;
BEGIN
elj1(0);
END;
/
/*
Eredmény:
elj1: 0
elj2: 1
elj1: 2
A PL/SQL eljárás sikeresen befejeződött.
*/
```

3. példa

```
DECLARE
/* p és j inicializálása minden híváskor megtörténik. */
PROCEDURE alprg(p NUMBER DEFAULT 2*(-i)) IS
j NUMBER := i*3;
BEGIN
DBMS_OUTPUT.PUT_LINE(' '|| i || ' ' || p || ' ' || j);
END alprg;
BEGIN
DBMS_OUTPUT.PUT_LINE(' i p j');
DBMS_OUTPUT.PUT_LINE('-- -- --');
i := 1;
alprg;
i := 2;
```

```
alprg;

END;

/

/*

Eredmény:

i p j

-- -- --

1 -2 3

2 -4 6

A PL/SQL eljárás sikeresen befejeződött.

*/
```

7. fejezet - Kivételkezelés

Egy PL/SQL programban a kivételek *események*. A kivételek rendszere a PL/SQL-ben a futás közben bekövetkező események (kiemelt módon a *hibák*) kezelését teszi lehetővé. A kivételek lehetnek *beépített* kivételek (amelyeket általában a futtató rendszer vált ki), vagy *felhasználói* kivételek, amelyeket valamelyik programegység deklarációs részében határoztunk meg. A beépített kivételeknek *lehet* nevük (például ZERO_DIVIDE), a felhasználói kivételeknek *mindig van* nevük. A felhasználói kivételeket mindig explicit módon, külön utasítással kell kiváltani. Ezzel az utasítással beépített kivétel is kiváltható.

A kivételek kezelésére a programegységekbe *kivételkezelőt* építhetünk be.

Egy kivételhez a PL/SQL-ben egy *kód* és egy *üzenet* tartozik. A beépített üzenetek kódja (egyetlen esettől eltekintve) negatív, a felhasználói kivételeké pozitív.

A beépített kivételeket a STANDARD csomag definiálja. A megnevezett beépített kivételek és a hozzájuk tartozó, az Oracle-rendszer által adott hibaüzenetek (ezek nyelve az NLS beállításoktól függ) a 7.1. táblázatban láthatók. A 7.2. táblázat ismerteti a beépített kivételek tipikus kiváltó okait.

7.1. táblázat - Hibaüzenetek

Kivétel	Hibakód SQLCODE	Hibaüzenet SQLERRM
ACCESS_INTO_NULL	-6530	ORA-06530: Incializálatlan összetett objektumra való hivatkozás
CASE_NOT_FOUND	-6592	ORA-06592: CASE nem található a CASE állítás végrehajtásakor
COLLECTION_IS_NULL	-6531	ORA-06531: Inicializálatlan gyűjtőre való hivatkozás
CURSOR_ALREADY_OPEN	-6511	ORA-06511: PL/SQL: A kurzor már meg van nyitva
DUP_VAL_ON_INDEX	-1	ORA-00001: A(z) (.) egyediségre vonatkozó megszorítás nem teljesül
INVALID_CURSOR	-1001	ORA-01001: Nem megengedett kurzor
INVALID_NUMBER	-1722	ORA-01722: Nem megengedett szám
LOGIN_DENIED	-1017	ORA-01017: Nem megengedett felhasználónév/jelszó; a bejelentkezés visszautasítva
NO_DATA_FOUND	100	ORA-01403: Nem talált adatot
NOT_LOGGED_ON	-1012	ORA-01012: Nincs bejelentkezve
PROGRAM_ERROR	–6501	ORA-06501: PL/SQL: programhiba
ROWTYPE_MISMATC	-6504	ORA-06504: PL/SQL: Az Eredmény halmaz

H		változói vagy a kérdés visszaadott típusai nem illeszkednek
SELF_IS_NULL	-30625	ORA-30625: A metódus használata nem engedélyezett NULL SELF argumentummal
STORAGE_ERROR	-6500	ORA-06500: PL/SQL: tárolási hiba
SUBSCRIPT_BEYOND_COUNT	-6533	ORA-06533: Számlálón kívüli indexérték
SUBSCRIPT_OUTSIDE_LIMIT	-6532	ORA-06532: Határon kívüli index
SYS_INVALID_ROWID	-1410	ORA-01410: Nem megengedett ROWID
TIMEOUT_ON_RESOURCE	-51	ORA-00051: Időtúllépés történt erőforrásra várakozás közben
TOO_MANY_ROWS	-1422	ORA-01422: A pontos lehívás (FETCH) a kívántnál több sorral tér vissza
VALUE_ERROR	-6502	ORA-06502: PL/SQL: numerikus- vagy értékhiba
ZERO_DIVIDE	-1476	ORA-01476: Az osztó értéke nulla

7.2. táblázat - Beépített kivételek tipikus kiváltó okai

Kivétel	Kiváltódik, ha …
ACCESS_INTO_NULL	Megpróbál értéket adni egy inicializálatlan (automatikusan NULL) objektum attribútumának.
CASE_NOT_FOUND	A CASE utasítás egyetlen WHEN ága sem egyezik meg a feltétellel és nincs ELSE ág.
COLLECTION_IS_NULL	Megpróbál hivatkozni egy EXISTS-től különböző kollekciómetódusra egy inicializálatlan (automatikusan NULL) beágyazott tábla vagy dinamikus tömb esetén, vagy megpróbál értéket adni egy inicializálatlan beágyazott tábla vagy dinamikus tömb elemének.
CURSOR_ALREADY_OPEN	Megpróbál újra megnyitni egy már megnyitott kurzort. A kurzorokat le kell zárni, mielőtt újra megnyitjuk. A kurzor FOR ciklusa automatikusan megnyitja a hozzárendelt kurzort, így azt a ciklusban nem lehet újra megnyitni.
DUP_VAL_ON_INDEX	Már létező értéket próbál meg tárolni egy adatbázistábla olyan oszlopában, amelyen egyedi (UNIQUE) indexmegszorítás van.
INVALID_CURSOR	Megpróbál műveletet végezni egy meg nem nyitott kurzoron.

INVALID_NUMBER	SQL utasításban sikertelen egy karakterlánc konverziója, mert annak tartalma ténylegesen nem egy számot reprezentál. (A procedurális utasításokban ilyen konverziós hiba esetén VALUE_ERROR kivétel váltódik ki.) Ez a kivétel váltódik ki akkor is, ha a LIMIT előírás nem pozitív számot eredményez egy együttes hozzárendelést tartalmazó FETCH utasításban.
LOGIN_DENIED	Bejelentkezéskor hibás a felhasználónév vagy a jelszó.
NO_DATA_FOUND	Egy SELECT INTO utasítás nem ad vissza sorokat, vagy a programban hivatkozik egy beágyazott tábla törölt, vagy egy asszociatív tömb inicializálatlan elemére. Az SQL csoportfüggvényei, például AVG, SUM, mindig adnak vissza értéket vagy NULL-t, ezért csoportfüggvényt tartalmazó SELECT INTO utasítás sohasem váltja ki ezt a kivételt.
NOT_LOGGED_ON	Megpróbál adatbázis-műveletet végrehajtani úgy, hogy nem kapcsolódik Oracle-példányhoz.
PROGRAM_ERROR	PL/SQL belső hiba.
ROWTYPE_MISMATCH	Egy értékadásban a kurzor gazdaváltozó és az értékül adandó PL/SQL kurzor visszatérési típusa nem kompatibilis. Például abban az esetben, ha egy már megnyitott kurzorváltozót adunk át egy alprogramnak, akkor a formális és aktuális paraméterek visszatérési típusának kompatibilisnek kell lennie. Ugyanígy előfordulhat, hogy egy megnyitott gyenge kurzorváltozót adunk értékül egy erős kurzorváltozónak, és azok visszatérési típusa nem kompatibilis.
SELF_IS_NULL	Megpróbálja egy NULL referenciájú objektum metódusát meghívni, azaz a rögzített SELF paraméter (ami minden metódus első paramétere) NULL.
STORAGE_ERROR	Elfogyott a PL/SQL számára rendelkezésre álló memória, vagy a memóriaterület megsérült.
SUBSCRIPT_BEYOND_COUNT	Egy beágyazott tábla vagy dinamikus tömb méreténél nagyobb indexű elemére hivatkozik.
SUBSCRIPT_OUTSIDE_LIMIT	Egy beágyazott tábla vagy dinamikus tömb elemére hivatkozik egy olyan indexszel, ami nincs a megengedett tartományban (például –1).
SYS_INVALID_ROWID	Sikertelen egy karakterlánc konverziója ROWID típussá, mert a karakterlánc ténylegesen nem egy ROWID értéket reprezentál.
TIMEOUT_ON_RESOURCE	Időtúllépés történik egy erőforrásra várakozás közben. Az erőforrást valaki zárolta.
TOO_MANY_ROWS	Egy SELECT INTO utasítás egynél több sort ad vissza.

VALUE_ERROR	Aritmetikai, konverziós, csonkítási vagy hosszmegszorítási hiba történik. Például ha egy sztring változónak a deklarált maximális hosszánál hosszabb sztringet próbál meg értékül adni, akár egy SELECT INTO utasítással. Ilyenkor az értékadás érvénytelen, semmis lesz és VALUE_ERROR kivétel váltódik ki. Procedurális utasítások esetén akkor is ez a kivétel váltódik ki, ha egy sztring konverziója számmá sikertelen (SQL utasításokban ilyenkor INVALID_NUMBER kivétel váltódik ki).
ZERO_DIVIDE	Megpróbál nullával osztani.

Felhasználói kivételeket a EXCEPTION alapszóval deklarálhatunk:

```
DECLARE

sajat_kivetel EXCEPTION;
```

Olyan beépített kivételhez, amely eredetileg nincs nevesítve, egy pragma segítségével a programunkban nevet rendelhetünk egy programegység deklarációs részében. Ekkor deklarálnunk kell egy felhasználói kivételnevet, majd ugyanezen deklarációs részben később alkalmazni rá a pragmát. Ezután az adott beépített kivételt név szerint tudjuk kezelni. A pragma alakja:

```
PRAGMA EXCEPTION_INIT(kivételnév,kód);
```

1. példa

```
DECLARE

i PLS_INTEGER;

j NUMBER NOT NULL := 1;

/* Egy névtelen hiba nevesítése. */

numeric_overflow EXCEPTION;

PRAGMA EXCEPTION_INIT(numeric_overflow, -1426); -- numeric overflow

/* Egy már amúgy is nevesített kivételhez még egy név rendelése. */

VE_szinonima EXCEPTION;

PRAGMA EXCEPTION_INIT(VE_szinonima, -6502); -- VALUE_ERROR

BEGIN

/* Kezeljük a numeric overflow hibát, PL/SQL-ben ehhez

a hibához nincs előre definiálva kivételnév. */

<<blokk1>>

BEGIN

i := 2**32;

EXCEPTION

WHEN numeric_overflow THEN

DBMS_OUTPUT.PUT_LINE('Blokk1 - numeric_overflow!' || SQLERRM);

END blokk1;

/* A VE_szinonima használható VALUE_ERROR helyett. */
```

```
<<blokk2>>
BEGIN
i := NULL;
j := i; -- VALUE_ERROR-t vált ki, mert i NULL.
DBMS_OUTPUT.PUT_LINE(j);
EXCEPTION
WHEN VE_szinonima THEN
DBMS_OUTPUT.PUT_LINE('Blokk2 - VALUE_ERROR: ' || SQLERRM);
END blokk2;
/* A VALUE_ERROR is használható VE_szinonima helyett. A két kivétel
megegyezik. */
<<blokk2>>
BEGIN
RAISE VE_szinonima;
EXCEPTION
WHEN VALUE_ERROR THEN -- A saját kivételünk szinonima a VALUE_ERROR-ra
DBMS_OUTPUT.PUT_LINE('Blokk3 - VALUE_ERROR: ' || SQLERRM);
END blokk1;
END;
/
/*
Eredmény:
Blokk1 - numeric_overflow!ORA-01426: numerikus túlcsordulás
Blokk2 - VALUE_ERROR: ORA-06502: PL/SQL: numerikus- vagy értékhiba ()
Blokk3 - VALUE_ERROR: ORA-06502: PL/SQL: numerikus- vagy értékhiba ()
A PL/SQL eljárás sikeresen befejeződött.
*/
```

Bármely megnevezett kivétel kiváltható a következő utasítással:

```
RAISE kivételnév;
```

Az utasítás bárhol elhelyezhető, ahol végrehajtható utasítás szerepelhet.

Kivételkezelő bármely programegység végén az EXCEPTION alapszó után helyezhető el. Felépítése a következő:

```
WHEN kivételnév [OR kivételnév]…
THEN utasítás [utasítás]…
[WHEN kivételnév [OR kivételnév]…
THEN utasítás [utasítás]…]…
```

```
[WHEN OTHERS THEN utasítás [utasítás]…]
```

A kivételkezelő tehát olyan WHEN ágakból áll, amelyek név szerint kezelik a kivételeket és legutolsó ágként szerepelhet egy OTHERS ág, amely minden kivételt kezel.

Ha egy blokkban vagy alprogramban a végrehajtható részben bekövetkezik egy kivétel, akkor a végrehajtható rész futása félbeszakad és a futtató rendszer megvizsgálja, hogy a blokk vagy alprogram tartalmaz-e kivételkezelőt. Ha igen, megnézi, hogy valamelyik WHEN ágban nevesítve van-e a bekövetkezett kivétel. Ha igen, akkor végrehajtódnak a THEN után megadott utasítások. Ha köztük van GOTO, akkor a megadott címkén, ha nincs GOTO, blokk esetén a blokkot követő utasításon (ha van tartalmazó programegység) folytatódik a futás, egyébként pedig a vezérlés visszaadódik a hívási környezetbe. Az a programegység, amelyben egy kivétel bekövetkezik, inaktívvá válik, tehát futása nem folytatható. Így GOTO-val sem lehet visszatérni azon végrehajtható részre, ahol a kivétel bekövetkezett.

2. példa

```
DECLARE

a NUMBER NOT NULL := 1;

b NUMBER;

BEGIN

<<tartalmazo>>

BEGIN

<<vissza>>

a := b; -- VALUE_ERRORt vált ki, mert b NULL

EXCEPTION

WHEN VALUE_ERROR THEN

b := 1; -- Hogy legközelebb b ne legyen NULL;

-- GOTO vissza; -- fordítási hibát eredményezne

GOTO tartalmazo; -- külső blokk cimkéjére ugorhatunk

END;

END;

/
```

Ha egyik WHEN ágban sincs nevesítve a kivétel, de van WHEN OTHERS ág, akkor az abban megadott utasítások hajtódnak végre és a folytatás a fent elmondottaknak megfelelő.

Ha nincs OTHERS ág vagy nincs is kivételkezelő, akkor a kivétel *továbbadódik*. A kivétel továbbadása azt jelenti, hogy a programegység befejeződik, és a futtató rendszer blokk esetén a tartalmazó blokkban, alprogram esetén a hívó programegységben keres megfelelő kivételkezelőt.

Ha nincs több tartalmazó vagy hívó programegység, akkor a hívási környezetben egy UNHANDLED EXCEPTION kivétel váltódik ki. A kivétel távoli eljáráshívás (RPC) esetén nem adódik tovább.

Ha deklarációs részben vagy kivételkezelőben következik be egy kivétel, az azonnal továbbadódik.

3. példa

```
BEGIN

DECLARE

/* A következő változó inicializációja során
```

```
VALUE_ERROR kivétel váltódik ki. */

iNUMBER(5) := 123456;

BEGIN

NULL;

EXCEPTION

WHEN VALUE_ERROR THEN

/* Ez a kezelő nem tudja elkapni a deklarációs

részben bekövetkezett kivételt. */

DBMS_OUTPUT.PUT_LINE('belső');

END;

EXCEPTION

WHEN VALUE_ERROR THEN

/* Ez a kezelő kapja el a kivételt. */

DBMS_OUTPUT.PUT_LINE('külső');

END;

/

/*

Eredmény:

külső

A PL/SQL eljárás sikeresen befejeződött.

*/
```

Néha szükség lehet egy bekövetkezett kivétel újbóli kiváltására. Például ha egy kivételkezelőben az adott kivétel lokális kezelése után ugyanazt a kivételt tovább kell adni. Erre szolgál a RAISE utasítás kivételnév nélküli alakja, amely csak kivételkezelőben alkalmazható. Hatására újra kiváltódik az a kivétel, amely az adott kivételkezelőt aktiválta (és miután kivételkezelőben váltódott ki, azonnal tovább is adódik).

A kivételkezelőben használható két beépített függvény az SQLCODE és az SQLERRM. Paraméter nélkül az SQLCODE a bekövetkezett kivétel kódját, az SQLERRM a hozzárendelt üzenetet adja meg az NLS beállításoktól függő nyelven. Felhasználói kivételek esetén SQLCODE értéke +1, SQLERRM értéke: 'User-Defined Exception'.

Az SQLERRM meghívható paraméterrel is, a paraméternek egy kivételkódnak kell lennie. Ekkor az adott kódú kivétel üzenetét adja. Pozitív érték esetén (kivéve a 100-at) mindig a 'User-Defined Exception' az eredmény, 0 esetén 'ORA-0000: normal, successful completion', negatív kód esetén pedig a beépített kivétel üzenete.

Az SQLCODE és SQLERRM függvények természetesen nem csak kivételkezelőben használhatók, ilyenkor, ha nem volt kivétel, az SQLCODE 0 értékkel tér vissza, az SQLERRM pedig az ennek megfelelő üzenettel.

4. példa

```
DECLARE

hibas_argumentum EXCEPTION;

v_Datum DATE;

/*
```

```
Megnöveli p_Datum-ot a második p_Ido-vel, aminek
a mértékegységét p_Egyseg tartalmazza. Ezek értéke
'perc', 'óra', 'nap', 'hónap' egyike lehet.
Ha hibás a mértékegység, akkor hibas_argumentum kivétel
váltódik ki.
*/
FUNCTION hozzaad(
p_Datum DATE,
p_Ido NUMBER,
p_Egyseg VARCHAR2
) RETURN DATE IS
rv DATE;
BEGIN
CASE p_Egyseg
WHEN 'perc' THEN
rv := p_Datum + p_Ido/(24*60);
WHEN 'óra' THEN
rv := p_Datum + p_Ido/24;
WHEN 'nap' THEN
rv := p_Datum + p_Ido;
WHEN 'hónap' THEN
rv := ADD_MONTHS(p_Datum, p_Ido);
ELSE
RAISE hibas_argumentum;
END CASE;
RETURN rv;
END hozzaad;
/* Ez a függvény hibás mértékegység esetén nem vált ki
kivételt, hanem NULL-t ad vissza. */
FUNCTION hozzaad2(
p_Datum DATE,
p_Ido NUMBER,
p_Egyseg VARCHAR2
) RETURN DATE IS
rv DATE;
BEGIN
```

```
RETURN hozzaad(p_Datum, p_Ido, p_Egyseg);
EXCEPTION
WHEN hibas_argumentum THEN
RETURN NULL;
END hozzaad2;
BEGIN
<<blokk1>>
BEGIN
v_Datum := hozzaad(SYSDATE, 1, 'kiskutyafüle');
EXCEPTION
WHEN hibas_argumentum THEN
DBMS_OUTPUT.PUT_LINE('Blokk1 - hibás argumentum: '
|| SQLCODE || ', ' || SQLERRM);
END blokk1;
<<blokk2>>
BEGIN
v_Datum := hozzaad2(SYSDATE, 1, 'kiskutyafüle');
DBMS_OUTPUT.PUT_LINE('Blokk2 - nincs kivétel.');
EXCEPTION
WHEN OTHERS THEN
DBMS_OUTPUT.PUT_LINE('Blokk2 - hiba: '
|| SQLCODE || ', ' || SQLERRM);
RAISE; -- A hiba továbbadása külső irányba.
END blokk2;
END;
/
/*
Eredmény:
Blokk1 - hibás argumentum: 1, User-Defined Exception
Blokk2 - nincs kivétel.
A PL/SQL eljárás sikeresen befejeződött.
*/
```

A STANDARD csomag tartalmaz egy RAISE_APPLICATION_ERROR eljárást, amelynek három paramétere van:

- Az első paraméter egy kivételkód –20000 és –20999 között.
- A második paraméter egy maximum 2048 bájt hosszúságú sztring.

- A harmadik opcionális paraméter TRUE vagy FALSE (ez az alapértelmezés). TRUE esetén a hibaveremben az első paraméter által megadott kód felülírja a legutolsónak bekövetkezett kivétel kódját, FALSE esetén kiürül a verem és csak a most megadott kód kerül bele.

Az eljárás hívása esetén kiváltódik a megadott felhasználói kivétel a megadott üzenettel.

5. példa

```
DECLARE

SUBTYPE t_ugyfelrec IS ugyfel%ROWTYPE;

v_Ugyfel t_ugyfelrec;

v_Nev VARCHAR2(10);

/*

A függvény megadja az adott keresztnevű ügyfél adatait.

Ha nem egyértelmű a kérdés, akkor ezt kivétellel jelzi.

*/

FUNCTION ugyfel_nevhez(p_Keresztnev VARCHAR2)

RETURN t_ugyfelrec IS

v_Ugyfel t_ugyfelrec;

BEGIN

SELECT * INTO v_Ugyfel FROM ugyfel

WHERE UPPER(nev) LIKE '% %' || UPPER(p_Keresztnev) || '%';

RETURN v_Ugyfel;

EXCEPTION

/* Egy WHEN ág több nevesített kivételt is kezelhet. */

WHEN NO_DATA_FOUND OR TOO_MANY_ROWS THEN

RAISE_APPLICATION_ERROR(-20010,

'A keresett ügyfél nem vagy nem egyértelműen létezik');

END ugyfel_nevhez;

BEGIN

FOR i IN 1..3 LOOP

CASE i

WHEN 1 THEN v_Nev := 'Máté'; -- Egy Máté van a könyvtárban.

WHEN 2 THEN v_Nev := 'István'; -- Több István is van.

WHEN 3 THEN v_Nev := 'Gergő'; -- Nincs Gergő nálunk.

END CASE;

<<blokk1>>

BEGIN

DBMS_OUTPUT.PUT_LINE(i || '. Keresett név: "' || v_Nev || '".');
```

```
v_Ugyfel := ugyfel_nevhez(v_Nev);
DBMS_OUTPUT.PUT_LINE('Nincs hiba, ügyfél: '
|| v_Ugyfel.nev);
EXCEPTION
/* Csak a WHEN OTHERS ág tudja elkapni a névtelen kivételeket. */
WHEN OTHERS THEN
DBMS_OUTPUT.PUT_LINE('Hiba: ' || SQLCODE || ', ' || SQLERRM);
END blokk1;
END LOOP;
/* A felhasználói hiba számához is rendelhetünk kivételt. */
<<blokk2>>
DECLARE
hibas_ugyfelnev EXCEPTION;
PRAGMA EXCEPTION_INIT(hibas_ugyfelnev, -20010);
BEGIN
DBMS_OUTPUT.NEW_LINE;
DBMS_OUTPUT.PUT_LINE('Blokk2 - Szilveszter van-e?');
v_Ugyfel := ugyfel_nevhez('Szilveszter'); -- Persze nincs.
DBMS_OUTPUT.PUT_LINE('Igen van: ' || v_Ugyfel.nev);
EXCEPTION
WHEN hibas_ugyfelnev THEN
DBMS_OUTPUT.PUT_LINE('Hiba: '
|| SQLCODE || ', ' || SQLERRM);
RAISE; -- A hiba továbbadása
END blokk2;
/* Mivel itt nincs kivételkezelő, a blokk2 kivételkezelőjéből
továbbadott kivételt a futtató rendszer kezeli. */
END;
/
/*
Eredmény:
1. Keresett név: "Máté".
Nincs hiba, ügyfél: Szabó Máté István
2. Keresett név: "István".
Hiba: -20010, ORA-20010: A keresett ügyfél nem vagy nem egyértelműen létezik
3. Keresett név: "Gergő".
```

```
Hiba: -20010, ORA-20010: A keresett ügyfél nem vagy nem egyértelműen létezik
Blokk2 - Szilveszter van-e?
Hiba: -20010, ORA-20010: A keresett ügyfél nem vagy nem egyértelműen létezik
DECLARE
*
Hiba a(z) 1. sorban:
ORA-20010: A keresett ügyfél nem vagy nem egyértelműen létezik
ORA-06512: a(z) helyen a(z) 64. sornál
*/
```

8. fejezet - Kurzorok és kurzorváltozók

Egy SQL utasítás feldolgozásához az Oracle a memóriában egy speciális területet használ, melyet *környezeti területnek* hívunk. A környezeti terület információkat tartalmaz az utasítás által feldolgozott sorokról, lekérdezés esetén tartalmazza a visszaadott sorokat (amit *aktív halmaznak* nevezünk) és tartalmaz egy mutatót az utasítás belső reprezentációjára.

A *kurzor* olyan eszköz, amellyel megnevezhetjük a környezeti területet, segítségével hozzáférhetünk az ott elhelyezett információkhoz és amennyiben az aktív halmaz több sort tartalmaz, azokat egyenként elérhetjük, feldolgozhatjuk.

A PL/SQL kétfajta kurzort kezel, az *explicit* és az *implicit* kurzort. A PL/SQL automatikusan felépít egy implicit kurzort minden DML utasításhoz, beleértve az olyan lekérdezéseket is, amelyek pontosan egy sort adnak vissza. A több sort visszaadó (pontosabban az akárhány sort visszaadó) lekérdezések eredményének kezeléséhez viszont explicit kurzort célszerű használnunk.

Egy explicit kurzor kezelésének négy lépése van, ezek az alábbiak:

- kurzor deklarálása;
- kurzor megnyitása;
- sorok betöltése PL/SQL változókba;
- kurzor lezárása.

1. Kurzorok

1.1. Kurzor deklarálása

Egy kurzordeklaráció elhelyezhető blokk, alprogram vagy csomag deklarációs részében. A kurzor deklarációjának formája a következő:

```
CURSOR név [(paraméter[,paraméter]…)]

[RETURN sortípus] IS select_utasítás;
```

A *név* megnevezi a kurzort. A *paraméter* a kurzor formális paramétere. Alakja:

```
paraméter_név [IN] típus [{:=|DEFAULT} kifejezés]
```

Az alprogramok formális paramétereinél elmondottak itt is érvényesek. A paraméterek lokálisak a kurzorra nézve és szerepelhetnek az IS után megadott SELECT utasításban minden olyan helyen, ahol konstans szerepelhet.

A *sortípus* a kurzor által szolgáltatott érték típusa, amely rekord vagy adatbázistábla sorának

típusa lehet. Alakja:

```
{{ab_tábla_név|kurzor_név|kurzorváltozó_név}%ROWTYPE|

rekord_név%TYPE|rekordtípus_név}
```

Az *ab_tábla_név* egy olyan adatbázisbeli tábla vagy nézet neve, amelyik a deklarációnál ismert.

A *kurzor_név* egy korábban deklarált explicit kurzor, a *kurzorváltozó_név* egy kurzorváltozó neve.

A *rekord_név* egy korábban deklarált rekord neve.

A *rekordtípus_név* egy korábban deklarált RECORD típus neve.

A *select_utasítás* egy INTO utasításrészt nem tartalmazó SELECT utasítás, amely a kurzor által feldolgozható sorokat állítja elő. A kurzor paraméterei csak itt használhatók fel.

Példák

```
/* Megadja az ügyfeleket ábécé sorrendben.

Van RETURN utasításrész. */

CURSOR cur_ugyfelek RETURN ugyfel%ROWTYPE IS

SELECT * FROM ugyfel

ORDER BY UPPER(nev);

v_Uid ugyfel.id%TYPE;

/* Megadja annak az ügyfélnek a nevét és telefonszámát, melynek

azonosítóját egy blokkbeli változó tartalmazza.

Nincs RETURN utasításrész, hisz a kérdésből ez úgyis kiderül. */

CURSOR cur_ugyfel1 IS

SELECT nev, tel_szam FROM ugyfel

WHERE id = v_Uid;

/* Megadja a paraméterként átadott azonosítóval

rendelkező ügyfelet. Ha nincs paraméter, akkor

a megadott kezdőérték érvényes. */

CURSOR cur_ugyfel2(p_Uid ugyfel.id%TYPE DEFAULT v_Uid) IS

SELECT * FROM ugyfel

WHERE id = p_Uid;

/* Megadja az adott dátum szerint lejárt

kölcsönzésekhez az ügyfél nevét, a könyv címét,

valamint a lejárat óta eltelt napok számának

egész részét.

Ha nem adunk meg dátumot, akkor az aktuális

dátum lesz a kezdeti érték. */

CURSOR cur_lejart_kolcsonzesek(

p_Datum DATE DEFAULT SYSDATE

) IS

SELECT napok, u.nev, k.cim

FROM ugyfel u, konyv k,

(SELECT TRUNC(p_Datum, 'DD') - TRUNC(datum)

- 30*(hosszabbitva+1) AS napok,

kolcsonzo, konyv
```

```
FROM kolcsonzes) uk
WHERE uk.kolcsonzo = u.id
AND uk.konyv = k.id
AND napok > 0
ORDER BY UPPER(u.nev), UPPER(k.cim)
;
/* Lekérdezi és zárolja az adott azonosítójú könyv sorát.
Nem az egész táblát zárolja, csak az aktív halmaz elemeit!
Erre akkor lehet például szükség, ha egy könyv kölcsönzésénél
ellenőrizzük, hogy van-e még példány.
Így biztosan nem lesz gond, ha két kölcsönzés egyszerre történik
ugyanarra a könyvre. Az egyik biztosan bevárja a másikat. */
CURSOR cur_konyvzarolo(p_Kid konyv.id%TYPE) IS
SELECT * FROM konyv
WHERE id = p_Kid
FOR UPDATE OF cim;
/* Kisérletet tesz az adott könyv zárolására.
Ha az erőforrást foglaltsága miatt nem
lehet megnyitni, ORA-00054 kivétel váltódik ki. */
CURSOR cur_konyvzarolo2(p_Kid konyv.id%TYPE) IS
SELECT * FROM konyv
WHERE id = p_Kid
FOR UPDATE NOWAIT;
/* Ezek a változók kompatibilisek az előző kurzorokkal.
Döntse el, melyik kurzor melyik változóval kompatibilis! */
v_Ugyfel ugyfel%ROWTYPE;
v_Konyv konyv%ROWTYPE;
v_Unev ugyfel.nev%TYPE;
v_Utel_szam ugyfel.tel_szam%TYPE;
```

1.2. Kurzor megnyitása

A kurzor megnyitásánál lefut a kurzorhoz rendelt lekérdezés, meghatározódik az aktív halmaz és az aktív halmazhoz rendelt kurzormutató az első sorra áll rá. Ha a SELECT utasításban van FOR UPDATE utasításrész, akkor az aktív halmaz sorai zárolódnak. A megnyitást a következő utasítással végezhetjük:

```
OPEN kurzor_név [(aktuális_paraméter_lista)];
```

Az aktuális és formális paraméterek viszonyára és a paraméterátadásra vonatkozó információkat a 6.2. alfejezet tartalmazza.

Megnyitott kurzort újra megnyitni nem lehet. Megnyitott kurzorra kiadott OPEN utasítás a CURSOR_ALREADY_OPEN kivételt váltja ki. A megnyitott kurzor neve nem szerepeltethető kurzor FOR ciklusban.

Példák

```
BEGIN

v_Uid := 15; -- József István ügyfél azonosítója

/* Mely sorok lesznek az aktív halmaz elemei ? */

OPEN cur_ugyfel1;

OPEN cur_ugyfel2(15);

/* Mivel a paraméter mindig IN típusú,

kifejezés is lehet aktuális paraméter. */

OPEN cur_lejart_kolcsonzesek(TO_DATE('02-MÁJ. -09'));

BEGIN

OPEN cur_lejart_kolcsonzesek; -- CURSOR_ALREADY_OPEN kivételt vált ki!

EXCEPTION

WHEN CURSOR_ALREADY_OPEN THEN

DBMS_OUTPUT.PUT_LINE('Hiba: ' || SQLERRM);

END;

⋮

END;
```

1.3. Sorok betöltése

Az aktív halmaz sorainak feldolgozását a FETCH utasítás teszi lehetővé, melynek alakja:

```
FETCH {kurzor_név|kurzorváltozó_név}

{INTO{rekord_név|változó_név[,változó_név]…}|

BULK COLLECT INTO kollekciónév[,kollekciónév]…

LIMIT sorok};
```

A FETCH utasítás az adott kurzorhoz vagy kurzorváltozóhoz (*lásd* 8.2. alfejezet) tartozó kurzormutató által címzett sort betölti a rekordba vagy a megadott skalárváltozókba, és a kurzormutatót a következő sorra állítja. A skalárváltozókba a sor oszlopainak értéke kerül, a változók és oszlopok típusának kompatibilisnek kell lenniük. Rekord megadása esetén az oszlopok és mezők típusa kell kompatibilis legyen. A skalárváltozók száma, illetve a rekord mezőinek száma meg kell egyezzen az oszlopok számával.

A BULK COLLECT utasításrészt a 12. fejezetben tárgyaljuk. Nem megnyitott kurzor vagy kurzorváltozó esetén a FETCH utasítás az INVALID_CURSOR kivételt váltja ki.

Ha a FETCH utasítást az utolsó sor feldolgozása után adjuk ki, akkor a változó vagy a rekord előző értéke megmarad. Nem létező sor betöltése nem vált ki kivételt. Ezen szituáció ellenőrzésére használjuk a %FOUND, %NOTFOUND attribútumokat (*lásd* 8.3. alfejezet).

Példa

⋮

```
LOOP

FETCH cur_ugyfel1 INTO v_Unev, v_Utel_szam;

EXIT WHEN cur_ugyfel1%NOTFOUND;

/* Itt jön a feldolgozás, kiíratjuk a neveket. */

DBMS_OUTPUT.PUT_LINE(v_Unev || ', ' || v_Utel_szam);

END LOOP;

⋮
```

1.4. Kurzor lezárása

A kurzor lezárása érvényteleníti a kurzor vagy kurzorváltozó és az aktív halmaz közötti kapcsolatot és megszünteti a kurzormutatót. A kurzor lezárása a CLOSE utasítással történik, melynek alakja:

```
CLOSE {kurzornév|kurzorváltozó_név};
```

Lezárni csak megnyitott kurzort vagy kurzorváltozót lehet, különben az INVALID_CURSOR kivétel váltódik ki.

1. példa

```
⋮

CLOSE cur_ugyfel1;

CLOSE cur_ugyfel2;

⋮
```

2. példa (Az előző kurzorpéldák egy blokkban és további kurzorpéldák)

```
DECLARE

/* Megadja az ügyfeleket ábécé sorrendben.

Van RETURN utasításrész. */

CURSOR cur_ugyfelek RETURN ugyfel%ROWTYPE IS

SELECT * FROM ugyfel

ORDER BY UPPER(nev);

v_Uid ugyfel.id%TYPE;

/* Megadja annak az ügyfélnek nevét és telefonszámát, melynek

azonosítóját egy blokkbeli változó tartalmazza.

Nincs RETURN utasításrész, hisz a kérdésből ez úgyis kiderül. */

CURSOR cur_ugyfel1 IS

SELECT nev, tel_szam FROM ugyfel

WHERE id = v_Uid;

/* Megadja a paraméterként átadott azonosítóval

rendelkező ügyfelet. Ha nincs paraméter, akkor

a megadott kezdőérték érvényes. */

CURSOR cur_ugyfel2(p_Uid ugyfel.id%TYPE DEFAULT v_Uid) IS
```

```
SELECT * FROM ugyfel
WHERE id = p_Uid;
/* Megadja az adott dátum szerint lejárt
kölcsönzésekhez az ügyfél nevét, a könyv címét,
valamint a lejárat óta eltelt napok számának
egész részét.
Ha nem adunk meg dátumot, akkor az aktuális
dátum lesz a kezdeti érték. */
CURSOR cur_lejart_kolcsonzesek(
p_Datum DATE DEFAULT SYSDATE
) IS
SELECT napok, u.nev, k.cim
FROM ugyfel u, konyv k,
(SELECT TRUNC(p_Datum, 'DD') - TRUNC(datum)
- 30*(hosszabbitva+1) AS napok,
kolcsonzo, konyv
FROM kolcsonzes) uk
WHERE uk.kolcsonzo = u.id
AND uk.konyv = k.id
AND napok > 0
ORDER BY UPPER(u.nev), UPPER(k.cim)
;
/* Egy megfelelő típusú változóba lehet majd a kurzor sorait betölteni */
v_Lejart cur_lejart_kolcsonzesek%ROWTYPE;
v_Nev v_Lejart.nev%TYPE;
/* Lekérdezi és zárolja az adott azonosítójú könyv sorát.
Nem az egész táblát zárolja, csak az aktív halmaz elemeit!
Erre akkor lehet például szükség, ha egy könyv kölcsönzésénél
ellenőrizzük, hogy van-e még példány.
Így biztosan nem lesz gond, ha két kölcsönzés egyszerre történik
ugyanarra a könyvre. Az egyik biztosan bevárja a másikat. */
CURSOR cur_konyvzarolo(p_Kid konyv.id%TYPE) IS
SELECT * FROM konyv
WHERE id = p_Kid
FOR UPDATE OF cim;
/* Kísérletet tesz az adott könyv zárolására.
```

```
Ha az erőforrást foglaltsága miatt nem
lehet megnyitni, ORA-00054 kivétel váltódik ki. */
CURSOR cur_konyvzarolo2(p_Kid konyv.id%TYPE) IS
SELECT * FROM konyv
WHERE id = p_Kid
FOR UPDATE NOWAIT;
/* Ezek a változók kompatibilisek az előző kurzorokkal.
Döntse el, melyik kurzor melyik változóval kompatibilis! */
v_Ugyfel ugyfel%ROWTYPE;
v_Konyv konyv%ROWTYPE;
v_Unev ugyfel.nev%TYPE;
v_Utel_szam ugyfel.tel_szam%TYPE;
BEGIN
v_Uid := 15; -- József István ügyfél azonosítója
/* Mely sorok lesznek az aktív halmaz elemei? */
OPEN cur_ugyfel1;
OPEN cur_ugyfel2(15);
LOOP
FETCH cur_ugyfel1 INTO v_Unev, v_Utel_szam;
EXIT WHEN cur_ugyfel1%NOTFOUND;
/* Itt jön a feldolgozás, kiíratjuk a neveket. */
DBMS_OUTPUT.PUT_LINE(v_Unev || ', ' || v_Utel_szam);
END LOOP;
CLOSE cur_ugyfel1;
CLOSE cur_ugyfel2;
DBMS_OUTPUT.NEW_LINE;
/* Mivel a paraméter mindig IN típusú,
kifejezés is lehet aktuális paraméter. */
OPEN cur_lejart_kolcsonzesek(TO_DATE('02-MÁJ. -09'));
BEGIN
OPEN cur_lejart_kolcsonzesek; -- CURSOR_ALREADY_OPEN kivételt vált ki!
EXCEPTION
WHEN CURSOR_ALREADY_OPEN THEN
DBMS_OUTPUT.PUT_LINE('Hiba: ' || SQLERRM);
END;
v_Nev := NULL;
```

```
LOOP
FETCH cur_lejart_kolcsonzesek INTO v_Lejart;
EXIT WHEN cur_lejart_kolcsonzesek%NOTFOUND;
/* Jöhet a feldolgozás, mondjuk figyelmeztető e-mail küldése.
Most csak kiírjuk az egyes nevekhez a lejárt könyveket. */
IF v_Nev IS NULL OR v_Nev <> v_Lejart.nev THEN
v_Nev := v_Lejart.nev;
DBMS_OUTPUT.NEW_LINE;
DBMS_OUTPUT.PUT_LINE('Ügyfél: ' || v_Nev);
END IF;
DBMS_OUTPUT.PUT_LINE(' ' || v_Lejart.napok || ' nap, '
|| v_Lejart.cim);
END LOOP;
CLOSE cur_lejart_kolcsonzesek;
END;
/
/*
Eredmény:
József István, 06-52-456654
Hiba: ORA-06511: PL/SQL: a kurzor már meg van nyitva
Ügyfél: Jaripekka Hämälainen
22 nap, A critical introduction to twentieth-century American drama -
Volume 2
22 nap, The Norton Anthology of American Literature - Second Edition -
Volume 2
Ügyfél: József István
17 nap, ECOOP 2001 - Object-Oriented Programming
17 nap, Piszkos Fred és a többiek
A PL/SQL eljárás sikeresen befejeződött.
*/
```

3. példa (FOR UPDATE használatára)

```
DECLARE
/* Kísérletet tesz az adott könyv zárolására.
Ha az erőforrást foglaltsága miatt nem
lehet megnyitni, ORA-00054 kivétel váltódik ki. */
```

```
CURSOR cur_konyvzarolo2(p_Kid konyv.id%TYPE) IS
SELECT * FROM konyv
WHERE id = p_Kid
FOR UPDATE NOWAIT;
probak NUMBER;
t NUMBER;
foglalt EXCEPTION;
PRAGMA EXCEPTION_INIT(foglalt, -54);
BEGIN
probak := 0;
/* Legfeljebb 10-szer próbáljuk megnyitni. */
DBMS_OUTPUT.PUT_LINE(SYSTIMESTAMP);
WHILE probak < 10 LOOP
probak := probak + 1;
BEGIN
OPEN cur_konyvzarolo2(15);
-- Sikerült! Kilépünk a ciklusból
EXIT;
EXCEPTION
WHEN foglalt THEN
NULL;
END;
/* Várunk kb. 10 ms-t. */
t := TO_CHAR(SYSTIMESTAMP, 'SSSSSFF');
WHILE t + 10**8 > TO_CHAR(SYSTIMESTAMP, 'SSSSSFF') LOOP
NULL;
END LOOP;
END LOOP;
DBMS_OUTPUT.PUT_LINE(SYSTIMESTAMP);
DBMS_OUTPUT.PUT_LINE('A kurzort '
|| CASE cur_konyvzarolo2%ISOPEN WHEN TRUE THEN '' ELSE 'nem ' END
|| 'sikerült megnyitni.');
IF cur_konyvzarolo2%ISOPEN THEN
CLOSE cur_konyvzarolo2;
END IF;
END;
```

```
/
```

Próbálja ki az utolsó példát a következő két szituációban:

a. Egyetlen munkamenettel kapcsolódjon az adatbázishoz. Futtassa le a blokkot. Mit tapasztal?

b. Két munkamenettel kapcsolódjon az adatbázishoz, például úgy, hogy két SQL*Plusszal kapcsolódik. Az egyikben módosítsa a *konyv* tábla sorát:

```
UPDATE konyv SET cim = 'Próba' WHERE id = 15;
```

A másik munkamenetben futtassa le a blokkot! Mit tapasztal? Ezután az első munkamenetben adjon ki egy ROLLBACK parancsot, a második munkamenetben pedig futtassa le ismét a blokkot! Mit tapasztal?

1.5. Az implicit kurzor

A PL/SQL minden DML utasításhoz (azon lekérdezésekhez is, amelyeket nem explicit kurzorral kezelünk) felépít egy implicit kurzort. Ennek neve: SQL, ezért az implicit kurzort gyakran SQL kurzornak is hívják. Az implicit kurzorra nem alkalmazhatók az OPEN, FETCH, CLOSE utasítások. A megnyitást, betöltést, lezárást automatikusan a PL/SQL motor végzi.

2. Kurzorváltozók

Az explicit kurzorhoz hasonlóan a kurzorváltozó is egy aktív halmaz valamelyik sorának feldolgozására alkalmas. Amíg azonban a kurzor *statikus*, azaz egyetlen, a fordításkor már ismert SELECT utasításhoz kötődik, addig a kurzorváltozó *dinamikus*, amelyhez futási időben bármely típuskompatibilis kérdés hozzákapcsolható.

A kurzorváltozó használata az explicit kurzorok használatával megegyező módon történik (deklaráció, megnyitás, betöltés, lezárás).

A kurzorváltozó lényegében egy *referencia* típusú változó, amely mindig a hivatkozott sor *címét* tartalmazza.

A kurzorváltozó deklarálása két lépésben történik. Először létre kell hozni egy REF CURSOR saját típust, majd ezzel a típussal deklarálni egy változót. A típusdeklaráció szintaxisa:

```
TYPE név IS REF CURSOR [RETURN

{{ab_tábla_név|kurzor_név|kurzorváltozó_név}%ROWTYPE|

rekord_név%TYPE|

rekordtípus_név|

kurzorreferenciatípus_név}];
```

Az *ab_tábla_név* olyan adatbázisbeli tábla vagy nézet neve, amelyik ismert a deklarációnál.

A *kurzor_név* egy korábban deklarált explicit kurzor, a *kurzorváltozó_név* egy kurzorváltozó neve.

A *rekord_név* egy korábban deklarált rekord neve.

A *rekordtípus_név* egy korábban deklarált RECORD típus neve.

A *kurzorreferenciatípus_név* egy már deklarált REF CURSOR típus.

Ha szerepel a RETURN utasításrész, akkor *erős*, egyébként *gyenge* kurzorreferencia típusról beszélhetünk. A kettő között a különbség abban áll, hogy az erős kurzorreferencia

típusnál a fordító tudja ellenőrizni a kapcsolt kérdés típuskompatibilitását, míg a gyengénél bármely kérdés kapcsolható.

A PL/SQL jelen verziója tartalmaz egy előre definiált gyenge kurzorreferencia típust,

ennek neve SYS_REFCURSOR. Saját gyenge kurzorreferencia típus deklarálása helyett javasoljuk ennek használatát.

Az általunk deklarált vagy előre definiált kurzorreferencia típussal aztán kurzorváltozót tudunk deklarálni.

1. példa

```
DECLARE

TYPE t_egyed IS RECORD ...;

/* gyenge típusú REF CURSOR */

TYPE t_refcursor IS REF CURSOR;

/* erős típusú REF CURSOR */

TYPE t_ref_egyed IS REF CURSOR RETURN t_egyed;

/* Kurzorváltozók */

v_Refcursor1 t_refcursor;

v_Refcursor2 SYS_REFCURSOR; -- ez is gyenge típusú

v_Egyedek1 t_ref_egyed;

v_Egyedek2 t_ref_egyed;

⋮
```

A kurzorváltozót is meg kell nyitni. A kurzorváltozóhoz a megnyitáskor kapcsolódik hozzá az aktív halmazt meghatározó lekérdezés. Az utasítás alakja:

```
OPEN kurzorváltozó_név FOR select_utasítás;
```

Egy adott kurzorváltozó bármennyi OPEN-FOR utasításban szerepelhet, egy újabb megnyitás előtt nem kell lezárni. Egy kurzorváltozó újramegnyitása azt jelenti, hogy egy új aktív halmaz jön létre, amelyre a kurzorváltozó hivatkozni fog és az előző törlődik.

A kurzorváltozó betöltése és lezárása a kurzoroknál tárgyalt FETCH és CLOSE utasításokkal történik.

Ha a kurzorváltozó egy alprogram formális paramétere és az alprogram csak betölti és lezárja azt, akkor a paraméterátadás IN vagy IN OUT lehet. Ha a megnyitása is az alprogramban történik, akkor a mód kötelezően IN OUT.

2. példa

```
DECLARE

TYPE t_egyed IS RECORD (

id NUMBER,

leiras VARCHAR2(100)

);

/* erős típusú REF CURSOR */

TYPE t_ref_egyed IS REF CURSOR RETURN t_egyed;

/* gyenge típusú REF CURSOR típust nem kell deklarálnunk,

a SYS_REFCURSOR típust használjuk helyette */

/* Kurzorváltozók */
```

```
v_Refcursor SYS_REFCURSOR;
v_Egyedek1 t_ref_egyed;
v_Egyedek2 t_ref_egyed;
v_Egyed t_egyed;
/* Megnyit egy gyengén típusos kurzort. */
PROCEDURE megnyit_konyv(p_cursor IN OUT SYS_REFCURSOR) IS
BEGIN
OPEN p_cursor FOR
SELECT id, cim FROM konyv;
END;
/* Megnyit egy erősen típusos kurzort. */
FUNCTION megnyit_ugyfel RETURN t_ref_egyed IS
rv t_ref_egyed;
BEGIN
OPEN rv FOR
SELECT id, nev FROM ugyfel;
RETURN rv;
END;
/* Egy sort betölt a kurzorból és visszaadja.
A visszatérési érték másolása miatt nem
célszerű használni. */
FUNCTION betolt(p_cursor IN t_ref_egyed)
RETURN t_egyed IS
rv t_egyed;
BEGIN
FETCH p_cursor INTO rv;
RETURN rv;
END;
/* Bezár egy tetszőleges kurzort. */
PROCEDURE bezar(p_cursor IN SYS_REFCURSOR) IS
BEGIN
IF p_cursor%ISOPEN THEN
CLOSE p_cursor;
END IF;
END;
BEGIN
```

```
/* Elemezze a típuskompatibilitási problémákat! */
megnyit_konyv(v_Egyedek1);
v_Refcursor := megnyit_ugyfel;
/* Innentől kezdve a v_Refcursor és a v_Egyedek2 ugyanazt
a kurzort jelenti! */
v_Egyedek2 := v_Refcursor;
v_Egyed := betolt(v_Egyedek2);
DBMS_OUTPUT.PUT_LINE(v_Egyed.id || ', ' || v_Egyed.leiras);
v_Egyed := betolt(v_Refcursor);
DBMS_OUTPUT.PUT_LINE(v_Egyed.id || ', ' || v_Egyed.leiras);
/* Most pedig v_Refcursor és a v_Egyedek1 egyezik meg. */
v_Refcursor := v_Egyedek1;
v_Egyed := betolt(v_Egyedek1);
DBMS_OUTPUT.PUT_LINE(v_Egyed.id || ', ' || v_Egyed.leiras);
v_Egyed := betolt(v_Refcursor);
DBMS_OUTPUT.PUT_LINE(v_Egyed.id || ', ' || v_Egyed.leiras);
bezar(v_Egyedek1);
bezar(v_Egyedek2);
BEGIN
/* Ezt a kurzort már bezártuk egyszer v_Egyedek1 néven! */
v_Egyed := betolt(v_Refcursor);
DBMS_OUTPUT.PUT_LINE(v_Egyed.id || ', ' || v_Egyed.leiras);
EXCEPTION
WHEN INVALID_CURSOR THEN
DBMS_OUTPUT.PUT_LINE('Tényleg be volt zárva!');
END;
/* Itt nem szerepelhetne sem v_Egyedek1, sem v_Egyedek2. */
OPEN v_Refcursor FOR
SELECT 'alma',2,3,4 FROM DUAL;
/* Ez az értékadás most kompatibilitási problémát eredményez!
Az előbb nem okozott hibát, mert az ellenőrzés futási
időben történik. */
v_Egyedek2 := v_Refcursor;
EXCEPTION
WHEN OTHERS THEN
CLOSE v_Refcursor;
```

```
RAISE;

END;

/

/*

Eredmény:

5, Kovács János

10, Szabó Máté István

5, A római jog története és institúciói

10, A teljesség felé

Tényleg be volt zárva!

DECLARE

*

Hiba a(z) 1. sorban:

ORA-06504: PL/SQL: Az Eredményhalmaz-változók vagy a kérdés visszaadott

típusai nem illeszkednek

ORA-06512: a(z) helyen a(z) 99. sornál

*/
```

A PL/SQL-ben kezelhetők ún. CURSOR kifejezések is. A *CURSOR kifejezés* egy beágyazott kérdés sorait kezeli kurzor segítségével. Ilyenkor a kurzor által kezelt aktív halmaz sorai értékeket és kurzorokat tartalmaznak. A feldolgozás során először mindig az aktív halmaz sorai töltődnek be, majd egy beágyazott ciklus segítségével a beágyazott kurzorok alapján a beágyazott kérdések sorai.

Egy CURSOR kifejezés szerepelhet kurzordeklaráció SELECT-jében vagy kurzorváltozóhoz kapcsolt SELECT-ben. Alakja:

```
CURSOR(select_utasítás)
```

Egy beágyazott kurzor implicit módon nyílik meg, amikor a szülő kurzor valamelyik sora betöltésre kerül. A beágyazott kurzor lezáródik, ha:

- a szülő kurzor új sort tölt be;
- a szülő kurzor lezáródik;
- a szülő kurzorral történő betöltésnél hiba keletkezik.

3. példa (CURSOR kifejezés használatára)

```
DECLARE

TYPE t_Konyv_rec IS RECORD (

id konyv.id%TYPE,

cim konyv.cim%TYPE

);

TYPE t_konyvref IS REF CURSOR RETURN t_Konyv_rec;

/* Lekérdezzük az ügyfeleket és a kölcsönzött könyveiket,
```

```
azt az ügyfélt is, akinél nincs könyv.
A könyveket egy CURSOR kifejezés segítségével adjuk vissza. */
CURSOR cur_ugyfel_konyv IS
SELECT id, nev,
CURSOR(SELECT k.id, k.cim FROM konyv k, TABLE(konyvek) uk
WHERE k.id = uk.konyv_id) AS konyvlista
FROM ugyfel
ORDER BY UPPER(nev);
v_Uid ugyfel.id%TYPE;
v_Unev ugyfel.nev%TYPE;
v_Konyvek t_konyvref;
v_Konyv t_Konyv_rec;
BEGIN
OPEN cur_ugyfel_konyv;
LOOP
FETCH cur_ugyfel_konyv INTO v_Uid, v_Unev, v_Konyvek;
EXIT WHEN cur_ugyfel_konyv%NOTFOUND;
DBMS_OUTPUT.NEW_LINE;
DBMS_OUTPUT.PUT_LINE('Ügyfél: ' || v_Uid || ', ' || v_Unev);
/* Most a beágyazott kurzor elemeit írjuk ki, ha nem üres.
A beágyazott kurzort nem kell külön megnyitni és lezárni sem. */
FETCH v_Konyvek INTO v_Konyv;
IF v_Konyvek%FOUND THEN
DBMS_OUTPUT.PUT_LINE(' A kölcsönzött könyvek:');
WHILE v_Konyvek%FOUND LOOP
DBMS_OUTPUT.PUT_LINE(' ' || v_Konyv.id || ', ' || v_Konyv.cim);
FETCH v_Konyvek INTO v_Konyv;
END LOOP;
ELSE
DBMS_OUTPUT.PUT_LINE(' jelenleg nem kölcsönöz könyvet.');
END IF;
END LOOP;
CLOSE cur_ugyfel_konyv;
END;
/
/*
```

```
Eredmény:
Ügyfél: 25, Erdei Anita
A kölcsönzött könyvek:
35, A critical introduction to twentieth-century American drama -
Volume 2
Ügyfél: 35, Jaripekka Hämälainen
A kölcsönzött könyvek:
35, A critical introduction to twentieth-century American drama -
Volume 2
40, The Norton Anthology of American Literature - Second Edition -
Volume 2
Ügyfél: 15, József István
A kölcsönzött könyvek:
15, Piszkos Fred és a többiek
20, ECOOP 2001 - Object-Oriented Programming
25, Java - start!
45, Matematikai zseblexikon
50, Matematikai Kézikönyv
Ügyfél: 30, Komor Ágnes
A kölcsönzött könyvek:
5, A római jog története és institúciói
10, A teljesség felé
Ügyfél: 5, Kovács János
jelenleg nem kölcsönöz könyvet.
Ügyfél: 10, Szabó Máté István
A kölcsönzött könyvek:
30, SQL:1999 Understanding Relational Language Components
45, Matematikai zseblexikon
50, Matematikai Kézikönyv
Ügyfél: 20, Tóth László
A kölcsönzött könyvek:
30, SQL:1999 Understanding Relational Language Components
A PL/SQL eljárás sikeresen befejeződött.
*/
```

Az Oracle10*g* a kurzorváltozók használatára az alábbi korlátozásokat alkalmazza:

- Kurzorváltozó nem deklarálható csomagban.
- Távoli alprogramoknak nem adható át kurzorváltozó értéke.
- Kurzorreferencia típusú gazdaváltozó PL/SQL-nek való átadása esetén a szerveroldali betöltés csak akkor lehetséges, ha a megnyitás is ugyanazon szerverhívásban történt.
- A kurzorváltozónak nem adható NULL érték.
- Kurzorváltozók értéke nem hasonlítható össze.
- Adatbázistábla oszlopában nem tárolható kurzorreferencia típusú érték.
- Kollekció eleme nem lehet kurzorreferencia típusú érték.

3. Kurzorattribútumok

Négy olyan attribútum van, amelyik az explicit és implicit kurzorokra és a kurzorváltozókra alkalmazható. Formálisan a kurzor vagy kurzorváltozó neve után állnak. Ezek az attribútumok a DML utasítás végrehajtásáról szolgáltatnak információkat. Csak procedurális utasításokban használhatók, SQL utasításokban nem. Lássuk ezeket az attribútumokat explicit kurzorok és kurzorváltozók esetén.

%FOUND

A kurzor vagy kurzorváltozó megnyitása után közvetlenül (az első betöltése előtt), értéke NULL. Sikeres sorbetöltés esetén értéke TRUE, egyébként FALSE.

%ISOPEN

Értéke TRUE, ha a kurzor vagy kurzorváltozó meg van nyitva, egyébként FALSE.

%NOTFOUND

A %FOUND logikai ellentettje. Értéke a kurzor vagy kurzorváltozó megnyitása után közvetlenül (az első sor betöltése előtt) NULL. Sikeres sorbetöltés esetén értéke FALSE, egyébként TRUE.

%ROWCOUNT

Értéke közvetlenül a megnyitás után, az első sor betöltése előtt 0. Ezután értéke minden egyes sikeres sorbetöltés után 1-gyel nő, tehát a mindenkori betöltött sorok darabszámát adja meg. Ha az explicit kurzor vagy kurzorváltozó nincs megnyitva, a %FOUND, %NOTFOUND és %ROWCOUNT alkalmazása az INVALID_CURSOR kivételt váltja ki.

Példák

```
/*

A könyvtár úgy döntött, hogy minden olyan

ügyfele, aki már több mint egy éve iratkozott be,

legalább 10 könyvet kölcsönözhet.

Szeretnénk elvégezni a szükséges változtatásokat úgy,

hogy a változásokról feljegyzésünk legyen egy szöveges

állományban.

A megoldásunk egy olyan PL/SQL blokk, amely elvégzi a

változtatásokat és közben ezekről információt ír

a képernyőre. Így az eredmény elmenthető (spool).
```

```
A program nem tartalmaz tranzakciókezelő utasításokat,
így a változtatott sorok a legközelebbi ROLLBACK vagy
COMMIT utasításig zárolva lesznek.
*/
DECLARE
/* Az ügyfelek lekérdezése. */
CURSOR cur_ugyfelek(p_Datum DATE DEFAULT SYSDATE) IS
SELECT * FROM ugyfel
WHERE p_Datum - beiratkozas >= 365
AND max_konyv < 10
ORDER BY UPPER(nev)
FOR UPDATE OF max_konyv;
v_Ugyfel cur_ugyfelek%ROWTYPE;
v_Ma DATE;
v_Sum NUMBER := 0;
BEGIN
DBMS_OUTPUT.PUT_LINE('Kölcsönzési kvóta emelése');
DBMS_OUTPUT.PUT_LINE('-------------------------');
/* A példa kedvéért rögzítjük a mai dátum értékét. */
v_Ma := TO_DATE('2002-05-02 09:01:12', 'YYYY-MM-DD HH24:MI:SS');
OPEN cur_ugyfelek(v_Ma);
LOOP
FETCH cur_ugyfelek INTO v_Ugyfel;
EXIT WHEN cur_ugyfelek%NOTFOUND;
/* A módosítandó rekordot a kurzorral azonosítjuk. */
UPDATE ugyfel SET max_konyv = 10
WHERE CURRENT OF cur_ugyfelek;
DBMS_OUTPUT.PUT_LINE(cur_ugyfelek%ROWCOUNT
|| ', ügyfél: ' || v_Ugyfel.id || ', ' || v_Ugyfel.nev
|| ', beíratkozott: ' || TO_CHAR(v_Ugyfel.beiratkozas, 'YYYY-MON-DD')
|| ', régi érték: ' || v_Ugyfel.max_konyv || ', új érték: 10');
v_Sum := v_Sum + 10 - v_Ugyfel.max_konyv;
END LOOP;
DBMS_OUTPUT.PUT_LINE('-------------------------');
DBMS_OUTPUT.PUT_LINE('Összesen ' || cur_ugyfelek%ROWCOUNT
|| ' ügyfél adata változott meg.');
```

```
DBMS_OUTPUT.PUT_LINE('Így ügyfeleink összesen ' || v_Sum
|| ' könyvvel kölcsönözhetnek többet ezután.');
DBMS_OUTPUT.PUT_LINE('Dátum: '
|| TO_CHAR(v_Ma, 'YYYY-MON-DD HH24:MI:SS'));
CLOSE cur_ugyfelek;
EXCEPTION
WHEN OTHERS THEN
IF cur_ugyfelek%ISOPEN THEN
CLOSE cur_ugyfelek;
END IF;
RAISE;
END;
/
/*
Eredmény (első alkalommal, rögzítettük a mai dátumot: 2002. május 2.):
Kölcsönzési kvóta emelése
-------------------------
1, ügyfél: 25, Erdei Anita, beíratkozott: 1997-DEC. -05, régi érték: 5,
új érték: 10
2, ügyfél: 30, Komor Ágnes, beíratkozott: 2000-JÚN. -11, régi érték: 5,
új érték: 10
3, ügyfél: 20, Tóth László, beíratkozott: 1996-ÁPR. -01, régi érték: 5,
új érték: 10
-------------------------
Összesen 3 ügyfél adata változott meg.
Így ügyfeleink összesen 15 könyvvel kölcsönözhetnek többet ezután.
2002-MÁJ. -02 09:01:12
A PL/SQL eljárás sikeresen befejeződött.
*/
```

4. Az implicit kurzor attribútumai

Az implicit kurzor attribútumai az INSERT, DELETE, UPDATE és SELECT INTO utasítások végrehajtásáról adnak információt. Az információ mindig a legutoljára végrehajtott utasításra vonatkozik. Mielőtt az Oracle megnyitja az implicit kurzort, az attribútumok értéke NULL.

%FOUND

Az utasítás végrehajtása alatt értéke NULL. Utána értéke TRUE, ha az INSERT, DELETE, UPDATE utasítás egy vagy több sort érintett, illetve a SELECT INTO legalább egy sort visszaadott. Különben értéke FALSE.

%ISOPEN

Az Oracle automatikusan lezárja az implicit kurzort az utasítás végrehajtása után, ezért értéke mindig FALSE.

%NOTFOUND

Értéke TRUE, ha az INSERT, DELETE, UPDATE egyetlen sorra sem volt hatással, illetve ha SELECT INTO nem adott vissza sort, egyébként FALSE.

%ROWCOUNT

Értéke az INSERT, DELETE, UPDATE utasítások által kezelt sorok darabszáma. A SELECT INTO utasítás esetén értéke 0, ha nincs visszaadott sor és 1, ha van. Ha a SELECT INTO utasítás egynél több sort ad vissza, akkor a TOO_MANY_ROWS kivétel váltódik ki. Ekkor tehát a %ROWCOUNT **nem** a visszaadott sorok tényleges számát tartalmazza.

Az implicit kurzor rendelkezik még két további attribútummal (%BULK_ROWCOUNT, %BULK_EXCEPTIONS), ezeket a 12. fejezetben tárgyaljuk.

Példák

```
BEGIN

/* Az SQL-kurzor attribútumai, az SQL%ISOPEN kivételével

NULL-t adnak vissza első híváskor. */

IF SQL%FOUND IS NULL

AND SQL%NOTFOUND IS NULL AND SQL%ROWCOUNT IS NULL THEN

DBMS_OUTPUT.PUT_LINE('Az attribútumok NULL-t adtak.');

END IF;

IF NOT SQL%ISOPEN THEN

DBMS_OUTPUT.PUT_LINE('Az SQL%ISOPEN hamis.');

END IF;

/* Néhány könyvből utánpótlást kapott a könyvtár. */

UPDATE konyv SET keszlet = keszlet + 5, szabad = szabad + 5

WHERE id IN (45, 50);

IF (SQL%FOUND) THEN

DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT || ' rekord módosítva.');

ELSE

DBMS_OUTPUT.PUT_LINE('Nincsenek ilyen könyveink.');

END IF;

/* Most nem létező könyveket adunk meg. */

UPDATE konyv SET keszlet = keszlet + 5, szabad = szabad + 5

WHERE id IN (-45, -50);

IF (SQL%FOUND) THEN

DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT || ' rekord módosítva.');

ELSE
```

```
DBMS_OUTPUT.PUT_LINE('Nincsenek ilyen könyveink.');
END IF;
DECLARE
i NUMBER;
BEGIN
/* Ez bizony sok lesz. */
SELECT id INTO i FROM ugyfel
WHERE id = id;
EXCEPTION
WHEN OTHERS THEN
DBMS_OUTPUT.PUT_LINE('SQL%ROWCOUNT: ' || SQL%ROWCOUNT);
END;
DECLARE
i NUMBER;
BEGIN
/* Iyen nem lesz. */
SELECT id INTO i FROM ugyfel
WHERE id < 0;
EXCEPTION
WHEN OTHERS THEN
DBMS_OUTPUT.PUT_LINE('SQL%ROWCOUNT: ' || SQL%ROWCOUNT);
END;
END;
/
/*
Eredmény:
Az attribútumok NULL-t adtak.
Az SQL%ISOPEN hamis.
2 rekord módosítva.
Nincsenek ilyen könyveink.
SQL%ROWCOUNT: 1
SQL%ROWCOUNT: 0
A PL/SQL eljárás sikeresen befejeződött.
*/
```

Implicit kurzor esetén külön figyelni kell arra, hogy az attribútum által visszaadott eredmény mindig az időben utolsó SQL műveletre vonatkozik:

```
DECLARE
v_Temp NUMBER;
/* Eljárás, ami implicit kurzort használ. */
PROCEDURE alprg IS
i NUMBER;
BEGIN
SELECT 1 INTO i FROM DUAL;
END;
BEGIN
/* Ez a DELETE nem töröl egy sort sem. */
DELETE FROM konyv
WHERE 1 = 2;
/* Nem biztonságos használat! Az alprogramhívás megváltoztathatja az
implicit attribútumok értékét, mert azok mindig a legutolsó
SQL-utasításra vonatkoznak. */
alprg;
DBMS_OUTPUT.PUT_LINE('SQL%ROWCOUNT: ' || SQL%ROWCOUNT);
/* Ez a DELETE nem töröl egy sort sem. */
DELETE FROM konyv
WHERE 1 = 2;
/* Az a biztonságos, ha a szükséges attribútumok értékét
ideiglenesen tároljuk. */
v_Temp := SQL%ROWCOUNT;
alprg;
DBMS_OUTPUT.PUT_LINE('SQL%ROWCOUNT: ' || v_Temp);
END;
/
/*
Eredmény:
SQL%ROWCOUNT: 1
SQL%ROWCOUNT: 0
A PL/SQL eljárás sikeresen befejeződött.
*/
```

A 8.1. táblázat azt összegzi, hogy az egyes kurzorattribútumok mit adnak vissza az OPEN, FETCH és CLOSE utasítások végrehajtása előtt és után.

8.1. táblázat - Kurzorattribútumok értékei

		%FOUND	%ISOPEN	%NOTFOUND	%ROWCOUNT
OPEN	Előtt	Kivétel	FALSE	Kivétel	Kivétel
	Után	NULL	TRUE	NULL	0
Első	FETCH	Előtt	NULL	TRUE NULL	0
	Után	TRUE	TRUE	FALSE	1
Következő	Előtt	TRUE	RUE	FALSE	1
FETCH(-ek)	Után	TRUE	TRUE	FALSE	Az adatoktól függ
Utolsó FETCH	Előtt	TRUE	TRUE	FALSE	Az adatoktól függ
	Után	FALSE	TRUE	TRUE	Az adatoktól függ
CLOSE	Előtt	FALSE	TRUE	TRUE	Az adatoktól függ
	Után	Kivétel	FALSE	Kivétel	Kivétel

Megjegyzések:

1. Az INVALID_CURSOR kivétel váltódik ki, ha a %FOUND, %NOTFOUND vagy %ROWCOUNT attribútumokra hivatkozunk a kurzor megnyitása előtt, vagy a kurzor lezárása után.

2. Ha a kurzor nem adott vissza egy sort sem, akkor az első FETCH után a %FOUND értéke FALSE, a %NOTFOUND értéke TRUE, és a %ROWCOUNT értéke 0 lesz.

5. Kurzor FOR ciklus

Az explicit kurzor használatát a kurzor FOR ciklus alkalmazásával egyszerűsíthetjük. A kurzor FOR ciklus implicit módon %ROWTYPE típusúnak deklarálja a ciklusváltozóját, megnyitja a kurzort, betölti rendre az aktív halmaz összes sorát, majd ezután lezárja a kurzort. Alakja:

```
FOR ciklusváltozó IN {kurzornév [(paraméterek)]

|(select-utasítás)}

LOOP utasítás [utasítás]… END LOOP;
```

1. példa

```
DECLARE

/* Megadja az adott dátum szerint lejárt

kölcsönzésekhez az ügyfél nevét, a könyv címét,

valamint a lejárat óta eltelt napok számának

egész részét.
```

```
Ha nem adunk meg dátumot, akkor az aktuális
dátum lesz a kezdeti érték. */
CURSOR cur_lejart_kolcsonzesek(
p_Datum DATE DEFAULT SYSDATE
) IS
SELECT napok, u.nev, k.cim
FROM ugyfel u, konyv k,
(SELECT TRUNC(p_Datum, 'DD') - TRUNC(datum)
- 30*(hosszabbitva+1) AS napok,
kolcsonzo, konyv
FROM kolcsonzes) uk
WHERE uk.kolcsonzo = u.id
AND uk.konyv = k.id
AND napok > 0
ORDER BY UPPER(u.nev), UPPER(k.cim)
;
v_Nev ugyfel.nev%TYPE;
BEGIN
/* Hasonlítsuk össze ezt a megoldást a korábbival!
Vegyük észre, hogy a ciklusváltozó implicit módon van deklarálva. */
FOR v_Lejart IN cur_lejart_kolcsonzesek(TO_DATE('02-MÁJ. -09')) LOOP
/* Nincs FETCH és %NOTFOUND-ra sincs szükség. */
IF v_Nev IS NULL OR v_Nev <> v_Lejart.nev THEN
v_Nev := v_Lejart.nev;
DBMS_OUTPUT.NEW_LINE;
DBMS_OUTPUT.PUT_LINE('Ügyfél: ' || v_Nev);
END IF;
DBMS_OUTPUT.PUT_LINE(' ' || v_Lejart.napok || ' nap, '
|| v_Lejart.cim);
END LOOP;
/* A kurzor már le van zárva. */
END;
/
/*
Eredmény:
Ügyfél: Jaripekka Hämälainen
```

```
22 nap, A critical introduction to twentieth-century American drama -

Volume 2

22 nap, The Norton Anthology of American Literature - Second Edition -

Volume 2

Ügyfél: József István

17 nap, ECOOP 2001 - Object-Oriented Programming

17 nap, Piszkos Fred és a többiek

A PL/SQL eljárás sikeresen befejeződött.

*/
```

Explicit kurzor helyett SQL lekérdezést is használhatunk. Ilyenkor egy rejtett kurzor keletkezik, ami a programozó számára hozzáférhetetlen. Ez nem az implicit kurzor lesz, ezért az implicit attribútumok nem adnak róla információt.

2. példa

```
BEGIN

/* Explicit kurzor helyett SQL lekérdezést is használhatunk.

Ilyenkor egy rejtett kurzor keletkezik, ami a programozó számára

hozzáférhetetlen, azaz az implicit attribútumok nem

adnak róla információt. */

FOR v_Ugyfel IN (SELECT * FROM ugyfel) LOOP

DBMS_OUTPUT.PUT_LINE(v_Ugyfel.id || ', ' || v_Ugyfel.nev);

IF SQL%ROWCOUNT IS NOT NULL THEN

/* Ide NEM kerül a vezérlés, mert a SQL%ROWCOUNT NULL. */

DBMS_OUTPUT.PUT_LINE('SQL%ROWCOUNT: ' || SQL%ROWCOUNT);

END IF;

END LOOP;

END;

/

/*

Eredmény:

5, Kovács János

10, Szabó Máté István

15, József István

20, Tóth László

25, Erdei Anita

30, Komor Ágnes

35, Jaripekka Hämälainen
```

```
A PL/SQL eljárás sikeresen befejeződött.

*/
```

9. fejezet - Tárolt alprogramok

A 6.2. alfejezetben megismerkedtünk az alprogramokkal. Az ott tárgyalt alprogramok más programegységek *lokális* alprogramjai voltak. Lehetőség van azonban arra, hogy az alprogramokat *adatbázis-objektumokként* kezeljük (ugyanúgy, mint például a táblákat). Ekkor az SQL szokásos létrehozó, módosító és törlő DDL utasításait használhatjuk. A tárolt alprogramok lefordított formában, ún. *p-kódban* tárolódnak. A PL/SQL fordító a kódból előállít egy belső kódot, amit aztán az adatszótárban tárol. A *p*-kód a PL/SQL virtuális gép (PVM), az interpreter utasításait tartalmazza. Amikor egy tárolt alprogram kódját végre kell hajtani, a PVM ezt a *p*-kódot fogja értelmezni úgy, hogy megfelelő alacsony szintű API hívásokat hajt végre.

Tipp

Megjegyzés: *Az Oracle 10*g *az* m-kód *terminológiát használja, de mi kifejezőbbnek tartjuk a korábbi verziókban használt* p-*kód elnevezést, így ezt tartottuk meg.*

Az Oracle10*g* lehetőséget ad arra, hogy a PL/SQL fordító natív operációs rendszer kódra

fordítson. Ehhez a rendszerben telepíteni kell egy C fordítót és ezután a PL/SQL fordító

C kódot generál, amelyből a C fordító által előállított kódot az Oracle háttérfolyamata

futtatja majd. Részletesen a 16.5. alfejezetben tárgyaljuk ezt a lehetőséget.

A tárolt alprogramokról információkat elsősorban a következő adatszótárnézetekből nyerhetünk:

USER_OBJECTS, USER_SOURCE (a forráskódot tartalmazza), USER_ERRORS (a

fordítási hibákat tartalmazza).

SQL*Plusban a fordítási hibákat a SHOW ERRORS paranccsal is megnézhetjük, illetve

meggyőződhetünk arról, hogy nem volt hiba a tárolt alprogram fordítása során.

Tárolt eljárás létrehozása a következő SQL parancs segítségével történik:

```
CREATE [OR REPLACE] eljárásfej
[AUTHID {DEFINER|CURRENT_USER}]
eljárás_törzs
```

Az eljárás fejére és törzsére a 6.2. alfejezetben leírtak vonatkoznak.

A OR REPLACE újragenerálja az eljárást, ha az már létezik. Lényegében az eljárás definíciójának

megváltoztatására szolgál, így az eljáráshoz előzőleg már megadott objektumjogosultságokat

nem kell törölni, újra létrehozni és újra adományozni.

Az AUTHID segítségével megadható, hogy az eljárás létrehozójának (DEFINER – ez az

alapértelmezés), vagy aktuális hívójának (CURRENT_USER) a jogosultságai érvényesek-e a

hívásnál.

1. példa (Tekintsük a következő két eljárást, ahol a VISSZAHOZ eljárás a létrehozó jogosultságaival, a VISSZAHOZ_CURRENT_USER eljárás az aktuális hívó jogosultságaival van definiálva. A két eljárás működése minden másban megegyezik.)

```
CREATE OR REPLACE PROCEDURE visszahoz(
p_Konyv konyv.id%TYPE,
```

```
p_Kolcsonzo ugyfel.id%TYPE
)
AS
/* Ez az eljárás adminisztrálja egy könyv visszahozatalát.
Azaz törli a rekordot a kölcsönzések közül (ha több egyező is van,
akkor egy tetszőlegeset), valamint növeli a könyv szabad példányszámát.
-20020-as számú felhasználói kivétel jelzi, ha nem létezik
a kölcsönzési rekord.
*/
v_Datum kolcsonzes.datum%TYPE;
BEGIN
DELETE FROM kolcsonzes
WHERE konyv = p_Konyv
AND kolcsonzo = p_Kolcsonzo
AND ROWNUM = 1
RETURNING datum INTO v_Datum;
IF SQL%ROWCOUNT = 0 THEN
RAISE_APPLICATION_ERROR(-20020,
'Nem létezik ilyen kölcsönzési bejegyzés');
END IF;
UPDATE konyv SET szabad = szabad + 1
WHERE id = p_Konyv;
DELETE FROM TABLE(SELECT konyvek FROM ugyfel WHERE id = p_Kolcsonzo)
WHERE konyv_id = p_Konyv
AND datum = v_Datum;
END;
/
show errors
CREATE OR REPLACE PROCEDURE visszahoz_current_user(
p_Konyv konyv.id%TYPE,
p_Kolcsonzo ugyfel.id%TYPE
)
AUTHID CURRENT_USER
AS
/* Ez az eljárás adminisztrálja egy könyv visszahozatalát.
Azaz törli a rekordot a kölcsönzések közül (ha több egyező is van,
```

```
akkor egy tetszőlegeset), valamint növeli a könyv szabad példányszámát.
-20020-as számú felhasználói kivétel jelzi, ha nem létezik
a kölcsönzési rekord.
*/
v_Datum kolcsonzes.datum%TYPE;
BEGIN
DELETE FROM kolcsonzes
WHERE konyv = p_Konyv
AND kolcsonzo = p_Kolcsonzo
AND ROWNUM = 1
RETURNING datum INTO v_Datum;
IF SQL%ROWCOUNT = 0 THEN
RAISE_APPLICATION_ERROR(-20020,
'Nem létezik ilyen kölcsönzési bejegyzés');
END IF;
UPDATE konyv SET szabad = szabad + 1
WHERE id = p_Konyv;
DELETE FROM TABLE(SELECT konyvek FROM ugyfel WHERE id = p_Kolcsonzo)
WHERE konyv_id = p_Konyv
AND datum = v_Datum;
END;
/
show errors
```

PLSQL nevű felhasználóként hozzuk létre a VISSZAHOZ és a VISSZAHOZ_CURRENT_USER eljárásokat. Ezután hozzunk létre egy új felhasználót PLSQL2 néven. PLSQL felhasználóként mindkét eljárásra adjunk futtatási jogot a PLSQL2 felhasználónak:

```
GRANT EXECUTE ON visszahoz TO plsql2;
GRANT EXECUTE ON visszahoz_current_user TO plsql2;
```

Futtassuk az eljárásokat PLSQL2 felhasználóként és vizsgáljuk meg az eredményt:

```
SQL> CALL plsql.visszahoz(1,1);
CALL plsql.visszahoz(1,1)
*
Hiba a(z) 1. sorban:
ORA-20020: Nem létezik ilyen kölcsönzési bejegyzés
ORA-06512: a(z) "PLSQL.VISSZAHOZ", helyen a(z) 25. sornál
```

```
SQL> CALL plsql.visszahoz_current_user(1,1);

CALL plsql.visszahoz_current_user(1,1)

*

Hiba a(z) 1. sorban:

ORA-00942: a tábla vagy a nézet nem létezik

ORA-06512: a(z) "PLSQL.VISSZAHOZ_CURRENT_USER", helyen a(z) 18. sornál
```

Az első hívás során nem létező kölcsönzést adtunk meg, ez okozta, egyébként helyesen, a hibát.

A második hívás során más hibaüzenetet kaptunk, ennek az az oka, hogy a VISSZAHOZ_CURRENT_USER eljárás a hívó jogosultságaival fut. Így a rendszer az eljárásban szereplő táblák nevét a hívó, a PLSQL2 felhasználó nevében és jogosultságaival próbálja meg feloldani, sikertelenül.

Az első hívás során a PLSQL2 felhasználó hozzáfér a megfelelő adatbázis-objektumokhoz, annak ellenére hogy a táblákra semmilyen jogosultsága nincs. Ez a hozzáférés azonban jól definiáltan történik a VISSZAHOZ eljárás hívásával.

Alprogramjainkat a létrehozó jogaival futtatva előírhatjuk az alkalmazások használóinak, hogy csak egy általunk meghatározott, szabályos módon férjenek hozzá az adatbázisban tárolt objektumokhoz.

Az aktuális hívó jogaival definiált alprogramokat használhatjuk általános célú alprogramok készítésére olyan esetekben, amikor a tevékenység a hívó felhasználó objektumaihoz kötődik.

A következő SQL parancs újrafordít egy tárolt eljárást:

```
ALTER PROCEDURE eljárásnév COMPILE [DEBUG];
```

A DEBUG megadása azt írja elő, hogy a PL/SQL fordító a p-kód generálásánál használja a nyomkövetőt. A

```
DROP PROCEDURE eljárásnév;
```

törli az adatbázisból a megadott nevű tárolt eljárást.

Egy tárolt függvényt a következő SQL paranccsal hozhatunk létre.

```
CREATE [OR REPLACE] függvényfej

[AUTHID {DEFINER|CURRENT_USER}]

[DETERMINISTIC]

függvénytörzs
```

A függvény fejére és törzsére a 6.2. alfejezetben leírtak vonatkoznak.

A DETERMINISTIC egy optimalizálási előírás, amely a redundáns függvényhívások elkerülését szolgálja. Megadása esetén a függvény visszatérési értékéről másolat készül, és ha a függvényt ugyanazokkal az aktuális paraméterekkel hívjuk meg, az optimalizáló ezt a másolatot fogja használni.

Az egyéb utasításrészek jelentése megegyezik a tárolt eljárásnál ismertetett utasításrészekével.

2. példa (A FAKTORIALIS függvény determinisztikus megadása)

```
CREATE OR REPLACE FUNCTION faktorialis(

n NUMBER

)

RETURN NUMBER

DETERMINISTIC
```

```
AS

rv NUMBER;

BEGIN

IF n < 0 THEN

RAISE_APPLICATION_ERROR(-20001, 'Hibás paraméter!');

END IF;

rv := 1;

FOR i IN 1..n LOOP

rv := rv * i;

END LOOP;

RETURN rv;

END faktorialis;

/

show errors
```

A következő két tárolt függvény nem a paramétereitől függ, hanem az adatbázis tartalmától:

```
CREATE OR REPLACE FUNCTION aktiv_kolcsonzo

RETURN NUMBER

AS

/* Megadja azon ügyfelek számát, akik jelenleg

kölcsönöznek könyvet. */

rv NUMBER;

BEGIN

SELECT COUNT(1)

INTO rv

FROM (SELECT 1 FROM kolcsonzes GROUP BY kolcsonzo);

RETURN rv;

END aktiv_kolcsonzo;

/

show errors

CREATE OR REPLACE FUNCTION osszes_kolcsonzo

RETURN NUMBER

AS

/* Megadja a beíratkozott ügyfelek számát. */

rv NUMBER;

BEGIN
```

```
SELECT COUNT(1)
INTO rv
FROM ugyfel;
RETURN rv;
END osszes_kolcsonzo;
/
show errors
```

Egy tárolt függvényt újrafordítani az

```
ALTER FUNCTION függvénynév COMPILE [DEBUG];
```

paranccsal, törölni a

```
DROP FUNCTION függvénynév;
```

paranccsal lehet.

A tárolt alprogramok SQL-ben a következő paranccsal hívhatók meg:

```
CALL alprogram_név([aktuális_paraméter_lista])
[INTO gazdaváltozó];
```

A *gazdaváltozó* a visszatérési értéket tárolja, ha az alprogram függvény.

3. példa

```
/*
Gazdaváltozók az SQL*Plus futtató környezetben.
*/
VARIABLE v_Osszes NUMBER;
VARIABLE v_Aktiv NUMBER;
/*
Tárolt alprogramok hívása
*/
CALL osszes_kolcsonzo() INTO :v_Osszes;
CALL aktiv_kolcsonzo() INTO :v_Aktiv;
/*
Arány számolása és kiíratása
*/
PROMPT A jelenleg aktív ügyfelek aránya a kölcsönzők körében:
SELECT TO_CHAR(:v_Aktiv*100/:v_Osszes, '999.99') || '%' AS "Arány"
FROM dual;
```

Amikor egy tárolt alprogram lefordításra kerül, akkor az adatszótárban bejegyzés készül

az összes olyan adatbázis-objektumról, amelyre az alprogram hivatkozik. Az alprogram **függ** ezektől az objektumoktól. Ha az objektumok valamelyike megváltozik (végrehajtódik rajta egy DDL utasítás), akkor az alprogram *érvénytelenné* válik. Ha egy érvénytelen alprogramot hívunk meg, akkor a PL/SQL motor automatikusan újrafordítja azt futási időben. Ezt kerülhetjük el a tárolt alprogram explicit újrafordításával, az ALTER parancs alkalmazásával.

4. példa

```
CREATE OR REPLACE FUNCTION kombinacio(

n NUMBER,

m NUMBER

)

RETURN NUMBER

DETERMINISTIC

AS

BEGIN

RETURN faktorialis(n)/faktorialis(n-m)/faktorialis(m);

END kombinacio;

/

show errors
```

A KOMBINACIO tárolt függvény használja, és ezért függ az előzőleg már definiált FAKTORIALIS függvénytől. Ha a FAKTORIALIS függvényt lecseréljük egy másik implementációra, akkor a függőség miatt vagy nekünk, vagy a PL/SQL motornak újra kell fordítania a KOMBINACIO függvényt:

```
CREATE OR REPLACE FUNCTION faktorialis(

n NUMBER

)

RETURN NUMBER

DETERMINISTIC

AS

BEGIN

IF n = 0 THEN

RETURN 1;

ELSIF n < 0 THEN

RAISE_APPLICATION_ERROR(-20001, 'Hibás paraméter!');

ELSE

RETURN n*faktorialis(n-1);

END IF;

END faktorialis;

/

show errors
```

```
ALTER FUNCTION kombinacio COMPILE;
```

Ha az alprogram és az objektum, amelytől függ, ugyanazon adatbázisban van, akkor az objektum módosítása az alprogram azonnali érvénytelenítését jelenti. Ha azonban az objektum egy távoli adatbázis eleme, akkor ez nem így történik. Távoli adatbázisban elhelyezett objektum érvényességének ellenőrzése futási időben, a hivatkozásnál történik meg. Az Oracle két különböző modellt alkalmaz ebben a helyzetben.

Időbélyegmodell

Ekkor az objektum és az alprogram utolsó módosításának időbélyegét hasonlítja össze. Ezeket az időbélyegeket a USER_OBJECTS adatszótárnézet LAST_DDL_TIME mezője tartalmazza. Ha a hivatkozott objektum időbélyege későbbi, akkor az alprogramot újrafordítja. Ez az alapértelmezett mód.

Szignatúramodell

Amikor létrehozunk egy tárolt alprogramot, az adatszótárban p-kódban tárolódik annak *szignatúrája*. Ez a formális paraméterek sorrendjét és típusát tartalmazza. Ez a modell alprogramok egymás közötti függőségeinek kezelésénél használható. Ha ugyanis egy P1 alprogram meghív egy P2 alprogramot, akkor a P2 szignatúrája a P1 első fordításánál elhelyeződik a P1 információi között. P1-et újrafordítani ezek után csak akkor kell, ha P2 szignatúrája megváltozik.

Ezen modell használatához a REMOTE_DEPENDENCIES_MODE rendszerparamétert SIGNATURE értékre kell állítani.

Az SQL utasításokban meghívott tárolt függvények nem módosíthatják a lekérdezés vagy módosítás alatt álló táblákat (*lásd* 13.6. alfejezet).

Egy függvény *hozzáférési szintje* azt adja meg, hogy a függvény mely adatokat olvashatja vagy módosíthatja. A PL/SQL négy hozzáférési szintet értelmez, ezek a következők:

- WNDS (Writes No Database State). A függvény nem módosíthatja az adatbázis tábláit (nem használhat módosító utasítást).
- RNDS (Reads No Database State). A függvény nem olvashatja az adatbázis tábláit (nem használhat SELECT utasítást).
- WNPS (Writes No Package State). A függvény nem módosíthat csomagbeli változót (a változó nem szerepelhet értékadó utasítás bal oldalán vagy FETCH utasításban).
- RNPS (Reads No Package State). A függvény nem használhatja fel a csomagbeli változó értékét (a változó nem szerepelhet kifejezésben).

Egy függvény hozzáférési szintje a fordító által ellenőrizhető, ha megadjuk a következő

pragmát:

```
PRAGMA RESTRICT_REFERENCES({függvénynév|DEFAULT},

{RNDS|WNDS|RNPS|WNPS|TRUST}

[, {RNDS|WNDS|RNPS|WNPS|TRUST}]…);
```

A DEFAULT egy csomag vagy egy objektumtípus minden függvényéhez a megadott hozzáférési szintet rendeli. A *függvénynév* viszont csak az adott függvény hozzáférési szintjét határozza meg. Túlterhelt függvénynevek esetén a kódban a pragmát megelőző, a pragmához legközelebb eső deklarációra vonatkozik. A TRUST olyan függvények esetén használható, melyek implementációja nem PL/SQL nyelven történt (hanem például Java vagy C nyelven).

5. példa (Létrehozunk egy tárolt függvényt és azt SELECT utasításban használjuk)

```
CREATE OR REPLACE FUNCTION hatralevo_napok(

p_Kolcsonzo kolcsonzes.kolcsonzo%TYPE,
```

```
p_Konyv kolcsonzes.konyv%TYPE
) RETURN INTEGER
AS
/* Megadja egy kölcsönzött könyv hátralevő kölcsönzési idejét. */
v_Datum kolcsonzes.datum%TYPE;
v_Most DATE;
v_Hosszabbitva kolcsonzes.hosszabbitva%TYPE;
BEGIN
SELECT datum, hosszabbitva
INTO v_Datum, v_Hosszabbitva
FROM kolcsonzes
WHERE kolcsonzo = p_Kolcsonzo
AND konyv = p_Konyv;
/* Levágjuk a dátumokból az óra részt. */
v_Datum := TRUNC(v_Datum, 'DD');
v_Most := TRUNC(SYSDATE, 'DD');
/* Visszaadjuk a különbséget. */
RETURN (v_Hosszabbitva + 1) * 30 + v_Datum - v_Most;
EXCEPTION
WHEN NO_DATA_FOUND THEN
RETURN NULL;
END hatralevo_napok;
/
show errors
SELECT ugyfel, konyv, hatralevo_napok(ugyfel_id, konyv_id) AS hatra
FROM ugyfel_konyv
WHERE ugyfel_id = 10 -- Szabó Máté István
/
/*
Eredmény: (A példa futtatásakor a dátum: 2002. május 9.)
UGYFEL
------------------------------------------------------------
KONYV HATRA
------------------------------------------------------------ ----------
Szabó Máté István
Matematikai Kézikönyv 12
```

```
Szabó Máté István

Matematikai zseblexikon 12

Szabó Máté István

SQL:1999 Understanding Relational Language Components 12

Megjegyzés:

~~~~~~~~~~~

Adatbázispéldány szinten lehetőség van a SYSDATE értékét

rögzíteni a FIXED_DATE inicializációs paraméterrel

(ehhez ALTER SYSYTEM jogosultság szükséges):

ALTER SYSTEM SET FIXED_DATE='2002-MÁJ. -09' SCOPE=MEMORY;

ALTER SYSTEM SET FIXED_DATE=NONE SCOPE=MEMORY;

*/
```

10. fejezet - Csomagok

A csomag egy olyan PL/SQL programegység, amely adatbázis-objektum és PL/SQL programozási eszközök gyűjteményének tekinthető. Két részből áll, *specifikációból* és *törzsből*. A törzs nem kötelező. A specifikáció és a törzs külön tárolódik az adatszótárban. A specifikáció egy *interfészt* biztosít, amelyen keresztül hozzáférhetünk a csomag eszközeihez. A specifikációban típusdefiníciók, nevesítettkonstans-deklarációk, változódeklarációk, kivételdeklarációk, kurzorspecifikációk, alprogram-specifikációk és pragmák helyezhetők el. Tehát a specifikáció futtatható kódot nem tartalmaz. A törzs tartalmazza a kurzorok és alprogramok teljes deklarációját, és esetlegesen végrehajtható utasításokat.

A csomagspecifikáció létrehozásának szintaxisa:

```
CREATE [OR REPLACE] PACKAGE csomagnév

[AUTHID {DEFINER|CURRENT_USER}]

{IS|AS}

{típus_definíció|

nevesített_konstans_deklaráció|

változó_deklaráció|

kivétel_deklaráció|

kurzor_specifikáció|

alprogram_specifikáció|

pragma}

END [csomagnév];
```

A törzs létrehozásának szintaxisa:

```
CREATE [OR REPLACE] PACKAGE BODY csomagnév

{IS|AS}

deklarációk

[BEGIN

végrehajtható_utasítások]

END [csomagnév];
```

A specifikáció és a törzs utasításrészeinek jelentése megegyezik az alprogramoknál tárgyalt CREATE utasítás utasításrészeinek jelentésével.

A csomag specifikációs részének deklarációi *nyilvánosak*, az itt deklarált eszközök láthatók és tetszés szerint alkalmazhatók a séma bármely alkalmazásában. A nyilvános eszközökre a csomag nevével történő minősítéssel lehet hivatkozni.

Az eszközök felsorolásának sorrendje tetszőleges, kivéve azt, hogy az alprogramoknak kötelezően a többi eszköz deklarációja után kell állniuk, mögöttük már csak pragma szerepelhet.

A PL/SQL hatásköri szabályai természetesen a csomagban is érvényesek, egy eszköz csak a deklaráció után látszik. Tehát ha egy csomagbeli eszközre hivatkozik egy másik csomagbeli eszköz, akkor azt hamarabb kell deklarálni. Természetesen a csomag teljes törzsében már minden olyan csomagbeli eszköz látszik, amelyet a specifikációban deklaráltunk.

Ha a specifikációban szerepel kurzorspecifikáció vagy alprogramspecifikáció, akkor kötelező megadni a törzset, ahol az adott eszközök teljes deklarációját (implementációját) meg kell adni.

A fejezet végén szerepel egy *konyvtar_csomag* nevű csomag. Ez a csomag deklarál egy nyilvános kurzort, amelynek implementációját a törzs tartalmazza:

```
CREATE OR REPLACE PACKAGE konyvtar_csomag

AS

⋮

CURSOR cur_lejart_kolcsonzesek(

p_Datum DATE DEFAULT SYSDATE

) RETURN t_lejart_rec;

⋮

END;

CREATE OR REPLACE PACKAGE BODY konyvtar_csomag

AS

⋮

/* Megadja a lejárt kölcsönzéseket. */

CURSOR cur_lejart_kolcsonzesek(

p_Datum DATE DEFAULT SYSDATE

) RETURN t_lejart_rec

IS

SELECT kolcsonzo, konyv, datum,

hosszabbitva, megjegyzes, napok

FROM

(SELECT kolcsonzo, konyv, datum, hosszabbitva, megjegyzes,

TRUNC(p_Datum, 'DD') - TRUNC(datum, 'DD')

- 30*(hosszabbitva+1) AS napok

FROM kolcsonzes)

WHERE napok > 0

;

⋮

END;
```

A törzsben szereplő *deklarációk* nem nyilvánosak (*privát* deklarációk), az itteni eszközök csak a törzsben használhatók.

A törzs végrehajtható utasításai tipikusan változók inicializációjára használatosak.

A fejezet végi *konyvtar_csomag* csomagban például a következő deklarációk privátak:

```
CREATE OR REPLACE PACKAGE BODY konyvtar_csomag

AS

⋮
```

```
v_Stat t_statisztika_rec;

⋮

/* Beállítja a v_Stat utolsó módosítási dátumát. */

PROCEDURE stat_akt

IS

BEGIN

v_Stat.utolso_modositas := SYSDATE;

END;

⋮

END;
```

Csomag létrehozásánál a PL/SQL fordító a tárolt alprogramoknál említett p-kódra fordítja le a specifikációt és a törzset. Előbb mindig a specifikációt kell létrehozni. A specifikáció és a törzs elkülönült kezelése lehetővé teszi, hogy a specifikáció változatlanul hagyása mellett a törzset lecseréljük. Az interfész marad, az implementáció megváltozik.

A csomag alprogramjainak nevei túlterhelhetők.

Távoli csomag változói sem közvetlenül, sem közvetett módon nem hivatkozhatók.

Egy csomagban nem használhatunk gazdaváltozót.

A csomag törzsének inicializációs része csak egyszer, a csomagra való első hivatkozáskor fut le.

Ha egy csomag eszközeit csak egyetlen szerverhívásban használjuk, akkor a SERIALLY_REUSABLE pragmával szeriálisan újrafelhasználhatónak minősíthetjük. A pragmát el kell helyezni mind a specifikációban, mind a törzsben. Ekkor a csomag számára a Rendszer Globális Területen (SGA) belül lesz memória lefoglalva, nem pedig az egyes felhasználói területeken, ezért a csomag munkaterülete újrafelhasználható. Az ilyen csomag nyilvános változói minden újrafelhasználás előtt újra kezdőértéket kapnak és lefut a csomag törzsének inicializációs része. Meg kell jegyezni, hogy egy szerverhívásban többször is hivatkozhatunk a csomag elemeire, azok a két hivatkozás között megőrzik értéküket.

A csomag mint *adatbázis-objektum* megváltoztatására szolgál a következő SQL-parancs:

```
ALTER PACKAGE csomagnév COMPILE [DEBUG]

[PACKAGE|SPECIFICATION|BODY];
```

A DEBUG előírja a PL/SQL fordítónak, hogy a p-kód generálásánál használja a PL/SQL nyomkövetőjét.

A PACKAGE (ez az alapértelmezés) megadása esetén a csomag specifikációja és törzse is újrafordítódik. SPECIFICATION esetén a specifikáció, BODY esetén a törzs újrafordítására kerül sor.

A specifikáció újrafordítása esetén minden olyan alkalmazást, amely hivatkozik a csomag eszközeire, újra kell fordítani.

A csomag törlésére a következő utasítás szolgál:

```
DROP PACKAGE [BODY] csomagnév;
```

BODY megadása esetén csak a törzs, egyébként a törzs és a specifikáció is törlődik az adatszótárból.

A PL/SQL nyelvi környezetét beépített csomag határozza meg, neve: STANDARD. Ez a csomag globálisan használható típusokat, kivételeket és alprogramokat tartalmaz, amelyek közvetlenül, minősítés nélkül hivatkozhatók minden alkalmazásban. A STANDARD csomag nagyon sok beépített függvénye túlterhelt. Például a TO_CHAR függvény következő specifikációi léteznek benne:

```
FUNCTION TO_CHAR (right DATE) RETURN VARCHAR2;

FUNCTION TO_CHAR (left NUMBER) RETURN VARCHAR2;

FUNCTION TO_CHAR (left DATE, right VARCHAR2) RETURN VARCHAR2;

FUNCTION TO_CHAR (left NUMBER, right VARCHAR2) RETURN VARCHAR2;
```

Az Oracle és a különböző Oracle-eszközök termékspecifikus beépített csomagokat bocsátanak a felhasználó rendelkezésére. Ezekkel sokkal hatékonyabban és gyorsabban lehet alkalmazásokat fejleszteni.

Ha saját csomagot írunk, törekedjünk arra, hogy azt a lehető legáltalánosabb formában hozzuk létre, hogy a későbbi újrafelhasználást elősegítsük. A csomagspecifikáció mindig az alkalmazás tervéhez kapcsolódjon, mindig ezt írjuk meg először és csak a feltétlenül szükséges eszközök szerepeljenek benne.

A következőkben megadunk egy saját csomagot, amely a könyvtárunk adatainak kezelését segíti elő. A csomag neve *konyvtar_csomag*. A csomag specifikációja a következő:

```
CREATE OR REPLACE PACKAGE konyvtar_csomag

AS

/*****************************************/

/* Mindenki számára elérhető deklarációk. */

/*****************************************/

TYPE t_statisztika_rec IS RECORD (

uj_ugyfelek NUMBER := 0,

uj_konyvek NUMBER := 0,

kolcsonzesek NUMBER := 0,

visszahozott_konyvek NUMBER := 0,

utolso_modositas DATE := SYSDATE,

utolso_nullazas DATE := SYSDATE

);

TYPE t_lejart_rec IS RECORD (

kolcsonzo kolcsonzes.kolcsonzo%TYPE,

konyv kolcsonzes.kolcsonzo%TYPE,

datum kolcsonzes.datum%TYPE,

hosszabbitva kolcsonzes.hosszabbitva%TYPE,

megjegyzes kolcsonzes.megjegyzes%TYPE,

napok INTEGER

);

/* Megadja az adott dátum szerint lejárt

kölcsönzésekhez a kölcsönzési rekordot és

a lejárat óta eltelt napok számát.

Ha nem adunk meg dátumot, akkor az aktuális

dátum lesz a kezdeti érték.
```

```
Megjegyzés: a törzs nélküli kurzor deklarációnak
visszatérési típussal kell rendelkeznie
*/
CURSOR cur_lejart_kolcsonzesek(
p_Datum DATE DEFAULT SYSDATE
) RETURN t_lejart_rec;
/* Egy új ügyfélnek mindig ennyi lesz a kvótája. */
c_Max_konyv_init CONSTANT NUMBER := 5;
/* Az alprogramok által kiváltott kivételek. */
hibas_argumentum EXCEPTION;
PRAGMA EXCEPTION_INIT(hibas_argumentum, -20100);
ervenytelen_kolcsonzes EXCEPTION;
PRAGMA EXCEPTION_INIT(ervenytelen_kolcsonzes, -20110);
kurzor_hasznalatban EXCEPTION;
PRAGMA EXCEPTION_INIT(ervenytelen_kolcsonzes, -20120);
/************************************************/
/* A könyvtárban használatos gyakori alprogramok */
/************************************************/
/*
Megadja az ügyfelet az azonosítóhoz.
Nemlétező ügyfél esetén hibas_argumentum kivételt vált ki.
*/
FUNCTION az_ugyfel(p_Ugyfel ugyfel.id%TYPE) RETURN ugyfel%ROWTYPE;
/*
Megadja a könyvet az azonosítóhoz.
Nemlétező könyv esetén hibas_argumentum kivételt vált ki.
*/
FUNCTION a_konyv(p_Konyv konyv.id%TYPE) RETURN konyv%ROWTYPE;
/*
Felvesz egy ügyfelet a könyvtárba a szokásos
kezdeti kvótával és az aktuális dátummal, a
kölcsönzött könyvek oszlopot üres kollekcióra állítja.
Visszatérési értéke az új ügyfél azonosítója.
*/
FUNCTION uj_ugyfel(
p_Nev ugyfel.nev%TYPE,
```

```
p_Anyja_neve ugyfel.anyja_neve%TYPE,
p_Lakcim ugyfel.lakcim%TYPE,
p_Tel_szam ugyfel.tel_szam%TYPE,
p_Foglalkozas ugyfel.foglalkozas%TYPE
) RETURN ugyfel.id%type;
/*
Raktárra vesz az adott ISBN kódú könyvből megadott példányt.
Ha van már ilyen könyv, akkor hozzáadja az új készletet
a régihez. Ilyenkor csak az ISBN szám kerül egyeztetésre
a többi paraméter értékét a függvény nem veszi figyelembe.
Visszatérési érték a könyvek azonosítója.
*/
FUNCTION konyv_raktarba(
p_ISBN konyv.ISBN%TYPE,
p_Cím konyv.cim%TYPE,
p_Kiado konyv.kiado%TYPE,
p_Kiadasi_ev konyv.kiadasi_ev%TYPE,
p_Szerzo konyv.szerzo%TYPE,
p_Darab INTEGER
) RETURN konyv.id%type;
/*
Egy ügyfél számára kölcsönöz egy könyvet:
- felveszi a kölcsönzési rekordot a kolcsonzes táblába
és az ugyfel.konyvek beágyazott táblába.
- aktualizálja az ügyfél kvótáját és a könyv szabad
példányainak számát.
Nemlétező könyv, illetve ügyfél esetén hibas_argumentum kivétel,
ha az ügyfélnek 0 a kvótája, vagy nincs szabad példány, akkor
ervenytelen_kolcsonzes kivétel váltódik ki.
*/
PROCEDURE kolcsonoz(
p_Ugyfel ugyfel.id%TYPE,
p_Konyv konyv.id%TYPE
);
/*
Adminisztrálja egy könyv visszahozatalát:
```

```
- naplózza a kölcsönzést a kolcsonzes_naplo táblában
- törli a kölcsönzési rekordot a kolcsonzes táblából
és az ugyfel.konyvek beágyazott táblából.
- aktualizálja a könyv szabad példányainak számát.
Ha a paraméterek nem jelölnek valódi kölcsönzési
bejegyzést, akkor hibas_argumentum kivétel váltódik ki.
*/
PROCEDURE visszahoz(
p_Ugyfel ugyfel.id%TYPE,
p_Konyv konyv.id%TYPE
);
/*
Adminisztrálja egy ügyfél összes könyvének visszahozatalát:
- naplózza a kölcsönzéseket a kolcsonzes_naplo táblában
- törli a kölcsönzési rekordokat a kolcsonzes táblából
és az ugyfel.konyvek beágyazott táblából.
- aktualizálja a könyvek szabad példányainak számát.
Ha az ügyfél nem létezik hibas_argumentum kivétel váltódik ki.
Megj.: Ez az eljárásnév túl van terhelve.
*/
PROCEDURE visszahoz(
p_Ugyfel ugyfel.id%TYPE
);
/*
Kilistázza a lejárt kölcsönzési rekordokat.
Ha a cur_lejart_kolcsonzesek kurzor nyitva van,
az eljárás nem használható. Ilyenkor
kurzor_hasznalatban kivétel váltódik ki.
*/
PROCEDURE lejart_konyvek;
/*
Kilistázza a lejárt kölcsönzési rekordok közül
azokat, amelyek ma jártak le.
Ha a cur_lejart_kolcsonzesek kurzor nyitva van,
az eljárás nem használható. Ilyenkor
kurzor_hasznalatban kivétel váltódik ki.
```

```
*/
PROCEDURE mai_lejart_konyvek;
/*
Megadja egy kölcsönzött könyv hátralevő kölcsönzési idejét.
hibas_argumentum kivételt vált ki, ha nincs ilyen kölcsönzés.
*/
FUNCTION hatralevo_napok(
p_Ugyfel ugyfel.id%TYPE,
p_Konyv konyv.id%TYPE
) RETURN INTEGER;
/*****************************************************************/
/* A csomag használatára vonatkozó statisztikát kezelő alprogramok */
/*****************************************************************/
/*
Megadja a statisztikát a legutóbbi nullázás, ill.
az indulás óta.
*/
FUNCTION statisztika RETURN t_statisztika_rec;
/*
Kiírja a statisztikát a legutóbbi nullázás ill.
az indulás óta.
*/
PROCEDURE print_statisztika;
/*
Nullázza a statisztikát.
*/
PROCEDURE statisztika_nullaz;
END konyvtar_csomag;
/
show errors
```

A csomag törzse:

```
CREATE OR REPLACE PACKAGE BODY konyvtar_csomag
AS
/**********************/
/* Privát deklarációk. */
```

```
/***********************/
/* A csomag statisztikáját tartalmazó privát rekord. */
v_Stat t_statisztika_rec;
/******************/
/* Implementációk. */
/******************/
/* Megadja a lejárt kölcsönzéseket. */
CURSOR cur_lejart_kolcsonzesek(
p_Datum DATE DEFAULT SYSDATE
) RETURN t_lejart_rec
IS
SELECT kolcsonzo, konyv, datum,
hosszabbitva, megjegyzes, napok
FROM
(SELECT kolcsonzo, konyv, datum, hosszabbitva, megjegyzes,
TRUNC(p_Datum, 'DD') - TRUNC(datum, 'DD')
- 30*(hosszabbitva+1) AS napok
FROM kolcsonzes)
WHERE napok > 0
;
/************************/
/* Privát alprogramok. */
/************************/
/* Beállítja a v_Stat utolsó módosítási dátumát. */
PROCEDURE stat_akt
IS
BEGIN
v_Stat.utolso_modositas := SYSDATE;
END stat_akt;
/* Pontosan p_Hossz hosszan adja meg a p_Sztringet, ha kell
csonkolja, ha kell szóközökkel egészíti ki. */
FUNCTION str(
p_Sztring VARCHAR2,
p_Hossz INTEGER
) RETURN VARCHAR2 IS
BEGIN
```

```
RETURN RPAD(SUBSTR(p_Sztring, 1, p_Hossz), p_Hossz);
END str;
/* Kivételek kiváltása hibaüzenetekkel. */
PROCEDURE raise_hibas_argumentum
IS
BEGIN
RAISE_APPLICATION_ERROR(-20100, 'Hibás argumentum');
END raise_hibas_argumentum;
PROCEDURE raise_ervenytelen_kolcsonzes
IS
BEGIN
RAISE_APPLICATION_ERROR(-20110, 'Érvénytelen kölcsönzés');
END raise_ervenytelen_kolcsonzes;
PROCEDURE raise_kurzor_hasznalatban
IS
BEGIN
RAISE_APPLICATION_ERROR(-20120, 'A kurzor használatban van.');
END raise_kurzor_hasznalatban;
/************************************************/
/* A könyvtárban használatos gyakori alprogramok */
/************************************************/
/*
Megadja az ügyfelet az azonosítóhoz.
Nemlétező ügyfél esetén hibas_argumentum kivételt vált ki.
*/
FUNCTION az_ugyfel(p_Ugyfel ugyfel.id%TYPE)
RETURN ugyfel%ROWTYPE IS
v_Ugyfel ugyfel%ROWTYPE;
BEGIN
SELECT * INTO v_Ugyfel
FROM ugyfel WHERE id = p_Ugyfel;
RETURN v_Ugyfel;
EXCEPTION
WHEN NO_DATA_FOUND THEN
raise_hibas_argumentum;
END az_ugyfel;
```

```
/*
Megadja a könyvet az azonosítóhoz.
Nemlétező könyv esetén hibas_argumentum kivételt vált ki.
*/
FUNCTION a_konyv(p_Konyv konyv.id%TYPE)
RETURN konyv%ROWTYPE IS
v_Konyv konyv%ROWTYPE;
BEGIN
SELECT * INTO v_Konyv
FROM konyv WHERE id = p_Konyv;
RETURN v_Konyv;
EXCEPTION
WHEN NO_DATA_FOUND THEN
raise_hibas_argumentum;
END a_konyv;
/*
Felvesz egy ügyfelet a könyvtárba a szokásos
kezdeti kvótával és az aktuális dátummal, a
kölcsönzött könyvek oszlopot üres kollekcióra állítja.
Visszatérési értéke az új ügyfél azonosítója.
*/
FUNCTION uj_ugyfel(
p_Nev ugyfel.nev%TYPE,
p_Anyja_neve ugyfel.anyja_neve%TYPE,
p_Lakcim ugyfel.lakcim%TYPE,
p_Tel_szam ugyfel.tel_szam%TYPE,
p_Foglalkozas ugyfel.foglalkozas%TYPE
) RETURN ugyfel.id%type IS
v_Id ugyfel.id%type;
BEGIN
SELECT ugyfel_seq.nextval
INTO v_Id
FROM dual;
INSERT INTO ugyfel VALUES
(v_Id, p_Nev, p_Anyja_neve,
p_Lakcim, p_Tel_szam, p_Foglalkozas,
```

```
SYSDATE, c_Max_konyv_init, T_Konyvek());
/* A statisztika aktualizálása. */
stat_akt;
v_Stat.uj_ugyfelek := v_Stat.uj_ugyfelek + 1;
RETURN v_Id;
END uj_ugyfel;
/*
Raktárra vesz az adott ISBN kódú könyvből megadott példányt.
Ha van már ilyen könyv, akkor hozzáadja az új készletet
a régihez. Ilyenkor csak az ISBN számot egyezteti,
a többi paraméter értékét a függvény nem veszi figyelembe.
Visszatérési érték a könyvek azonosítója.
*/
FUNCTION konyv_raktarba(
p_ISBN konyv.ISBN%TYPE,
p_Cím konyv.cim%TYPE,
p_Kiado konyv.kiado%TYPE,
p_Kiadasi_ev konyv.kiadasi_ev%TYPE,
p_Szerzo konyv.szerzo%TYPE,
p_Darab INTEGER
) RETURN konyv.id%type IS
v_Id konyv.id%type;
BEGIN
/* Megpróbáljuk módosítani a könyv adatait. */
UPDATE konyv
SET keszlet = keszlet + p_Darab,
szabad = szabad + p_Darab
WHERE
UPPER(ISBN) = UPPER(p_ISBN)
RETURNING id INTO v_Id;
IF SQL%NOTFOUND THEN
/* Mégiscsak INSERT lesz ez. */
SELECT konyv_seq.nextval
INTO v_Id
FROM dual;
INSERT INTO konyv VALUES
```

```
(v_Id, p_ISBN, p_Cim,
p_Kiado, p_Kiadasi_ev, p_Szerzo,
p_Darab, p_Darab);
/* A statisztika aktualizálása. */
stat_akt;
v_Stat.uj_konyvek := v_Stat.uj_konyvek + 1;
END IF;
RETURN v_Id;
END;
/*
Egy ügyfél számára kölcsönöz egy könyvet:
- felveszi a kölcsönzési rekordot a kolcsonzes táblába
és az ugyfel.konyvek beágyazott táblába;
- aktualizálja az ügyfél kvótáját és a könyv szabad
példányainak számát.
Nemlétező könyv, illetve ügyfél esetén hibas_argumentum kivétel,
ha az ügyfélnek 0 a kvótája, vagy nincs szabad példány, akkor
ervenytelen_kolcsonzes kivétel váltódik ki.
*/
PROCEDURE kolcsonoz(
p_Ugyfel ugyfel.id%TYPE,
p_Konyv konyv.id%TYPE
) IS
v_Most DATE;
v_Szam NUMBER;
BEGIN
SAVEPOINT kezdet;
v_Most := SYSDATE;
/* Próbáljuk meg beszúrni a kölcsönzési rekordot. */
BEGIN
INSERT INTO kolcsonzes VALUES (p_Ugyfel, p_Konyv, v_Most, 0,
'A bejegyzést a konyvtar_csomag hozta létre');
EXCEPTION
WHEN OTHERS THEN
-- Csak az integritási megszorítással lehetett baj.
raise_hibas_argumentum;
```

```
END;
BEGIN
/* A következő SELECT nem ad vissza sort,
ha nem lehet többet kölcsönöznie az ügyfélnek. */
SELECT 1 INTO v_Szam
FROM ugyfel
WHERE id = p_Ugyfel
AND max_konyv - (SELECT COUNT(1) FROM TABLE(konyvek)) > 0;
/* Ha csökkentjük a szabad példányok számát,
megsérthetjük a konyv_szabad megszorítást. */
UPDATE konyv SET szabad = szabad - 1 WHERE id = p_Konyv;
INSERT INTO TABLE(SELECT konyvek FROM ugyfel WHERE id = p_Ugyfel)
VALUES (p_Konyv, v_Most);
/* A statisztika aktualizálása. */
stat_akt;
EXCEPTION
WHEN OTHERS THEN -- kvóta vagy szabad példány nem megfelelő
ROLLBACK TO kezdet;
raise_ervenytelen_kolcsonzes;
END;
END kolcsonoz;
/*
Adminisztrálja egy könyv visszahozatalát:
- naplózza a kölcsönzést a kolcsonzes_naplo táblában;
- törli a kölcsönzési rekordot a kolcsonzes táblából
és az ugyfel.konyvek beágyazott táblából;
- aktualizálja a könyv szabad példányainak számát.
Ha a paraméterek nem jelölnek valódi kölcsönzési
bejegyzést, akkor hibas_argumentum kivétel váltódik ki.
*/
PROCEDURE visszahoz(
p_Ugyfel ugyfel.id%TYPE,
p_Konyv konyv.id%TYPE
) IS
v_Datum kolcsonzes.datum%TYPE;
BEGIN
```

```
DELETE FROM kolcsonzes
WHERE konyv = p_Konyv
AND kolcsonzo = p_Ugyfel
AND rownum = 1
RETURNING datum INTO v_Datum;
IF SQL%ROWCOUNT = 0 THEN
raise_hibas_argumentum;
END IF;
UPDATE konyv SET szabad = szabad + 1
WHERE id = p_Konyv;
DELETE FROM TABLE(SELECT konyvek FROM ugyfel WHERE id = p_Ugyfel)
WHERE konyv_id = p_konyv
AND datum = v_Datum;
/* A statisztika aktualizálása */
stat_akt;
v_Stat.visszahozott_konyvek := v_Stat.visszahozott_konyvek - 1;
END visszahoz;
/*
Adminisztrálja egy ügyfél összes könyvének visszahozatalát:
- naplózza a kölcsönzéseket a kolcsonzes_naplo táblában;
- törli a kölcsönzési rekordokat a kolcsonzes táblából
és az ugyfel.konyvek beágyazott táblából;
- aktualizálja a könyvek szabad példányainak számát.
Ha az ügyfél nem létezik, hibas_argumentum kivétel váltódik ki.
Megj.: Ez az eljárásnév túl van terhelve.
*/
PROCEDURE visszahoz(
p_Ugyfel ugyfel.id%TYPE
) IS
v_Szam NUMBER;
BEGIN
/* Az csoportfüggvényeket tartalmazó SELECT
mindig ad vissza sort. */
SELECT COUNT(1) INTO v_Szam
FROM ugyfel WHERE id = p_Ugyfel;
IF v_Szam = 0 THEN
```

```
raise_hibas_argumentum;
END IF;
/* Töröljük egyenként a kölcsönzéseket. */
FOR k IN (
SELECT * FROM kolcsonzes
WHERE kolcsonzo = p_Ugyfel
) LOOP
visszahoz(p_Ugyfel, k.konyv);
END LOOP;
END visszahoz;
/*
Kilistázza a lejárt kölcsönzési rekordokat.
Ha a cur_lejart_kolcsonzesek kurzor nyitva van,
az eljárás nem használható. Ilyenkor
kurzor_hasznalatban kivétel váltódik ki.
*/
PROCEDURE lejart_konyvek
IS
BEGIN
IF cur_lejart_kolcsonzesek%ISOPEN THEN
raise_kurzor_hasznalatban;
END IF;
DBMS_OUTPUT.PUT_LINE(str('Ügyfél', 40) || ' ' || str('Könyv', 50)
|| ' ' || str('dátum', 18) || 'napok');
DBMS_OUTPUT.PUT_LINE(RPAD('', 120, '-'));
FOR k IN cur_lejart_kolcsonzesek LOOP
DBMS_OUTPUT.PUT_LINE(
str(k.kolcsonzo || ' ' || az_ugyfel(k.kolcsonzo).nev, 40)
|| ' ' || str(k.konyv || ' ' || a_konyv(k.konyv).cim, 50)
|| ' ' || TO_CHAR(k.datum, 'YYYY-MON-DD HH24:MI')
|| ' ' || k.napok
);
END LOOP;
END lejart_konyvek;
/*
Kilistázza a lejárt kölcsönzési rekordok közül
```

```
azokat, amelyek ma jártak le.
Ha a cur_lejart_kolcsonzesek kurzor nyitva van,
az eljárás nem használható. Ilyenkor
kurzor_hasznalatban kivétel váltódik ki.
*/
PROCEDURE mai_lejart_konyvek
IS
BEGIN
IF cur_lejart_kolcsonzesek%ISOPEN THEN
raise_kurzor_hasznalatban;
END IF;
DBMS_OUTPUT.PUT_LINE(str('Ügyfél', 40) || ' ' || str('Könyv', 50)
|| ' ' || str('dátum', 18) || 'napok');
DBMS_OUTPUT.PUT_LINE(RPAD('', 120, '-'));
FOR k IN cur_lejart_kolcsonzesek LOOP
IF k.napok = 1 THEN
DBMS_OUTPUT.PUT_LINE(
str(k.kolcsonzo || ' ' || az_ugyfel(k.kolcsonzo).nev, 40)
|| ' ' || str(k.konyv || ' ' || a_konyv(k.konyv).cim, 50)
|| ' ' || TO_CHAR(k.datum, 'YYYY-MON-DD HH24:MI')
);
END IF;
END LOOP;
END mai_lejart_konyvek;
/*
Megadja egy kölcsönzött könyv hátralevő kölcsönzési idejét.
hibas_argumentum kivételt vált ki, ha nincs ilyen kölcsönzés.
*/
FUNCTION hatralevo_napok(
p_Ugyfel ugyfel.id%TYPE,
p_Konyv konyv.id%TYPE
) RETURN INTEGER
IS
v_Datum kolcsonzes.datum%TYPE;
v_Most DATE;
v_Hosszabbitva kolcsonzes.hosszabbitva%TYPE;
```

```
BEGIN
SELECT datum, hosszabbitva
INTO v_Datum, v_Hosszabbitva
FROM kolcsonzes
WHERE kolcsonzo = p_Ugyfel
AND konyv = p_Konyv;
/* Levágjuk a dátumokból az óra részt. */
v_Datum := TRUNC(v_Datum, 'DD');
v_Most := TRUNC(SYSDATE, 'DD');
/* Visszaadjuk a különbséget. */
RETURN (v_Hosszabbitva + 1) * 30 + v_Datum - v_Most;
EXCEPTION
WHEN NO_DATA_FOUND THEN
raise_hibas_argumentum;
END hatralevo_napok;
/*********************************************************************/
/* A csomag használatára vonatkozó statisztikát kezelő alprogramok */
/*********************************************************************/
/*
Megadja a statisztikát a legutóbbi nullázás, ill.
az indulás óta.
*/
FUNCTION statisztika RETURN t_statisztika_rec
IS
BEGIN
RETURN v_Stat;
END statisztika;
/*
Kiírja a statisztikát a legutóbbi nullázás ill.
az indulás óta.
*/
PROCEDURE print_statisztika
IS
BEGIN
DBMS_OUTPUT.PUT_LINE('Statisztika a munkamenetben:');
DBMS_OUTPUT.NEW_LINE;
```

```
DBMS_OUTPUT.PUT_LINE(RPAD('Regisztrált új ügyfelek száma:', 40)
|| v_Stat.uj_ugyfelek);
DBMS_OUTPUT.PUT_LINE(RPAD('Regisztrált új könyvek száma:', 40)
|| v_Stat.uj_konyvek);
DBMS_OUTPUT.PUT_LINE(RPAD('Kölcsönzések száma:', 40)
|| v_Stat.kolcsonzesek);
DBMS_OUTPUT.PUT_LINE(RPAD('Visszahozott könyvek száma:', 40)
|| v_Stat.visszahozott_konyvek);
DBMS_OUTPUT.PUT_LINE(RPAD('Utolsó módosítás: ', 40)
|| TO_CHAR(v_Stat.utolso_modositas, 'YYYY-MON-DD HH24:MI:SS'));
DBMS_OUTPUT.PUT_LINE(RPAD('Statisztika kezdete: ', 40)
|| TO_CHAR(v_Stat.utolso_nullazas, 'YYYY-MON-DD HH24:MI:SS'));
END print_statisztika;
/*
Nullázza a statiszikát.
*/
PROCEDURE statisztika_nullaz
IS
v_Stat2 t_statisztika_rec;
BEGIN
v_Stat := v_Stat2;
END statisztika_nullaz;
/*********************************/
/* A csomag inicializáló része. */
/*********************************/
BEGIN
statisztika_nullaz;
END konyvtar_csomag;
/
show errors
```

11. fejezet - PL/SQL programok fejlesztése és futtatása

A PL/SQL kód különböző környezetekben futtatható. A PL/SQL motor természetes módon helyezkedik el a szerveren és az Oracle minden esetben elhelyez PL/SQL motort a szerveren. A kliensalkalmazás SQL és PL/SQL utasításokat egyaránt el tud küldeni feldolgozásra a szerverhez. Az Oracle kliensoldali beépített fejlesztői és futtatói környezetét az SQL*Plus, illetve ennek webböngészőben használható változata, az *i*SQL*Plus adja.

Ugyanakkor PL/SQL motor futtatható a kliensen is. Az Oracle fejlesztőeszközei közül például a Forms és a Reports saját, beépített PL/SQL motorral rendelkezik. Mivel ezek a fejlesztői eszközök a kliensen futnak, így a motor is ott fut. A kliensoldali motor különbözik a szerveroldalitól. Egy Forms alkalmazás például tartalmazhat triggereket és alprogramokat. Ezek a kliensen futnak le, és csak az általuk tartalmazott SQL utasítások kerülnek át feldolgozásra a szerverhez, illetve ott a szerveren tárolt alprogramok futnak.

Az Oracle előfordítói (például Pro*C/C++ vagy Pro*COBOL) segítségével harmadik generációs nyelveken írt alkalmazásokba ágyazhatunk be PL/SQL kódot. Ezek az alkalmazások természetesen nem tartalmaznak PL/SQL motort, a PL/SQL utasításokat a szerveroldali motor dolgozza fel. Maguk az előfordítók viszont tartalmaznak egy olyan PL/SQL motort, amely a beágyazott PL/SQL utasítások szintaktikáját és szemantikáját ellenőrzi az előfordítás közben. Ez a motor azonban soha nem futtat PL/SQL utasításokat.

Maga az Oracle és egyéb szoftverfejlesztő cégek is kidolgoztak komplett PL/SQL fejlesztő és futtató környezeteket. Ezek integrált módon tartalmazzák mindazon szolgáltatásokat, amelyek egy PL/SQL alkalmazás hatékony és gyors megírásához és teszteléséhez szükségesek. Az Oracle ingyenes integrált adatbázis-fejlesztői környezete az Oracle SQL Developer. A termék nem része az adatbázis-kezelőnek, külön kell telepíteni, *lásd* [27], [29].

A továbbiakban csak az SQL*Plus, *i*SQL*Plus és az Oracle SQL Developer lehetőségeit tárgyaljuk röviden, az egyéb eszközök ismertetése túlmutat e könyv keretein.

1. SQL*Plus

Az SQL*Plus a legegyszerűbb PL/SQL fejlesztőeszköz. Lehetővé teszi SQL utasítások és név nélküli PL/SQL blokkok interaktív bevitelét. Ezek az utasítások a szerverhez kerülnek végrehajtásra, az eredmény pedig megjelenik a képernyőn. Az SQL*Plus karakteres felületet biztosít.

Ha SQL*Plusban SQL utasítást akarunk futtatni, akkor az utasítást pontosvesszővel (vagy egy / karakterrel) kell lezárnunk. Ekkor a pontosvessző nem része az utasításnak, hanem azt jelzi, hogy az utasítás teljes, elküldhető feldolgozásra. Ugyanakkor a PL/SQL esetén a pontosvessző része a szintaktikának, a kód lényeges eleme. Az SQL*Plus a DECLARE vagy BEGIN alapszavakat tekinti egy blokk kezdetének, a végét pedig egy / jelzi. Ez az SQL*Plus RUN utasítását rövidíti. Ennek hatására küldi el a teljes blokkot feldolgozásra.

A könyv egyéb fejezeteinek példáiban sokszor alkalmazzuk ezt az eszközt, ezért itt külön példát most nem adunk.

Az SQL*Plus az interaktív használat közben a begépelt információt az SQL-pufferben tárolja. Ez a puffer karakteresen szerkeszthető. A puffer tartalma elmenthető egy állományba a SAVE, onnan visszatölthető a GET paranccsal. Az állomány végrehajtható a START paranccsal. Ezen parancsok segítségével tehát egy PL/SQL blokk tárolható, módosítható, ismételten végrehajtható. Részleteket illetően *lásd* [8], [22].

Tárolt vagy csomagban elhelyezett eljárás SQL*Plusból meghívható az

```
EXEC[UTE] eljáráshívás;
```

SQL parancs segítségével. Ekkor az SQL*Plus a következő blokkot küldi el feldolgozásra:

```
BEGIN

eljáráshívás;
```

```
END;
```

2. *i*SQL*Plus

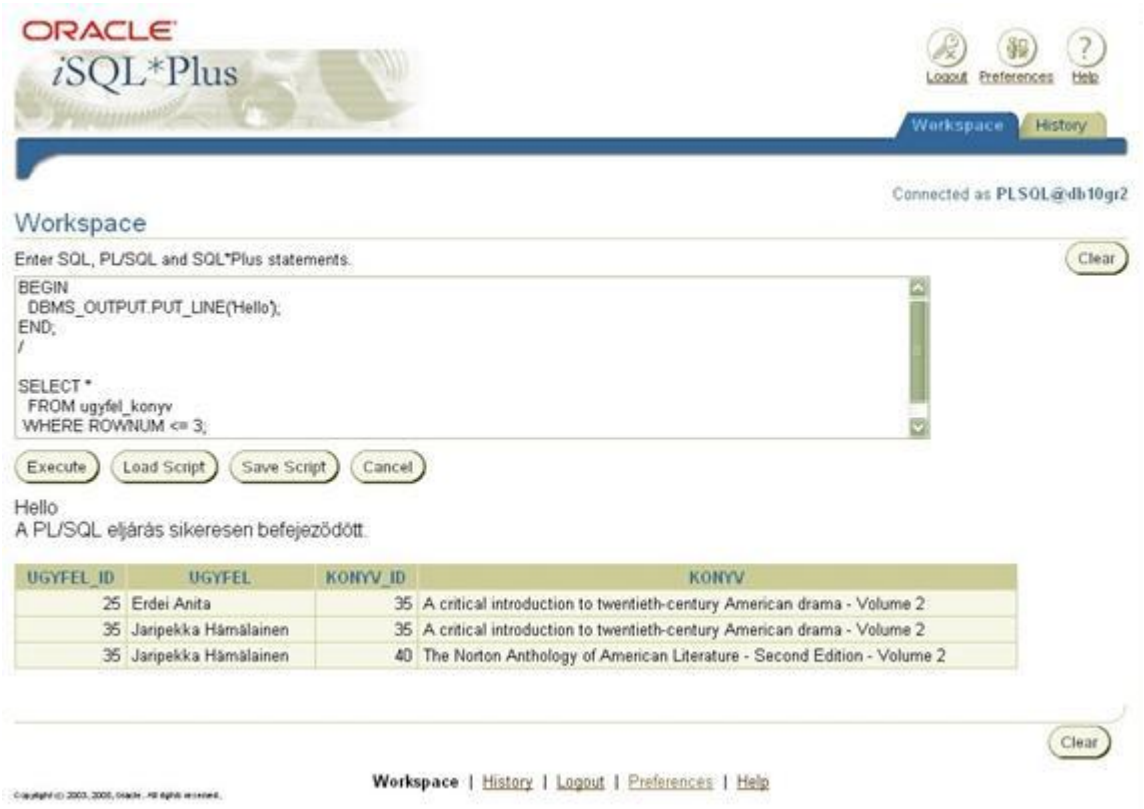
Az *i*SQL*Plus az SQL*Plus böngészőben használható változata. Az *i*SQL*Plus szolgáltatás egy szerveren fut (tipikusan ugyanazon a gépen, mint az adatbázis-kezelő), ahova az adatbázis-kezelő telepítése során automatikusan kerül fel. Használatához a kliensgépen nincs szükség Oracle-szoftverek telepítésére. Az *i*SQL*Plus szolgáltatás szerveroldali elindításáról, konfigurálásáról és kezeléséről [22] tartalmaz részletes dokumentációt.

Az *i*SQL*Plus használatához szükség van a szolgáltatás címére, URL-jére. Ezt kérjük a szoftver telepítőjétől, adminisztrátorától. A cím alapértelmezett beállításait megtaláljuk a dokumentációban, a testre szabott, egyéni beállításokról a szerveren a szoftver telepítési könyvtárában a $ORACLE_HOME/install/portlist.ini állományban találunk információt. A bejelentkezésnél vegyük figyelembe, hogy a beírt adatbázis-leírót a szerveroldalon fogja az alkalmazás feloldani.

Megjelenésében az *i*SQL*Plus a böngésző nyelvi beállításait veszi alapul, több nyelvet is támogat, a **magyart viszont nem**. A bejelentkezés után egy szövegablakba írhatjuk az utasításainkat, egy futtatás során többet is. Az utasítások elhatárolhatók az SQL*Plushoz hasonlóan ; vagy / jelekkel. Az utolsó utasítást nem kötelező lezárni. Lehetőség van szkriptek mentésére és betöltésére és a kimenet állományba történő mentésére. Meg tudjuk tekinteni és vissza tudjuk hozni a munkameneten belül korábban kiadott utasításokat, testre szabhatjuk a szövegablak méretét, a kimenet lapokra tördelését és egyéb formázási beállításait is.

Ez az eszköz kevés ismerettel is könnyen, gyorsan, jól használható, kliens oldali telepítést nem igényel. Szkriptek parancssorból történő futtatására, ütemezésére viszont nem alkalmas. Ezért a hagyományos SQL*Plus alapos ismerete elengedhetetlen adatbázisadminisztrátorok körében és azoknak a fejlesztőknek is ajánlott, akiknek a legegyszerűbb környezetből, parancssorból is kapcsolódniuk kell Oracle adatbázis-kezelőhöz. Az SQL*Plus lehetőségeinek és parancsainak ismerete az *i*SQL*Plus haladó használatában is előnyt jelent.

11.1. ábra - Az iSQL*Plus munkaterülete



3. Oracle SQL Developer

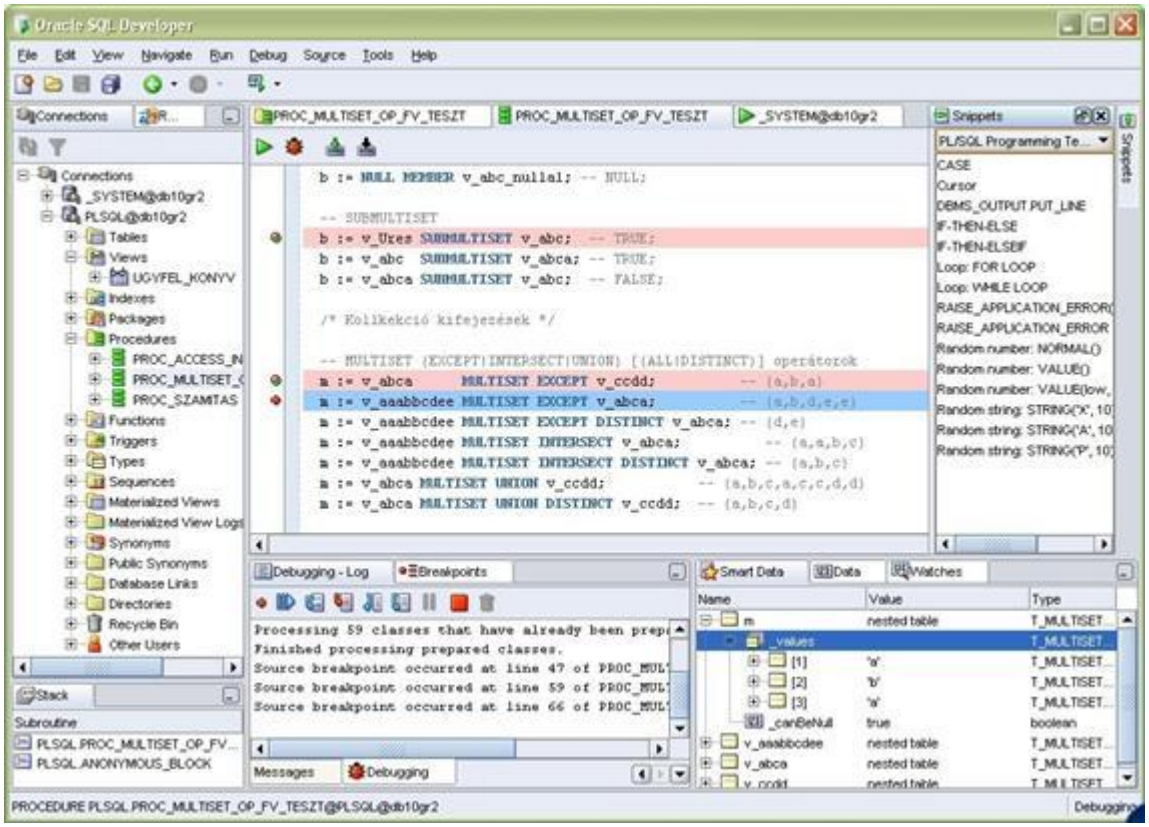
Az Oracle SQL Developer, korábbi nevén Project Raptor, egy Java nyelven írt teljes körű, ingyenes, grafikus integrált adatbázis-fejlesztői környezet.

Az SQL Developerben kapcsolódni tudunk egy vagy több adatbázispéldányhoz, akár egy időben is. A kapcsolathoz nincs szükség más Oracle-szoftver kliensoldali telepítésére, ha azonban ilyen telepítve van, akkor annak beállításait érzékeli és használni is tudja (például TNS beállítások, OCI elérés).

A fejlesztői környezet főbb funkciói az alábbiak:

- Az adatbázis tartalmának grafikus, hierarchikus böngészése, kezelése.
- Adatbázisban tárolt objektumok (táblák, típusok, tárolt programegységek stb.) létrehozása, kezelése.
- Szintaxis kiemelése.
- SQL és PL/SQL futtatása az SQL*Plus Worksheet panelben. Kimenetek megjelenítése. Táblák tartalmának táblázatos szerkesztése.
- SQL és PL/SQL szkriptek szerkesztése, fordítási hibák és figyelmeztetések jelzése, a kód automatikus formázása.
- PL/SQL nyomkövető (debugger). Töréspontok, kifejezések értékeinek futás közbeni figyelése, összetett típusok támogatása stb.
- Adatok és objektumok exportálása.
- Előre megírt kódminták használata.
- Előre megírt és saját riportok használata.
- Beépülő modulok támogatása.
- Automatikus frissítési lehetőség.

11.2. ábra - Az Oracle SQL Developer képernyője



Ez az eszköz is könnyen, gyorsan használható, könnyen telepíthető. Használatát javasoljuk.

A termék elérhető az internetről [27], [29]. Ugyanitt a további funkciók leírását, a használattal kapcsolatos információkat, cikkeket, fórumokat és beépülő modulokat is találunk.

Az Oracle egy másik ingyenes fejlesztői környezete, a JDeveloper – amely elsősorban Java alkalmazások fejlesztőeszköze – is tartalmaz hasonló funkcionalitásokat.

4. A DBMS_OUTPUT csomag

A PL/SQL nem tartalmaz I/O utasításokat. A tesztelésnél viszont a program különböző fázisaiban a képernyőn megjelenő információk sokat segíthetnek a belövéshez. A DBMS_OUTPUT csomag eszközöknek egy olyan gyűjteménye, amelynek segítségével a tárolt és a csomagbeli alprogram vagy trigger is üzenetet tud elhelyezni egy belső pufferbe, amelyet azután egy másik alprogram vagy trigger olvashat, vagy a puffer tartalma a képernyőre kerülhet. Ez utóbbi segít a nyomkövetésben.

Az üzenetek az alprogram vagy trigger működése közben kerülnek a pufferbe, de csak a befejeződés után olvashatók. Ha ezeket nem olvassa egy másik alprogram vagy nem kerülnek képernyőre, akkor elvesznek.

A csomag a következő típust tartalmazza:

```
TYPE CHARARR TABLE OF VARCHAR2(32767)

INDEX BY BINARY_INTEGER;

Ez szolgál az üzenetek kezelésére.

A csomag eljárásai a következők:

ENABLE(m IN INTEGER DEFAULT 20000)
```

Engedélyezi a csomag további eljárásainak használatát és beállítja az üzenetpuffer méretét *m*-re (2000 <= *m* <= 1000000).

```
DISABLE
```

Paraméter nélküli, letiltja a csomag további eljárásainak használatát és törli az üzenetpuffer tartalmát.

`PUT(a IN VARCHAR2) és PUT_LINE(a IN VARCHAR2)`

Az *a* üzenetet elhelyezik az üzenetpufferben. A PUT_LINE ezután még odaír egy sorvége jelet is.

`NEW_LINE`

Paraméter nélküli, az üzenetpufferben elhelyez egy sorvége jelet.

`GET_LINE(l OUT VARCHAR2, s OUT INTEGER)`

Az üzenetpufferből az *l*-be olvas egy sort (sorvége jelig). Az *s* értéke 0, ha az olvasás

sikeres volt, egyébként 1. Törli az üzenetpuffer tartalmát.

`GET_LINES(l OUT CHARARR, n IN OUT INTEGER)`

n számú sort olvas be az üzenetpufferből *l*-be. Az olvasás után *n* értéke a ténylegesen olvasott üzenetsorok darabszáma. Törli az üzenetpuffer tartalmát.

Az SQL*Plus SET parancsának segítségével szabályozhatjuk az üzenetpuffer tartalmának automatikus képernyőre írását.

Ha kiadjuk a következő parancsot, akkor az automatikus ENABLE hívást eredményez *n* puffermérettel vagy UNLIMITED esetén a maximális puffermérettel. A SIZE elhagyása esetén UNLIMITED az alapértelmezett.

`SET SERVEROUTPUT ON [SIZE {n | UNLIMITED}]`

Ezután pedig minden név nélküli PL/SQL blokk végrehajtása után a rendszer végrehajt egy GET_LINES hívást és az üzenetsorokat a képernyőre írja. Tehát a futás *közben* az üzenetpufferbe írt információk a futás *után* láthatók a képernyőn.

A könyv egyéb fejezeteinek példáiban sokszor alkalmazzuk ezt az eszközrendszert, ezért itt külön példát most nem adunk.

12. fejezet - Kollekciók

Egy *kollekció* azonos típusú adatelemek egy rendezett együttese. A kollekción belül minden elemnek egy egyedi *indexe* van, amely egyértelműen meghatározza az elem kollekción belüli helyét. A PL/SQL három kollekciótípust kezel, ezek az *asszociatív tömb*, a *beágyazott tábla* és a *dinamikus* (vagy *változó méretű*) *tömb.*

A kollekciók a harmadik generációs programozási nyelvek tömb fogalmával analóg eszközök a PL/SQL-ben. A kollekciók mindig egydimenziósak és indexeik egész típusúak, kivéve az asszociatív tömböt, amely indexelhető sztringgel is. A többdimenziós tömböket olyan kollekciókkal tudjuk kezelni, melyek elemei maguk is kollekciók.

A beágyazott táblák és a dinamikus tömbök elemei lehetnek objektumtípus példányai és ők lehetnek objektumtípus attribútumai. A kollekciók átadhatók paraméterként, így segítségükkel adatbázis tábláinak oszlopai mozgathatók az alkalmazások és az adatbázis között.

Az asszociatív tömb lényegében egy hash tábla, indexei (amelyek itt a kulcs szerepét játsszák) tetszőleges egészek vagy sztringek lehetnek. A beágyazott tábla és dinamikus tömb indexeinek alsó határa 1. Az asszociatív tömb esetén az indexeknek sem az alsó, sem a felső határa, beágyazott táblánál a felső határ nem rögzített.

Mindhárom kollekciótípus deklarálható PL/SQL blokk, alprogram és csomag deklarációs részében, a beágyazott tábla és dinamikus tömb típusok létrehozhatók adatbázis-objektumokként is a CREATE TYPE utasítással.

Az asszociatív tömb csak a PL/SQL programokban használható, a másik két kollekciótípus viszont lehet adatbázistábla oszlopának típusa is. Ilyen beágyazott tábla az *ugyfel* tábla *konyvek* oszlopa, és ilyen dinamikus tömb a *konyv* tábla *szerzo* oszlopa.

Egy dinamikus tömb típusának deklarációjánál meg kell adni egy maximális méretet, ami rögzíti az indexek felső határát. Az adott típusú dinamikus tömb ezután változó számú elemet tartalmazhat, a nulla darabtól a megadott maximális értékig. Egy dinamikus tömbben az elemek mindig *folytonosan*, a kisebb indexű „helyeken” helyezkednek el. Egy dinamikus tömb bővíthető (a maximális méretig) új elemmel a végén, de az egyszer bevitt lemek nem törölhetők, csak cserélhetők. Dinamikus tömb típusú oszlop értékei 4K méret alatt a táblában, efölött ugyanazon táblaterület egy külön helyén, egy *tárolótáblában* tárolódnak.

A beágyazott táblánál az elemek *szétszórtan* helyezkednek el, az elemek között lehetnek „lyukak”. Új elemeket tetszőleges indexű helyre bevihetünk és bármelyik elem törölhető (a helyén egy „lyuk” keletkezik).

A beágyazott tábla típusú oszlop értékei soha nem a táblában, hanem mindig a tárolótáblában helyezkednek el. A tárolótábla saját táblaterülettel és egyéb fizikai paraméterekkel rendelkezhet.

Explicit módon nem inicializált kollekciók esetén a beágyazott tábla és a dinamikus tömb automatikusan NULL kezdőértéket kap (tehát maga a kollekció, és nem az elemei), az asszociatív tömb viszont nem (csak egyszerűen nincs egyetlen eleme sem).

Az asszociatív tömb és a gazdanyelvek tömbje között a PL/SQL automatikus konverziót biztosít.

1. Kollekciók létrehozása

A kollekciótípusokat is felhasználói típusként kell deklarálni. Tekintsük át az egyes kollekciótípusok

deklarációjának formáját.

Asszociatív tömb:

```
TYPE név IS TABLE OF elemtípus [NOT NULL]

INDEX BY indextípus;
```

Az *indextípus* egy PLS_INTEGER, BINARY_INTEGER vagy VARCHAR2 típus vagy altípus

specifikáció.

Beágyazott tábla:

```
TYPE név IS TABLE OF elemtípus [NOT NULL];
```

Dinamikus tömb:

```
TYPE név IS {VARRAY | VARYING ARRAY} (max_méret)
OF elemtípus [NOT NULL];
```

A *név* a kollekciótípus neve, amelyet ezen deklaráció után tetszőleges kollekció deklarációjában használhatunk típusként az alábbi módon:

```
kollekciónév kollekciótípus_név;
```

A *max_méret* pozitív egész literál, a dinamikus tömb indexeinek felső határát adja meg. Az *elemtípus* egy PL/SQL adattípus, amely nem lehet REF CURSOR.

CREATE TYPE utasítással létrehozott beágyazott tábla esetén tiltottak még az alábbi típusok is:

BINARY_INTEGER	LONG RAW	POSITIVEN
PLS_INTEGER	NATURAL	SIGNTYPE
BOOLEAN	NATURALN	STRING
LONG	POSITIVE	

Egy kollekció elemére a következő módon hivatkozhatunk:

```
kollekciónév(index)
```

Ha kollekció típusú visszatérési értékkel rendelkező függvényt használunk, akkor a hivatkozás alakja:

```
függvénynév(aktuális_paraméter_lista)(index)
```

A PL/SQL lehetővé teszi olyan kollekciók használatát is, amelyeknek elemei kollekció típusúak.

1. példa (Kollekciótípusok és ilyen típusú változók deklarációja)

```
DECLARE
/* Kollekciót nem tartalmazó kollekciók */
-- asszociatív tömb, BINARY_INTEGER index
TYPE t_kolcsonzesek_at_binint IS
TABLE OF kolcsonzes%ROWTYPE
INDEX BY BINARY_INTEGER;
-- asszociatív tömb, PLS_INTEGER index,
-- szerkezete egyezik az előző típussal, de nem azonos típus
```

```
TYPE t_kolcsonzesek_at_plsint IS
TABLE OF kolcsonzes%ROWTYPE
INDEX BY PLS_INTEGER;
-- asszociatív tömb, VARCHAR2 index, nem rekord típusú elemek
TYPE t_konyv_idk_at_vc2 IS
TABLE OF konyv.id%TYPE
INDEX BY konyv.isbn%TYPE; -- VARCHAR2(30)
-- beágyazott tábla
TYPE t_kolcsonzesek_bt IS
TABLE OF kolcsonzes%ROWTYPE;
-- dinamikus tömb, objektumot tartalmaz
-- megj.: Hasonlítsa össze a T_Konyvek adatbázistípussal
TYPE t_konyvlista IS
VARRAY(10) OF T_Tetel;
-- Nem rekordot tartalmazó kollekció
TYPE t_vektor IS TABLE OF NUMBER
INDEX BY BINARY_INTEGER;
/* Kollekciók kollekciói */
-- mátrix, mindkét dimenziójában szétszórt!
TYPE t_matrix IS TABLE OF t_vektor
INDEX BY BINARY_INTEGER;
-- a konyv tábla egyik oszlopa kollekció
TYPE t_konyvek_bt IS TABLE OF konyv%ROWTYPE;
/* Változódeklarációk és inicializációk */
-- segédváltozók
v_Tetel T_Tetel;
v_Szam NUMBER;
v_Szerzo VARCHAR2(50);
v_ISBN konyv.isbn%TYPE;
-- asszociatív tömböket nem lehet inicializálni
v_Kolcsonzesek_at_binint1 t_kolcsonzesek_at_binint;
v_Kolcsonzesek_at_binint2 t_kolcsonzesek_at_binint;
v_Kolcsonzesek_at_plsint t_kolcsonzesek_at_plsint;
v_Konyv_id_by_ISBN t_konyv_idk_at_vc2;
v_Vektor t_vektor;
v_Matrix t_matrix;
```

```
-- Ha nincs inicializáció, akkor a beágyazott tábla és a
-- dinamikus tömb NULL kezdőértéket kap
v_Konyvek_N t_konyvek_bt;
v_Konyvlista_N t_konyvlista;
-- beágyazott táblát és dinamikus tömböt lehet inicializálni
-- Megj.: v_Konyvlista_I 2 elemű lesz az inicializálás után
v_Konyvek_I t_konyvek_bt := t_konyvek_bt();
v_Konyvlista_I t_konyvlista := t_konyvlista(T_Tetel(10, SYSDATE),
T_Tetel(15, SYSDATE));
-- Lehet adatbázisbeli kollekciót is használni
v_Konyvek_ab T_Konyvek := T_Konyvek();
-- Ha a tábla elemei skalár típusúak, akkor az inicializálás:
v_Szerzok_ab T_Szerzok := T_Szerzok('P. Howard', NULL, 'Rejtő Jenő');
/*
Rekordelemű (nem objektum és nem skalár)
kollekció is inicializálható megfelelő elemekkel.
Megj.: rekord változó NEM lehet NULL, ha
inicializációnál NULL-t adunk meg, akkor az adott elem minden mezője
NULL lesz.
*/
-- Megfelelő rekordtípusú elemek
v_Kolcsonzes1 kolcsonzes%ROWTYPE;
FUNCTION a_kolcsonzes(
p_Kolcsonzo kolcsonzes.kolcsonzo%TYPE,
p_Konyv kolcsonzes.konyv%TYPE
) RETURN kolcsonzes%ROWTYPE;
-- Az inicializáció
v_Kolcsonzesek_bt t_kolcsonzesek_bt
:= t_kolcsonzesek_bt(v_Kolcsonzes1, a_kolcsonzes(15, 20));
-- A függvény implementációja
FUNCTION a_kolcsonzes(
p_Kolcsonzo kolcsonzes.kolcsonzo%TYPE,
p_Konyv kolcsonzes.konyv%TYPE
) RETURN kolcsonzes%ROWTYPE IS
v_Kolcsonzes kolcsonzes%ROWTYPE;
BEGIN
```

```
SELECT * INTO v_Kolcsonzes
FROM kolcsonzes
WHERE kolcsonzo = p_Kolcsonzo
AND konyv = p_Konyv;
RETURN v_Kolcsonzes;
END a_kolcsonzes;
/* Kollekciót visszaadó függvény */
FUNCTION fv_kol(
p_Null_legyen BOOLEAN
) RETURN t_konyvlista IS
BEGIN
RETURN CASE
WHEN p_Null_legyen THEN NULL
ELSE v_Konyvlista_I
END;
END fv_kol;
⋮
```

2. példa (Kollekcióelemre történő hivatkozások (az előző blokk folytatása))

```
⋮
BEGIN
/* hivatkozás kollekció elemére: */
DBMS_OUTPUT.PUT_LINE(v_Kolcsonzesek_bt(2).kolcsonzo);
DBMS_OUTPUT.PUT_LINE(fv_kol(FALSE)(1).konyv_id);
-- értékadás lekérdezéssel
SELECT *
INTO v_Kolcsonzesek_at_binint1(100)
FROM kolcsonzes
WHERE ROWNUM <=1;
-- Értékadás a teljes kollekció másolásával.
-- Ez költséges művelet, OUT és IN OUT módú paramétereknél
-- fontoljuk meg a NOCOPY használatát.
v_Kolcsonzesek_at_binint2 := v_Kolcsonzesek_at_binint1;
-- A következő értékadás fordítási hibát okozna, mert
-- a szerkezeti egyezőség nem elég az értékadáshoz:
--
```

```
-- v_Kolcsonzesek_at_plsint := v_Kolcsonzesek_at_binint1;

--

-- a v_ISBN segédváltozó inicializálása

SELECT isbn

INTO v_ISBN

FROM konyv

WHERE cim like 'A teljesség felé'

;

-- értékadás karakteres indexű asszociatív tömb egy elemének

SELECT id

INTO v_Konyv_id_by_ISBN(v_ISBN)

FROM konyv

WHERE isbn = v_ISBN

;

-- elem hivatkozása

DBMS_OUTPUT.PUT_LINE('ISBN: "' || v_ISBN

|| '", id: ' || v_Konyv_id_by_ISBN(v_ISBN));

-- A v_Matrix elemeit sakktáblaszerűen feltöltjük.

<<blokk1>>

DECLARE

k PLS_INTEGER;

BEGIN

k := 1;

FOR i IN 1..8 LOOP

FOR j IN 1..4 LOOP

v_Matrix(i)(j*2 - i MOD 2) := k;

k := k + 1;

END LOOP;

END LOOP;

END blokk1;

:
```

2. Kollekciók kezelése

BINARY_INTEGER típusú indexekkel rendelkező asszociatív tömb indexeinek maximális tartományát ezen típus tartománya határozza meg: –231..231.

Beágyazott tábla és dinamikus tömb esetén az indexek lehetséges maximális felső határa 231.

Asszociatív tömbök esetén egy *i* indexű elemnek történő értékadás létrehozza az adott elemet, ha eddig még nem létezett, illetve felülírja annak értékét, ha az már létezik. Beágyazott tábla és dinamikus tömb esetén biztosítani kell, hogy az adott indexű elem létezzen az értékadás előtt. Az *i* indexű elemre való hivatkozás csak a létrehozása után lehetséges, különben a NO_DATA_FOUND kivétel váltódik ki.

1. példa (Az előző példa folytatása)

```
⋮

-- inicializálatlan elemre hivatkozás

<<blokk2>>

BEGIN

v_Szam := v_Matrix(20)(20);

EXCEPTION

WHEN NO_DATA_FOUND THEN

DBMS_OUTPUT.PUT_LINE('Kivétel blokk2-ben: ' || SQLERRM);

END blokk2;

⋮
```

A beágyazott tábla és a dinamikus tömb speciális objektumtípusok (az objektumtípusok részletes tárgyalását *lásd* a 14. fejezetben). Amikor egy ilyen kollekciót deklarálunk, tulajdonképpen egy referencia típusú változó jön létre, amelynek automatikus kezdőértéke NULL. A kollekciót explicit módon inicializálni az objektumtípusnak megfelelően *példányosítással* (*lásd* 14. fejezet) lehet. A példányosításhoz az adott típus *konstruktorát* kell meghívni. A konstruktor egy rendszer által létrehozott függvény, amelynek neve megegyezik a típus nevével, paramétereinek száma tetszőleges, paraméterei típusának a megfelelő kollekciótípus elemeinek típusával kompatibilisnek kell lennie. A dinamikus tömb konstruktorának maximum annyi paraméter adható meg, amennyi a deklarált maximális elemszám.

A konstruktor meghívható paraméterek nélkül, ekkor egy üres kollekció jön létre. Javasoljuk, hogy a kollekció deklarálásakor hajtsunk végre explicit inicializálást, itt hívjuk meg a konstruktort.

Ha a beágyazott tábla és a dinamikus tömb kollekcióknál nem létező elemre hivatkozunk, akkor a SUBSCRIPT_BEYOND_COUNT kivétel, ha az index a LIMIT-nél nagyobb vagy nem pozitív szám, akkor SUBSCRIPT_OUTSIDE_LIMIT kivétel váltódik ki.

2. példa

```
⋮

v_Szerzo := v_Szerzok_ab(2); -- Létező elem, értéke NULL

<<blokk3>>

BEGIN

v_Tetel := v_Konyvlista_I(3); -- Nem létező elem

EXCEPTION

WHEN SUBSCRIPT_BEYOND_COUNT THEN

DBMS_OUTPUT.PUT_LINE('Kivétel blokk3-ban: ' || SQLERRM);

END blokk3;

<<blokk4>>

BEGIN
```

```
-- t_konyvlista dinamikus tömb maximális mérete 10

v_Tetel := v_Konyvlista_I(20); -- A maximális méreten túl hivatkozunk

EXCEPTION

WHEN SUBSCRIPT_OUTSIDE_LIMIT THEN

DBMS_OUTPUT.PUT_LINE('Kivétel blokk4-ben: ' || SQLERRM);

END blokk4;

⋮
```

Ha az index NULL, vagy nem konvertálható a kulcs típusára, akár a kulcs típusának korlátozása miatt, akkor a VALUE_ERROR kivétel következik be. Ha beágyazott tábla és dinamikus tömb kollekciók esetén nem inicializált kollekcióra hivatkozunk, a COLLECTION_IS_NULL kivétel következik be.

Beágyazott tábla és dinamikus tömb NULL értéke tesztelhető. Beágyazott táblák egyenlősége is vizsgálható akkor, ha azonos típusúak és az elemek is összehasonlíthatók egyenlőség szerint. Rendezettségre nézve még a beágyazott táblák sem hasonlíthatók össze.

3. példa

```
⋮

/* Kivétel NULL kollekció esetén */

<<blokk5>>

BEGIN

-- v_Konyvlista_N nem volt explicite inicializálva, értéke NULL.

v_Tetel := v_Konyvlista_N(1);

EXCEPTION

WHEN COLLECTION_IS_NULL THEN

DBMS_OUTPUT.PUT_LINE('Kivétel blokk5-ben: ' || SQLERRM);

END blokk5;

/* Nem asszociatív tömb kollekciók NULL tesztelése lehetséges */

IF v_Konyvlista_N IS NULL THEN

DBMS_OUTPUT.PUT_LINE('v_Konyvlista_N null volt.');

END IF;

IF v_Konyvlista_I IS NOT NULL THEN

DBMS_OUTPUT.PUT_LINE('v_Konyvlista_I nem volt null.');

END IF;

/* Csak beágyazott táblák egyenlősége vizsgálható, és csak akkor,

ha az elemeik is összehasonlíthatók. */

DECLARE

TYPE t_vektor_bt IS TABLE OF NUMBER;

v_Vektor_bt t_vektor_bt := t_vektor_bt(1,2,3);

BEGIN
```

```
IF v_Vektor_bt = v_Vektor_bt THEN

DBMS_OUTPUT.PUT_LINE('Egyenlőség...');

END IF;

END;

/* A rekordot tartalmazó beágyazott tábla és bármilyen elemű dinamikus

tömb vagy asszociatív tömb egyenlőségvizsgálata fordítási hibát

eredményezne. Például ez is:

IF v_Matrix(1) = v_Vektor THEN

DBMS_OUTPUT.PUT_LINE('Egyenlőség...');

END IF;

*/

END;

/
```

Az előző példák blokkjának futási eredménye:

```
15

10

ISBN: "ISBN 963 8453 09 5", id: 10

Kivétel blokk2-ben: ORA-01403: Nem talált adatot

Kivétel blokk3-ban: ORA-06533: Számlálón kívüli index érték

Kivétel blokk4-ben: ORA-06532: Határon kívüli index

Kivétel blokk5-ben: ORA-06531: Inicializálatlan gyűjtőre való hivatkozás

v_Konyvlista_N null volt.

v_Konyvlista_I nem volt null.

Egyenlőség ...

A PL/SQL eljárás sikeresen befejeződött.
```

Csak beágyazott tábláknál alkalmazhatók az SQL nyelv kollekciót kezelő operátorai és függvényei.

Logikai operátorok: IS [NOT] A SET, IS [NOT] EMPTY, MEMBER, SUBMULTISET.

Kollekció operátorok: MULTISET EXCEPT [{ALL|DISTINCT}], MULTISET INTERSECT [{ALL|DISTINCT}], MULTISET UNION.

PL/SQL-ben is használható kollekció függvények: CARDINALITY, SET.

A COLLECT, POWERMULTISET és POWERMULTISET_BY_CARDINALITY kollekciófüggvények, a DECODE-hoz hasonlóan, PL/SQL-ben közvetlenül nem használhatók.

A következő példa ezek használatát szemlélteti PL/SQL-ben. Részletes leírásukat *lásd* [8].

4. példa

```
/*

Az egyes műveletek mögött láthatók az eredmények.
```

```
Ezek ellenőrzéséhez az SQL*Developer vagy más IDE debuggerét javasoljuk.
Ehhez szükséges a DEBUG CONNECT SESSION jogosultság
Megoldás lenne még az eredmények köztes kiíratása is.
*/
-- Néhány példához adatbázisban kell létrehozni típust
CREATE TYPE T_Multiset_ab IS
TABLE OF CHAR(1)
/
CREATE TYPE T_Multiset_multiset_ab IS
TABLE OF T_Multiset_ab;
/
ALTER SESSION SET plsql_debug=true;
CREATE OR REPLACE PROCEDURE proc_multiset_op_fv_teszt IS
TYPE t_multiset_plsql IS TABLE OF CHAR(1);
-- t_multiset változók - fő operandusok
v_Ures t_multiset_plsql := t_multiset_plsql();
v_abc t_multiset_plsql := t_multiset_plsql('a','b','c');
v_abca t_multiset_plsql := t_multiset_plsql('a','b','c','a');
v_abc_nullal t_multiset_plsql := t_multiset_plsql('a','b','c',NULL);
v_aaabbcdee t_multiset_plsql :=
t_multiset_plsql('a','a','a','b','b','c','d','e','e');
v_ccdd t_multiset_plsql := t_multiset_plsql('c','c','d','d');
v_abc_ab T_Multiset_ab := T_Multiset_ab('a','b','c');
-- eredménytárolók
b BOOLEAN;
m t_multiset_plsql;
i BINARY_INTEGER;
mm T_Multiset_multiset_ab;
-- segédeljárás: az adatbázisbeli típusú paraméter tartalmát
-- a lokális típusú paraméterbe másolja, mert a debugger
-- csak a lokális típusú változókba tud belenézni
PROCEDURE convert_to_plsql(
p_From T_Multiset_ab,
p_To IN OUT NOCOPY t_multiset_plsql
) IS
j BINARY_INTEGER;
```

```
BEGIN
p_To.DELETE;
p_To.EXTEND(p_From.COUNT);
FOR i IN 1..p_From.COUNT
LOOP
p_To(i) := p_From(i);
END LOOP;
END convert_to_plsql;
BEGIN
/* Logikai kifejezések */
-- IS [NOT] A SET
b := v_abc IS A SET; -- TRUE;
b := v_abca IS A SET; -- FALSE;
-- IS [NOT] EMPTY
b := v_abc IS NOT EMPTY; -- TRUE;
b := v_Ures IS NOT EMPTY; -- FALSE;
-- MEMBER
b := 'a' MEMBER v_abc; -- TRUE;
b := 'z' MEMBER v_abc; -- FALSE;
b := NULL MEMBER v_abc; -- NULL;
b := 'a' MEMBER v_abc_nullal; -- TRUE;
b := 'z' MEMBER v_abc_nullal; -- NULL;
b := NULL MEMBER v_abc_nullal; -- NULL;
-- SUBMULTISET
b := v_Ures SUBMULTISET v_abc; -- TRUE;
b := v_abc SUBMULTISET v_abca; -- TRUE;
b := v_abca SUBMULTISET v_abc; -- FALSE;
/* Kollekció kifejezések */
-- MULTISET {EXCEPT|INTERSECT|UNION} [{ALL|DISTINCT}] operátorok
m := v_abca MULTISET EXCEPT v_ccdd; -- {a,b,a}
m := v_aaabbcdee MULTISET EXCEPT v_abca; -- {a,b,d,e,e}
m := v_aaabbcdee MULTISET EXCEPT DISTINCT v_abca; -- {d,e}
m := v_aaabbcdee MULTISET INTERSECT v_abca; -- {a,a,b,c}
m := v_aaabbcdee MULTISET INTERSECT DISTINCT v_abca; -- {a,b,c}
m := v_abca MULTISET UNION v_ccdd; -- {a,b,c,a,c,c,d,d}
m := v_abca MULTISET UNION DISTINCT v_ccdd; -- {a,b,c,d}
```

```
/* PL/SQL-ben közvetlenül is alkalmazható kollekció függvények */
-- CARDINALITY, vesd össze a COUNT metódussal
i := CARDINALITY(v_abc); -- 3
i := v_abc.COUNT; -- 3
i := CARDINALITY(v_Ures); -- 0
i := v_Ures.COUNT; -- 0
-- SET
m := SET(v_abca); -- {a,b,c}
b := v_abc = SET(v_abca); -- TRUE;
/* PL/SQL-ben közvetlenül nem alkalmazható kollekció függvények */
-- COLLECT
FOR r IN (
SELECT grp, CAST(COLLECT(col) AS T_Multiset_ab) collected
FROM (
SELECT 1 grp, 'a' col FROM dual UNION ALL
SELECT 1 grp, 'b' col FROM dual UNION ALL
SELECT 2 grp, 'c' col FROM dual
)
GROUP BY grp
) LOOP
i := r.grp; -- 1 majd 2
-- debuggerrel tudjuk vizsgálni m értékét a konverzió után
convert_to_plsql(r.collected, m); -- {a,b} majd {c}
END LOOP;
-- POWERMULTISET
SELECT CAST(POWERMULTISET(v_abc_ab) AS T_Multiset_multiset_ab)
INTO mm
FROM dual;
-- mm : { {a}, {b}, {a,b}, {c}, {a,c}, {b,c}, {a,b,c} }
i := mm.COUNT; -- 7
convert_to_plsql(mm(1), m); -- {a}
convert_to_plsql(mm(2), m); -- {b}
convert_to_plsql(mm(3), m); -- {a,b}
convert_to_plsql(mm(4), m); -- {c}
convert_to_plsql(mm(5), m); -- {a,c}
convert_to_plsql(mm(6), m); -- {b,c}
```

```
convert_to_plsql(mm(7), m); -- {a,b,c}

-- POWERMULTISET_BY_CARDINALITY

SELECT CAST(POWERMULTISET_BY_CARDINALITY(v_abc_ab, 2)

AS T_Multiset_multiset_ab)

INTO mm

FROM dual;

-- mm : { {a,b}, {a,c}, {b,c} }

i := mm.COUNT; -- 3

convert_to_plsql(mm(1), m); -- {a,b}

convert_to_plsql(mm(2), m); -- {a,c}

convert_to_plsql(mm(3), m); -- {b,c}

END proc_multiset_op_fv_teszt;

/

show errors
```

A DML utasításokban használható a

```
TABLE (kollekciókifejezés)
```

utasításrész, ahol a *kollekciókifejezés* lehet alkérdés, oszlopnév, beépített függvény hívása. Minden esetben beágyazott tábla vagy dinamikus tömb típusú kollekciót kell szolgáltatnia. A TABLE segítségével az adott kollekció elemeihez mint egy tábla soraihoz férhetünk hozzá. Ha a kollekció elemei objektum típusúak, akkor a TABLE által szolgáltatott virtuális tábla oszlopainak a neve az objektumtípus attribútumainak nevével egyezik meg. Skalár típusú elemek kollekciójánál COLUMN_VALUE lesz az egyetlen oszlop neve.

Ugyancsak DML utasításokban alkalmazható a CAST függvény. Segítségével adatbázis vagy kollekció típusú értékeket tudunk másik adatbázis- vagy kollekciótípusra konvertálni. Alakja:

```
CAST({kifejezés|(alkérdés)|MULTISET(alkérdés)} AS típusnév)
```

A *típusnév* adja meg azt a típust, amelybe a konverzió történik. A *típusnév* lehet adatbázistípus vagy adatbázisban tárolt kollekciótípus neve. A *kifejezés* és az *alkérdés* határozza meg a konvertálandó értéket. Az egyedül álló *alkérdés* csak egyetlen értéket szolgáltathat. MULTISET esetén az *alkérdés* akárhány sort szolgáltathat, ezekből a típusnév által meghatározott kollekció elemei lesznek.

Kollekciók csak kompatibilis elem típusú kollekciókká konvertálhatók.

A 12.1. táblázat a CAST-tal megvalósítható konverziókat szemlélteti.

12.1. táblázat - Az adatbázistípusok konverziói

	CHAR, VARCHAR2	NUMBER	Dátum/ intervallum	RAW	ROWID, UROWID	NCHAR, NVARCHAR2
CHAR, VARCHAR2	X	X	X	X	X	
NUMBER	X	X				
Dátum/intervallum	X		X			

RAW	X			X		
ROWID, UROWID	X				X	
NCHAR, NVARCHAR2		X	X	X	X	X

Nézzünk néhány példát a TABLE és a CAST használatára.

5. példa

```
/*
CAST csak SQL-ben van. A típusnak adatbázistípusnak
kell lennie, viszont skalár is lehet.
*/
CREATE TYPE T_Rec IS OBJECT (
szam NUMBER,
nev VARCHAR2(100)
)
CREATE TYPE T_Dinamikus IS VARRAY(10) OF T_Rec
/
CREATE TYPE T_Beagyazott IS TABLE OF T_Rec
/
DECLARE
v_Dinamikus T_Dinamikus;
v_Beagyazott T_Beagyazott;
BEGIN
SELECT CAST(v_Beagyazott AS T_Dinamikus)
INTO v_Dinamikus
FROM dual;
SELECT CAST(MULTISET(SELECT id, cim FROM konyv ORDER BY UPPER(cim))
AS T_Beagyazott)
INTO v_Beagyazott
FROM dual;
END;
/
DROP TYPE T_Beagyazott;
DROP TYPE T_Dinamikus;
DROP TYPE T_Rec;
```

6. példa

```
SELECT * FROM ugyfel, TABLE(konyvek);

SELECT * FROM TABLE(SELECT konyvek FROM ugyfel WHERE id = 15);

CREATE OR REPLACE FUNCTION fv_Szerzok(p_Konyv konyv.id%TYPE)

RETURN T_Szerzok IS

v_Szerzo T_Szerzok;

BEGIN

SELECT szerzo

INTO v_Szerzo

FROM konyv

WHERE id = p_Konyv;

RETURN v_Szerzo;

EXCEPTION

WHEN NO_DATA_FOUND THEN

RETURN T_Szerzok();

END fv_Szerzok;

/

show errors

SELECT * FROM TABLE(fv_Szerzok(15));

SELECT * FROM TABLE(fv_Szerzok(150));

BEGIN

/* Ha skalár elemű kollekción végzünk el lekérdezést,

akkor az egyetlen oszlop neve COLUMN_VALUE lesz. */

FOR szerzo IN (

SELECT * FROM TABLE(fv_Szerzok(15))

) LOOP

DBMS_OUTPUT.PUT_LINE(szerzo.COLUMN_VALUE);

END LOOP:

END;

/
```

7. példa

```
DECLARE

/* Kilistázzuk a kölcsönzött könyvek azonosítóját és a kölcsönzött

példányok számát.

*/
```

```
v_Konyvek T_Konyvek;
v_Cim konyv.cim%TYPE;
/* Megadja egy könyv címét */
FUNCTION a_cim(p_Konyv konyv.id%TYPE)
RETURN konyv.cim%TYPE IS
v_Konyv konyv.cim%TYPE;
BEGIN
SELECT cim INTO v_Konyv FROM konyv WHERE id = p_Konyv;
RETURN v_Konyv;
END a_cim;
BEGIN
/* Lekérdezzük az összes kölcsönzést egy változóba */
SELECT CAST(MULTISET(SELECT konyv, datum FROM kolcsonzes) AS T_Konyvek)
INTO v_Konyvek
FROM dual;
/* Noha v_Konyvek T_Konyvek típusú, mivel változó, szükség van
a CAST operátorra, hogy az SQL utasításban használhassuk. */
FOR konyv IN (
SELECT konyv_id AS id, COUNT(1) AS peldany
FROM TABLE(CAST(v_Konyvek AS T_Konyvek))
GROUP BY konyv_id
ORDER BY peldany ASC
) LOOP
DBMS_OUTPUT.PUT_LINE(LPAD(konyv.id, 3) || ' '
|| LPAD(konyv.peldany, 2) || ' ' || a_cim(konyv.id));
END LOOP;
END;
/
/*
Eredmény:
5 1 A római jog története és institúciói
10 1 A teljesség felé
15 1 Piszkos Fred és a többiek
20 1 ECOOP 2001 - Object-Oriented Programming
40 1 The Norton Anthology of American Literature - Second Edition - Volume 2
25 1 Java - start!
```

```
30 2 SQL:1999 Understanding Relational Language Components

45 2 Matematikai zseblexikon

50 2 Matematikai Kézikönyv

35 2 A critical introduction to twentieth-century American drama - Volume 2

A PL/SQL eljárás sikeresen befejeződött.

*/
```

Kollekciókat visszaadó függvények hatékonyabban implementálhatók a PIPELINED opció használatával (*lásd* [19]).

3. Kollekciómetódusok

Az objektumtípusú kollekciók esetén a PL/SQL implicit módon metódusokat definiál a kezelésükhöz. Ezen metódusok egy része asszociatív tömb esetén is használható. Ezek a metódusok csak procedurális utasításokban hívhatók, SQL utasításokban nem.

A metódusok áttekintését szolgálja a 12.2. táblázat.

12.2. táblázat - Kollekciómetódusok

Metódus	Visszatérési típus	Tevékenység	Kollekció
EXISTS	BOOLEAN	Igaz értéket ad, ha az adott indexű elem létezik a kollekcióban.	Minden kollekció
COUNT	NUMBER	Visszaadja a kollekció elemeinek számát.	Minden kollekció
LIMIT	NUMBER	Visszaadja a kollekció maximális méretét.	Minden kollekció
FIRST	indextípus	Visszaadja a kollekció első elemének indexét.	Minden kollekció
LAST	indextípus	Visszaadja a kollekció utolsó elemének indexét.	Minden kollekció
NEXT	indextípus	Visszaadja egy megadott indexű elemet követő elem indexét.	Minden kollekció
PRIOR	indextípus	Visszaadja egy megadott indexű elemet megelőző elem indexét.	Minden kollekció
EXTEND	nincs	Bővíti a kollekciót.	Beágyazott tábla, dinamikus tömb
TRIM	nincs	Eltávolítja a kollekció utolsó elemeit.	Beágyazott tábla, dinamikus tömb
DELETE	nincs	A megadott elemeket törli a kollekcióból.	Asszociatív tömb, beágyazott tábla, dinamikus tömb esetén csak paraméter nélkül

Az egyes metódusokat a következőkben részletezzük.

EXISTS

Alakja

```
EXISTS(i)
```

ahol *i* egy az index típusának megfelelő kifejezés.

Igaz értéket ad, ha az *i* indexű elem létezik a kollekcióban, egyébként hamisat. Ha nem létező indexre hivatkozunk, az EXISTS hamis értékkel tér vissza és nem vált ki kivételt.

Az EXISTS az egyetlen metódus, amely nem inicializált beágyazott táblára és dinamikus tömbre is meghívható (ekkor hamis értékkel tér vissza), az összes többi metódus ekkor a COLLECTION_IS_NULL kivételt váltja ki.

Az EXISTS használatával elkerülhető, hogy nem létező elemre hivatkozzunk és ezzel kivételt váltsunk ki.

1. példa

```
DECLARE

v_Szerzok T_Szerzok;

i PLS_INTEGER;

BEGIN

SELECT szerzo

INTO v_Szerzok

FROM konyv

WHERE id = 15;

i := 1;

WHILE v_Szerzok.EXISTS(i) LOOP

DBMS_OUTPUT.PUT_LINE(v_Szerzok(i));

i := i+1;

END LOOP;

END;

/

/*

Eredmény:

P. Howard

Rejtő Jenő

A PL/SQL eljárás sikeresen befejeződött.

*/
```

2. példa

```
DECLARE

TYPE t_vektor IS TABLE OF NUMBER

INDEX BY BINARY_INTEGER;
```

```
v_Vektor t_vektor;
BEGIN
FOR i IN -2..2 LOOP
v_Vektor(i*2) := i;
END LOOP;
FOR i IN -5..5 LOOP
IF v_Vektor.EXISTS(i) THEN
DBMS_OUTPUT.PUT_LINE(LPAD(i,2) || ' '
|| LPAD(v_Vektor(i), 2));
END IF;
END LOOP;
END;
/
/*
Eredmény:
-4 -2
-2 -1
0 0
2 1
4 2
A PL/SQL eljárás sikeresen befejeződött.
*/
```

3. példa

```
DECLARE
TYPE t_tablazat IS TABLE OF NUMBER
INDEX BY VARCHAR2(10);
v_Tablazat t_tablazat;
v_Kulcs VARCHAR2(10);
BEGIN
FOR i IN 65..67 LOOP
v_Kulcs := CHR(i);
v_Tablazat(v_Kulcs) := i;
END LOOP;
DBMS_OUTPUT.PUT_LINE('Kulcs Elem');
DBMS_OUTPUT.PUT_LINE('----- ----');
```

```
FOR i IN 0..255 LOOP

v_Kulcs := CHR(i);

IF v_Tablazat.EXISTS(v_Kulcs) THEN

DBMS_OUTPUT.PUT_LINE(RPAD(v_Kulcs, 7) || v_Tablazat(v_Kulcs));

END IF;

END LOOP;

END;

/

/*

Eredmény:

Kulcs Elem

----- ----

A 65

B 66

C 67

A PL/SQL eljárás sikeresen befejeződött.

*/
```

COUNT

Paraméter nélküli függvény. Megadja a kollekció aktuális (nem törölt) elemeinek számát. Dinamikus tömb esetén visszatérési értéke azonos a LAST visszatérési értékével, a többi kollekciónál ez nem feltétlenül van így.

1. példa

```
DECLARE

v_Szerzok T_Szerzok;

BEGIN

SELECT szerzo

INTO v_Szerzok

FROM konyv

WHERE id = 15;

FOR i IN 1..v_Szerzok.COUNT LOOP

DBMS_OUTPUT.PUT_LINE(v_Szerzok(i));

END LOOP;

END;

/

/*

Eredmény:

P. Howard
```

```
Rejtő Jenő

A PL/SQL eljárás sikeresen befejeződött.

*/
```

2. példa

```
DECLARE

TYPE t_vektor IS TABLE OF NUMBER

INDEX BY BINARY_INTEGER;

v_Vektor t_vektor;

BEGIN

FOR i IN -2..2 LOOP

v_Vektor(i*2) := i;

END LOOP;

DBMS_OUTPUT.PUT_LINE(v_Vektor.COUNT);

END;

/

/*

Eredmény:

5

A PL/SQL eljárás sikeresen befejeződött.

*/
```

LIMIT

Paraméter nélküli függvény. Beágyazott tábla és asszociatív tömb esetén visszatérési értéke NULL. Dinamikus tömb esetén a típusdefinícióban megadott maximális méretet adja.

Példa

```
DECLARE

v_Szerzok T_Szerzok;

v_Konyvek T_Konyvek;

BEGIN

SELECT szerzo INTO v_Szerzok FROM konyv WHERE id = 15;

SELECT konyvek INTO v_Konyvek FROM ugyfel WHERE id = 10;

DBMS_OUTPUT.PUT_LINE('1. szerzo count: ' || v_Szerzok.COUNT

|| ' Limit: ' || NVL(TO_CHAR(v_Szerzok.LIMIT), 'NULL'));

DBMS_OUTPUT.PUT_LINE('2. konyvek count: ' || v_Konyvek.COUNT

|| ' Limit: ' || NVL(TO_CHAR(v_Konyvek.LIMIT), 'NULL'));

END;

/
```

```
/*

Eredmény:

1. szerzo count: 2 Limit: 10

2. konyvek count: 3 Limit: NULL

A PL/SQL eljárás sikeresen befejeződött.

*/
```

FIRST és LAST

Paraméter nélküli függvények. A FIRST a kollekció legelső (nem törölt) elemének indexét (a kollekció legkisebb indexét), a LAST a legutolsó (nem törölt) elem indexét (a kollekció legnagyobb indexét) adja vissza. Ha a kollekció üres, akkor értékük NULL. Dinamikus tömbnél a FIRST visszatérési értéke 1, a LAST-é COUNT.

VARCHAR2 típusú kulccsal rendelkező asszociatív tömbnél a legkisebb és legnagyobb kulcsértéket adják meg. Ha az NLS_COMP inicializációs paraméter értéke ANSI, akkor a kulcsok rendezési sorrendjét az NLS_SORT inicializációs paraméter határozza meg.

1. példa

```
DECLARE

v_Szerzok T_Szerzok;

j PLS_INTEGER;

BEGIN

BEGIN

j := v_Szerzok.FIRST;

EXCEPTION

WHEN COLLECTION_IS_NULL THEN

DBMS_OUTPUT.PUT_LINE('Kivétel! ' || SQLERRM);

END;

v_Szerzok := T_Szerzok();

DBMS_OUTPUT.PUT_LINE('first: '

|| NVL(TO_CHAR(v_Szerzok.FIRST), 'NULL')

|| ' last: ' || NVL(TO_CHAR(v_Szerzok.LAST), 'NULL'));

DBMS_OUTPUT.NEW_LINE;

SELECT szerzo

INTO v_Szerzok

FROM konyv

WHERE id = 15;

FOR i IN v_Szerzok.FIRST..v_Szerzok.LAST LOOP

DBMS_OUTPUT.PUT_LINE(v_Szerzok(i));

END LOOP;
```

```
END;
/
/*
Eredmény:
Kivétel! ORA-06531: Inicializálatlan gyűjtőre való hivatkozás
first: NULL last: NULL
P. Howard
Rejtő Jenő
A PL/SQL eljárás sikeresen befejeződött.
*/
```

2. példa

```
DECLARE
TYPE t_vektor IS TABLE OF NUMBER
INDEX BY BINARY_INTEGER;
v_Vektor t_vektor;
BEGIN
FOR i IN -2..2 LOOP
v_Vektor(i*2) := i;
END LOOP;
DBMS_OUTPUT.PUT_LINE('first: '
|| NVL(TO_CHAR(v_Vektor.FIRST), 'NULL')
|| ' last: ' || NVL(TO_CHAR(v_Vektor.LAST), 'NULL'));
END;
/
/*
Eredmény:
first: -4 last: 4
A PL/SQL eljárás sikeresen befejeződött.
*/
```

NEXT és PRIOR

Alakjuk:

```
NEXT(i)
PRIOR(i)
```

ahol *i* az index típusának megfelelő kifejezés.

A NEXT visszaadja az *i* indexű elemet követő (nem törölt), a PRIOR a megelőző (nem törölt) elem indexét. Ha ilyen elem nem létezik, értékük NULL.

VARCHAR2 típusú kulccsal rendelkező asszociatív tömbnél a sztringek rendezettségének megfelelő következő és megelőző kulcsértéket adják meg. Ha az NLS_COMP inicializációs paraméter értéke ANSI, akkor a kulcsok rendezési sorrendjét az NLS_SORT inicializációs paraméter határozza meg.

A FIRST, LAST, NEXT, PRIOR alkalmas arra, hogy bármely kollekció aktuális elemeit növekvő vagy csökkenő indexek szerint feldolgozzuk.

1. példa

```
DECLARE

TYPE t_vektor IS TABLE OF NUMBER

INDEX BY BINARY_INTEGER;

v_Vektor t_vektor;

i PLS_ INTEGER;

BEGIN

FOR i IN -2..2 LOOP

v_Vektor(i*2) := i;

END LOOP;

i := v_Vektor.FIRST;

WHILE i IS NOT NULL LOOP

DBMS_OUTPUT.PUT_LINE(LPAD(i,2) || ' '

|| LPAD(v_Vektor(i), 2));

i := v_Vektor.NEXT(i);

END LOOP;

DBMS_OUTPUT.NEW_LINE;

i := v_Vektor.LAST;

WHILE i IS NOT NULL LOOP

DBMS_OUTPUT.PUT_LINE(LPAD(i,2) || ' '

|| LPAD(v_Vektor(i), 2));

i := v_Vektor.PRIOR(i);

END LOOP;

END;

/

/*

Eredmény:

-4 -2

-2 -1

0 0

2 1

4 2
```

```
4 2
2 1
0 0
-2 -1
-4 -2
A PL/SQL eljárás sikeresen befejeződött.
*/
```

2. példa

```
CREATE OR REPLACE PROCEDURE elofordulasok(p_Szoveg VARCHAR2)
IS
c VARCHAR2(1 CHAR);
TYPE t_gyakorisag IS TABLE OF NUMBER
INDEX BY c%TYPE;
v_Elofordulasok t_gyakorisag;
BEGIN
FOR i IN 1..LENGTH(p_Szoveg)
LOOP
c := LOWER(SUBSTR(p_Szoveg, i, 1));
IF v_Elofordulasok.EXISTS(c) THEN
v_Elofordulasok(c) := v_Elofordulasok(c)+1;
ELSE
v_Elofordulasok(c) := 1;
END IF;
END LOOP;
-- Fordított sorrendhez LAST és PRIOR kellene
c := v_Elofordulasok.FIRST;
WHILE c IS NOT NULL LOOP
DBMS_OUTPUT.PUT_LINE(' ''' || c || ''' - '
|| v_Elofordulasok(c));
c := v_Elofordulasok.NEXT(c);
END LOOP;
END elofordulasok;
/
show errors;
ALTER SESSION SET NLS_COMP='ANSI';
```

```
-- Ha az NLS_LANG magyar, akkor az NLS_SORT is magyar rendezést ír most elő

-- Egyébként kell még: ALTER SESSION SET NLS_SORT='Hungarian';

EXEC elofordulasok('Babámé');

/*

'a' - 1

'á' - 1

'b' - 2

'é' - 1

'm' - 1

*/

ALTER SESSION SET NLS_COMP='BINARY';

EXEC elofordulasok('Babámé');

/*

'a' - 1

'b' - 2

'm' - 1

'á' - 1

'é' - 1

*/
```

EXTEND

Alakja:

```
EXTEND[(n[,m])]
```

ahol *n* és *m* egész.

Beágyazott tábla vagy dinamikus tömb aktuális méretének növelésére szolgál. A paraméter nélküli EXTEND egyetlen NULL elemet helyez el a kollekció végén. Az EXTEND(*n*) *n* darab NULL elemet helyez el a kollekció végére. EXTEND(*n*,*m*) esetén pedig az *m* indexű elem *n*-szer helyeződik el a kollekció végén. Ha a kollekció típusának deklarációjában szerepel a NOT NULL megszorítás, csak az EXTEND harmadik formája alkalmazható.

Egy dinamikus tömb csak a deklarált maximális méretig terjeszthető ki (*n* értéke legfeljebb LIMIT – COUNT lehet).

Az EXTEND a kollekció belső méretén operál. A PL/SQL megtartja a törölt elemek helyét, ezeknek új érték adható. Az EXTEND használatánál, ha a kollekció végén törölt elemek vannak, a bővítés mögéjük történik.

Például, ha létrehozunk egy beágyazott táblát 5 elemmel, majd a 2. és 5. elemet töröljük, akkor a bővítés az 5. elem **után** történik (tehát a 6. elemtől). Ekkor COUNT értéke 3, LAST értéke 4.

Példa

```
DECLARE

v_Szerzok T_Szerzok := T_Szerzok();

BEGIN
```

```
DBMS_OUTPUT.PUT_LINE('1. count: ' || v_Szerzok.COUNT
|| ' Limit: ' || NVL(TO_CHAR(v_Szerzok.LIMIT), 'NULL'));
-- Egy NULL elemmel bővítünk
v_Szerzok.EXTEND;
DBMS_OUTPUT.PUT_LINE('2. count: ' || v_Szerzok.COUNT
|| ' v_Szerzok(1): ' || NVL(v_Szerzok(1), 'NULL'));
v_Szerzok(1) := 'Móra Ferenc';
DBMS_OUTPUT.PUT_LINE('2. count: ' || v_Szerzok.COUNT
|| ' v_Szerzok(v_Szerzok.COUNT): '
|| NVL(v_Szerzok(v_Szerzok.COUNT), 'NULL'));
-- 3 NULL elemmel bővítünk
v_Szerzok.EXTEND(3);
DBMS_OUTPUT.PUT_LINE('2. count: ' || v_Szerzok.COUNT
|| ' v_Szerzok(v_Szerzok.COUNT): '
|| NVL(v_Szerzok(v_Szerzok.COUNT), 'NULL'));
-- 4 elemmel bővítünk, ezek értéke az 1. elem értékét veszi fel.
v_Szerzok.EXTEND(4, 1);
DBMS_OUTPUT.PUT_LINE('2. count: ' || v_Szerzok.COUNT
|| ' v_Szerzok(v_Szerzok.COUNT): '
|| NVL(v_Szerzok(v_Szerzok.COUNT), 'NULL'));
BEGIN
-- Megpróbáljuk a dinamikus tömböt túlbővíteni
v_Szerzok.EXTEND(10);
EXCEPTION
WHEN SUBSCRIPT_OUTSIDE_LIMIT THEN
DBMS_OUTPUT.PUT_LINE('Kivétel! ' || SQLERRM);
DBMS_OUTPUT.NEW_LINE;
FOR i IN 1..v_Szerzok.COUNT LOOP
DBMS_OUTPUT.PUT_LINE(LPAD(i,2) || ' '
|| NVL(v_Szerzok(i), 'NULL'));
END LOOP;
END;
/
/*
Eredmény:
1. count: 0 Limit: 10
```

```
2. count: 1 v_Szerzok(1): NULL

2. count: 1 v_Szerzok(v_Szerzok.COUNT): Móra Ferenc

2. count: 4 v_Szerzok(v_Szerzok.COUNT): NULL

2. count: 8 v_Szerzok(v_Szerzok.COUNT): Móra Ferenc

Kivétel! ORA-06532: Határon kívüli index

1 Móra Ferenc

2 NULL

3 NULL

4 NULL

5 Móra Ferenc

6 Móra Ferenc

7 Móra Ferenc

8 Móra Ferenc

A PL/SQL eljárás sikeresen befejeződött.

*/
```

TRIM

Alakja

```
TRIM[(n)]
```

ahol *n* egész.

TRIM eltávolítja a kollekció utolsó elemét, TRIM(*n*) pedig törli az utolsó *n* elemet. Ha *n*>COUNT, akkor a SUBSCRIPT_BEYOND_COUNT kivétel váltódik ki. A TRIM a belső méreten operál és az eltávolított elemek helye nem őrződik meg, ezért egy értékadással nem adható nekik új érték.

1. példa

```
DECLARE

v_Szerzok T_Szerzok;

BEGIN

v_Szerzok := T_Szerzok('Móricz Zsigmond', 'Móra Ferenc',

'Ottlik Géza', 'Weöres Sándor');

DBMS_OUTPUT.PUT_LINE('1. count: ' || v_Szerzok.COUNT);

-- Törlünk egy elemet

v_Szerzok.TRIM;

DBMS_OUTPUT.PUT_LINE('2. count: ' || v_Szerzok.COUNT);

-- Törlünk 2 elemet

v_Szerzok.TRIM(2);

DBMS_OUTPUT.PUT_LINE('3. count: ' || v_Szerzok.COUNT);

BEGIN
```

```
-- Megpróbálunk túl sok elemet törölni
v_Szerzok.TRIM(10);
EXCEPTION
WHEN SUBSCRIPT_BEYOND_COUNT THEN
DBMS_OUTPUT.PUT_LINE('Kivétel! ' || SQLERRM);
END;
DBMS_OUTPUT.PUT_LINE('4. count: ' || v_Szerzok.COUNT);
END;
/
/*
Eredmény:
1. count: 4
2. count: 3
3. count: 1
Kivétel! ORA-06533: Számlálón kívüli indexérték
4. count: 1
A PL/SQL eljárás sikeresen befejeződött.
*/
```

2. példa

```
DECLARE
v_Konyvek T_Konyvek := T_Konyvek();
BEGIN
v_Konyvek.EXTEND(20);
DBMS_OUTPUT.PUT_LINE('1. count: ' || v_Konyvek.COUNT);
-- Törlünk pár elemet
v_Konyvek.TRIM(5);
DBMS_OUTPUT.PUT_LINE('2. count: ' || v_Konyvek.COUNT);
END;
/
/*
Eredmény:
1. count: 20
2. count: 15
A PL/SQL-eljárás sikeresen befejeződött.
*/
```

DELETE

Alakja

```
DELETE[(i[,j])]
```

ahol *i* és *j* egész.

A paraméter nélküli DELETE törli a kollekció összes elemét (üres kollekció jön létre). DELETE(*i*) törli az *i* indexű elemet, DELETE(*i*,*j*) pedig az *i* és *j* indexek közé eső minden elemet. Ha *i*>*j* vagy valamelyik NULL, akkor a DELETE nem csinál semmit. Dinamikus tömb esetén csak a paraméter nélküli alak használható.

VARCHAR2 típusú kulccsal rendelkező asszociatív tömbnél ha az NLS_COMP inicializációs paraméter értéke ANSI, akkor a kulcsok rendezési sorrendjét az NLS_SORT inicializációs paraméter határozza meg.

Ha beágyazott táblából paraméteres DELETE metódussal törlünk egy vagy több elemet, az lyukat hozhat létre a beágyazott táblában. Az így törölt elemek által lefoglalt hely továbbra is a memóriában marad (a kollekció belső mérete nem változik), de az elemek értékére történő hivatkozás NO_DATA_FOUND kivételt vált ki. A törölt indexeket a FIRST, LAST, NEXT és PRIOR metódusok figyelmen kívül hagyják, a COUNT értéke a törölt elemek számával csökken. A paraméteres alakkal törölt elemeket újra lehet használni egy értékadás után. Tehát az így törölt elemekre nézve a beágyazott tábla az asszociatív tömbhöz hasonlóan viselkedik.

1. példa

```
DECLARE

TYPE t_nevek IS TABLE OF VARCHAR2(10);

v_Nevek t_nevek := t_nevek('A1', 'B2', 'C3',

'D4', 'E5', 'F6', 'G7', 'H8', 'I9', 'J10');

i PLS_INTEGER;

BEGIN

DBMS_OUTPUT.PUT_LINE('1. count: ' || v_Nevek.COUNT);

-- Törlünk pár elemet

v_Nevek.DELETE(3);

v_Nevek.DELETE(6, 8);

-- Gond nélkül megy ez is

v_Nevek.DELETE(10, 12);

v_Nevek.DELETE(60);

DBMS_OUTPUT.PUT_LINE('2. count: ' || v_Nevek.COUNT);

DBMS_OUTPUT.NEW_LINE;

i := v_Nevek.FIRST;

WHILE i IS NOT NULL LOOP

DBMS_OUTPUT.PUT_LINE(LPAD(i,2) || ' '

|| LPAD(v_Nevek(i), 2));

i := v_Nevek.NEXT(i);

END LOOP;

END;
```

```
/

/*

Eredmény:

1. count: 10

2. count: 5

1 A1

2 B2

4 D4

5 E5

9 I9

A PL/SQL eljárás sikeresen befejeződött.

*/
```

2. példa

```
DECLARE

TYPE t_vektor IS TABLE OF NUMBER

INDEX BY BINARY_INTEGER;

v_Vektor t_vektor;

BEGIN

FOR i IN -2..2 LOOP

v_Vektor(i*2) := i;

END LOOP;

DBMS_OUTPUT.PUT_LINE('1. count: ' || v_Vektor.COUNT);

-- Hány tényleges elem esik ebbe az intervallumba?

v_Vektor.DELETE(-1, 2);

DBMS_OUTPUT.PUT_LINE('2. count: ' || v_Vektor.COUNT);

END;

/

/*

Eredmény:

1. count: 5

2. count: 3

A PL/SQL eljárás sikeresen befejeződött.

*/
```

3. példa

```
DECLARE
```

```
TYPE t_tablazat IS TABLE OF NUMBER

v_Tablazat t_tablazat;

BEGIN

v_Tablazat('a') := 1;

v_Tablazat('A') := 2;

v_Tablazat('z') := 3;

v_Tablazat('Z') := 4;

DBMS_OUTPUT.PUT_LINE('1. count: ' || v_Tablazat.COUNT);

v_Tablazat.DELETE('a','z');

DBMS_OUTPUT.PUT_LINE('2. count: ' || v_Tablazat.COUNT);

END;

/

/*

Eredmény:

1. count: 4

2. count: 2

A PL/SQL eljárás sikeresen befejeződött.

*/
```

4. Együttes hozzárendelés

A PL/SQL motor minden procedurális utasítást végrehajt, azonban a PL/SQL programba beépített SQL utasításokat átadja az SQL motornak. Az SQL motor fogadja az utasítást, végrehajtja azt és esetleg adatokat szolgáltat vissza a PL/SQL motornak. Minden egyes ilyen motorváltás növeli a végrehajtási időt. Ha sok a váltás, csökken a teljesítmény. Különösen igaz ez, ha az SQL utasítás ciklus magjába van ágyazva, például egy kollekció elemeinek egyenkénti feldolgozásánál.

1. példa

```
DECLARE

/* Összegyűjtjük a lejárt kölcsönzésekhez tartozó ügyfeleket */

TYPE t_ugyfelek IS TABLE OF ugyfel%ROWTYPE;

TYPE t_konyvek IS TABLE OF konyv%ROWTYPE;

v_Ugyfelek t_ugyfelek := t_ugyfelek();

v_Konyvek t_konyvek := t_konyvek();

-- Rögzítjük a mát a példa kedvéért

v_Ma DATE := TO_DATE('2002-05-11', 'YYYY-MM-DD');

PROCEDURE feldolgoz(

p_Ugyfelek t_ugyfelek,

p_Konyvek t_konyvek
```

```
) IS
BEGIN
FOR i IN 1..p_Ugyfelek.COUNT LOOP
DBMS_OUTPUT.PUT_LINE(p_Ugyfelek(i).nev || ' - '
|| v_Konyvek(i).cim);
END LOOP;
END feldolgoz;
BEGIN
-- Ebben a ciklusban sok a SELECT, sok a motorváltás
FOR bejegyzes IN (SELECT * FROM kolcsonzes) LOOP
IF TRUNC(bejegyzes.datum)
+ 30*(bejegyzes.hosszabbitva + 1) < v_Ma THEN
v_Ugyfelek.EXTEND;
v_Konyvek.EXTEND;
SELECT *
INTO v_Ugyfelek(v_Ugyfelek.LAST)
FROM ugyfel
WHERE id = bejegyzes.kolcsonzo;
SELECT *
INTO v_Konyvek(v_Konyvek.LAST)
FROM konyv
WHERE id = bejegyzes.konyv;
END IF;
END LOOP;
END;
/
/*
Eredmény:
József István - Piszkos Fred és a többiek
József István - ECOOP 2001 - Object-Oriented Programming
József István - Java - start!
József István - Matematikai zseblexikon
József István - Matematikai Kézikönyv
Jaripekka Hämälainen - A critical introduction to twentieth-century
American drama - Volume 2
Jaripekka Hämälainen - The Norton Anthology of American Literature - Second
```

```
Edition - Volume 2

A PL/SQL eljárás sikeresen befejeződött.

*/
```

Hozzárendelésnek hívjuk azt a tevékenységet, amikor egy PL/SQL változónak SQL utasításban adunk értéket. Egy kollekció minden elemének egyszerre történő hozzárendelését *együttes hozzárendelésnek* nevezzük. Az együttes hozzárendelés csökkenti a PL/SQL és SQL motorok közötti átváltások számát és így növeli a teljesítményt.

A PL/SQL oldali együttes hozzárendelés eszköze a FORALL utasítás, az SQL oldalié a BULK COLLECT utasításrész. A FORALL utasítás alakja:

```
FORALL index IN {alsó_határ..felső_határ

| INDICES OF kollekció

[BETWEEN alsó_határ AND felső_határ]

| VALUES OF indexkollekció_név}

[SAVE EXCEPTIONS] sql_utasítás;
```

Az *index* explicit módon nem deklarált változó, amelynek hatásköre a FORALL utasítás, és azon belül is csak kollekció indexeként használható fel. Kifejezésben nem szerepelhet és érték nem adható neki. Az *alsó_határ* és *felső_határ* numerikus értékű kifejezések, amelyek egyszer értékelődnek ki a FORALL végrehajtásának kezdetén, és értékük egész vagy egészre kerekítődik. Az egészeknek egy érvényes kollekcióindex-tartományt kell megadniuk.

Az INDICES OF utasításrész esetén az *index* a megadott *kollekció* elemeinek indexeit veszi fel. A BETWEEN segítségével ezt az indextartományt korlátozhatjuk le. Ha az indextartomány valamely indexe a kollekcióban nem létezik, akkor figyelmen kívül marad. Ez az utasításrész törölt elemeket tartalmazó beágyazott tábla vagy numerikus kulcsú asszociatív tömb esetén használható.

A VALUES OF utasításrész azt írja elő, hogy az *index* által felveendő értékeket egy, az *indexkollekció_név* által megnevezett kollekció tartalmazza. Ekkor a megadott indexkollekció tetszőleges (akár ismétlődő) indexeket tartalmazhat. Az indexkollekció beágyazott tábla, vagy numerikus kulcsú asszociatív tömb lehet, az elemek típusa pedig vagy PLS_INTEGER vagy BINARY_INTEGER. Ha az indexkollekció üres, akkor a FORALL nem fut le, és kivétel váltódik ki.

Az *sql_utasítás* egy olyan INSERT, DELETE vagy UPDATE utasítás, amely kollekcióelemeket hivatkozik WHERE, VALUES vagy SET utasításrészében. Összetett adattípust tartalmazó kollekciók esetén a ciklusváltozóval történő indexelés után a további alstruktúrákba történő hivatkozás nem megengedett, azaz nem hivatkozhatunk rekord elemek esetén a rekordelem mezőire, objektum elemeknél attribútumokra, kollekció elemeknél a beágyazott kollekciók egyes elemeire.

Az SQL motor az SQL utasítást a megadott indextartomány minden értéke mellett egyszer végrehajtja. Az adott indexű kollekcióelemeknek létezniük kell.

A FORALL utasítás csak szerveroldali programokban alkalmazható.

2. példa

```
CREATE OR REPLACE TYPE T_Id_lista IS

TABLE OF NUMBER

/

/* FORALL nélkül */

CREATE OR REPLACE PROCEDURE kolcsonzes_torol(

p_Ugyfelek T_Id_lista,
```

```
p_Konyvek T_Id_lista
) IS
/* Több kölcsönzésbejegyzést töröl, nem módosítja a
szabad példányok számát stb. */
BEGIN
FOR i IN 1..p_Ugyfelek.COUNT LOOP
DELETE FROM kolcsonzes
WHERE kolcsonzo = p_Ugyfelek(i)
AND konyv = p_Konyvek(i);
END LOOP;
END kolcsonzes_torol;
/
show errors
/* FORALL használatával */
CREATE OR REPLACE PROCEDURE kolcsonzes_torol(
p_Ugyfelek T_Id_lista,
p_Konyvek T_Id_lista
) IS
/* Több kölcsönzésbejegyzést töröl, nem módosítja a
szabad példányok számát stb. */
BEGIN
FORALL i IN 1..p_Ugyfelek.COUNT
DELETE FROM kolcsonzes
WHERE kolcsonzo = p_Ugyfelek(i)
AND konyv = p_Konyvek(i);
END kolcsonzes_torol;
/
show errors
/*
Megjegyzés:
Szintaktikájukban alig van különbség, viszont
a FORALL esetében csak egy hozzárendelés van, FORALL
nélkül annyi, ahányszor lefut a ciklusmag.
*/
```

3. példa

```
INDICES OF és VALUES OF használatára
CREATE TABLE t1
AS
SELECT ROWNUM id, LPAD('x', 10) adat FROM dual CONNECT BY LEVEL <= 4;
/
SELECT * FROM t1;
/*
ID ADAT
---------- ----------
1 x
2 x
3 x
4 x
*/
/* FORALL - INDICES OF */
DECLARE
TYPE T_Id_lista IS TABLE OF NUMBER;
v_Id_lista T_Id_lista := T_Id_lista(3,1);
BEGIN
-- Töröljük az 1. elemet, 1 elem marad
v_Id_lista.DELETE(1);
FORALL i IN INDICES OF v_Id_lista -- i: {2}
UPDATE t1
SET adat = 'INDICES OF'
WHERE id = v_Id_lista(i)
; -- T_Id_lista(2) = 1 -> 1-es Id-re fut le az UPDATE
END;
/
SELECT * FROM t1;
/*
ID ADAT
---------- ----------
1 INDICES OF
2 x
3 x
4 x
```

```
*/

/* FORALL - VALUES OF */

DECLARE

TYPE T_Id_lista IS TABLE OF NUMBER;

TYPE T_Index_lista IS TABLE OF BINARY_INTEGER;

v_Id_lista T_Id_lista := T_Id_lista(5,1,4,3,2);

v_Index_lista T_Index_lista := T_Index_lista(4,5);

BEGIN

FORALL i IN VALUES OF v_Index_lista -- i: {4, 5}

UPDATE t1

SET adat = 'VALUES OF'

WHERE id = v_Id_lista(i)

;

-- p_Id_lista(4) = 3

-- p_Id_lista(5) = 2

-- -> 3-as és 2-es Id-kre fut le az UPDATE

END;

/

SELECT * FROM t1;

/*

ID ADAT

---------- ----------

1 INDICES OF

2 VALUES OF

3 VALUES OF

4 x

*/

DROP TABLE t1;
```

Ha egy FORALL utasításban az SQL utasítás nem kezelt kivételt vált ki, akkor az egész FORALL visszagörgetődik. Ha viszont a kiváltott kivételt kezeljük, akkor csak a kivételt okozó végrehajtás görgetődik vissza az SQL utasítás előtt elhelyezett implicit mentési pontig, a korábbi végrehajtások eredménye megmarad.

4. példa

```
CREATE OR REPLACE PROCEDURE kolcsonoz(

p_Ugyfelek T_Id_lista,

p_Konyvek T_Id_lista

) IS
```

```
/* Több könyv kölcsönzését végzi el.
p_Ugyfelek és p_Konyvek mérete meg kell egyezzen. */
v_Most DATE := SYSDATE;
v_Tulkolcsonzes NUMBER;
BEGIN
SAVEPOINT kezdet;
/* Ha nem létezik az ügyfél vagy a könyv, akkor a ciklus
magjában lesz egy kivétel. */
FORALL i IN 1..p_Ugyfelek.COUNT
INSERT INTO kolcsonzes VALUES
(p_Ugyfelek(i), p_Konyvek(i), v_Most, 0,
/* Ha elfogy valamelyik könyv, akkor lesz egy kivétel */
FORALL i IN 1..p_Ugyfelek.COUNT
UPDATE konyv SET szabad = szabad -1
WHERE id = p_Konyvek(i);
/* Az ügyfelek konyvek táblájának bővítése */
FORALL i IN 1..p_Ugyfelek.COUNT
INSERT INTO TABLE(SELECT konyvek FROM ugyfel WHERE id = p_Ugyfelek(i))
VALUES (p_Konyvek(i), v_Most);
/* Nézzük, lett-e túlkölcsönzés ? */
SELECT COUNT(1) INTO v_Tulkolcsonzes
FROM ugyfel
WHERE max_konyv < (SELECT COUNT(1) FROM TABLE(konyvek));
IF v_Tulkolcsonzes > 0 THEN
RAISE_APPLICATION_ERROR(-20010,
'Valaki túl sok könyvet akar kölcsönözni');
END IF;
EXCEPTION
WHEN OTHERS THEN
ROLLBACK TO kezdet;
RAISE;
END kolcsonoz;
/
show errors
SELECT COUNT(1) "Kölcsönzések" FROM kolcsonzes;
BEGIN
```

```
kolcsonoz(T_Id_lista(10,10,15,15,15), T_Id_lista(35,35,25,20,10));
END;
/
BEGIN
kolcsonoz(T_Id_lista(15), T_Id_lista(5));
END;
/
SELECT COUNT(1) "Kölcsönzések" FROM kolcsonzes;
/*
Kölcsönzések
------------
14
BEGIN
*
Hiba a(z) 1. sorban:
ORA-02290: ellenőrző megszorítás (PLSQL.KONYV_SZABAD) megsértése
ORA-06512: a(z) "PLSQL.KOLCSONOZ", helyen a(z) 40. sornál
ORA-06512: a(z) helyen a(z) 2. sornál
BEGIN
*
Hiba a(z) 1. sorban:
ORA-20010: Valaki túl sok könyvet akar kölcsönözni
ORA-06512: a(z) "PLSQL.KOLCSONOZ", helyen a(z) 40. sornál
ORA-06512: a(z) helyen a(z) 2. sornál
Kölcsönzések
------------
14
*/
```

A DML utasítások végrehajtásához az SQL motor felépíti az implicit kurzort (*lásd* 8. fejezet). A FORALL utasításhoz kapcsolódóan a szokásos kurzorattribútumok (%FOUND, %ISOPEN, %NOTFOUND, %ROWCOUNT) mellett az implicit kurzornál használható a %BULK_ROWCOUNT attribútum is. Ezen attribútum szemantikája megegyezik egy asszociatív tömbével. Az attribútum *i.* eleme a DML utasítás *i.* futásánál feldolgozott sorok számát tartalmazza. Értéke 0, ha nem volt feldolgozott sor. Indexeléssel lehet rá hivatkozni. A %BULK_ROWCOUNT indextartománya megegyezik a FORALL indextartományával.

5. példa

```
CREATE OR REPLACE PROCEDURE ugyfel_visszahoz(
p_Ugyfelek T_Id_lista
```

```
) IS
/* Több ügyfél minden könyvének visszahozatalát adminisztrálja
a függvény. Kiírja, hogy ki hány könyvet hozott vissza. */
BEGIN
/* Könyvek szabad példányainak adminisztrálása */
FOR k IN (
SELECT konyv, COUNT(1) peldany FROM kolcsonzes
WHERE kolcsonzo IN (SELECT COLUMN_VALUE
FROM TABLE(CAST(p_Ugyfelek AS T_Id_lista)))
GROUP BY konyv
) LOOP
UPDATE konyv SET szabad = szabad + k.peldany
WHERE id = k.konyv;
END LOOP;
/* Az ügyfelek konyvek tábláinak üresre állítása. */
FORALL i IN 1..p_Ugyfelek.COUNT
UPDATE ugyfel SET konyvek = T_Konyvek()
WHERE id = p_Ugyfelek(i);
/* A kölcsönzések törlése */
FORALL i IN 1..p_Ugyfelek.COUNT
/* A %BULK_ROWCOUNT segítségével jelentés készítése */
FOR i IN 1..p_Ugyfelek.COUNT LOOP
DBMS_OUTPUT.PUT_LINE('Ügyfél: ' || p_Ugyfelek(i)
|| ', visszahozott könyvek: ' || SQL%BULK_ROWCOUNT(i));
END LOOP;
END ugyfel_visszahoz;
/
show errors
```

A FORALL utasítás SAVE EXCEPTIONS utasításrésze lehetőséget ad arra, hogy a FORALL működése közben kiváltódott kivételeket tároljuk, csak az utasítás végrehajtása után kezeljük őket. Az Oracle ehhez egy új kurzorattribútumot értelmez, amelynek neve %BULK_EXCEPTIONS. Ez rekordok asszociatív tömbje. A rekordoknak két mezőjük van. A %BULK_EXCEPTIONS(i). ERROR_INDEX a FORALL indexének azon értékét tartalmazza, amelynél a kivétel bekövetkezett, a %BULK_EXCEPTIONS(i). ERROR_CODE értéke pedig a megfelelő Oracle hibakód.

A %BULK_EXCEPTIONS mindig a legutoljára végrehajtott FORALL információit tartalmazza. Az eltárolt kivételek számát a %BULK_EXCEPTIONS.COUNT szolgáltatja, az indexek 1-től eddig mehetnek.

Ha a SAVE EXCEPTIONS utasításrészt nem adjuk meg, akkor egy kivétel bekövetkezte után a FORALL működése befejeződik. Ekkor a %BULK_EXCEPTIONS a bekövetkezett kivétel információit tartalmazza csak.

6. példa

```
CREATE OR REPLACE TYPE T_Id_lista IS
TABLE OF NUMBER;
CREATE OR REPLACE TYPE T_Szamok IS
TABLE OF NUMBER;
/
CREATE OR REPLACE FUNCTION selejtez(
p_Konyvek T_Id_lista,
p_Mennyit T_Szamok
) RETURN T_Id_lista IS
/* A könyvtárból selejtezi a megadott könyvekből
a megadott számú példányt. Visszaadja azoknak a könyveknek
az azonosítóit, amelyekből nem lehet ennyit selejtezni
(mert nincs annyi vagy kölcsönzik). Ha nem volt ilyen,
akkor üres kollekciót (nem NULL-t) ad vissza.
A paraméterek méretének meg kell egyeznie.
*/
v_Konyvek T_Id_lista := T_Id_lista();
v_Index PLS_INTEGER;
bulk_kivetel EXCEPTION;
PRAGMA EXCEPTION_INIT(bulk_kivetel, -24381);
BEGIN
/* Amit tudunk módosítunk */
FORALL i IN 1..p_Konyvek.COUNT SAVE EXCEPTIONS
UPDATE konyv SET szabad = szabad - p_Mennyit(i),
keszlet = keszlet - p_Mennyit(i)
WHERE id = p_Konyvek(i);
RETURN v_Konyvek;
EXCEPTION
WHEN bulk_kivetel THEN
/* A sikertelen módosítások összegyűjtése */
FOR i IN 1..SQL%BULK_EXCEPTIONS.COUNT LOOP
v_Index := SQL%BULK_EXCEPTIONS(i).ERROR_INDEX;
v_Konyvek.EXTEND;
v_Konyvek(v_Konyvek.LAST) := p_Konyvek(v_Index);
```

```
END LOOP;

RETURN v_Konyvek;

END selejtez;

/

show errors

DECLARE

/* Kipróbáljuk */

v_Konyvek T_Id_lista;

v_Konyv konyv%ROWTYPE;

BEGIN

DBMS_OUTPUT.NEW_LINE;

DBMS_OUTPUT.PUT_LINE('Könyvek selejtezése: 5, 35');

DBMS_OUTPUT.NEW_LINE;

v_Konyvek := selejtez(T_Id_lista(5, 35), T_Szamok(1, 1));

IF v_Konyvek.COUNT > 0 THEN

DBMS_OUTPUT.PUT_LINE('Voltak hibák - nem mindent lehetett törölni');

FOR i IN 1..v_Konyvek.COUNT LOOP

SELECT * INTO v_Konyv FROM konyv WHERE id = v_Konyvek(i);

DBMS_OUTPUT.PUT_LINE(v_Konyv.id || ', ' || v_Konyv.szabad

|| ', ' || v_Konyv.cim );

END LOOP;

END IF;

END;

/

/*

Eredmény:

Könyvek selejtezése: 5, 35

Voltak hibák - nem mindent lehetett törölni

35, 0, A critical introduction to twentieth-century American drama - Volume 2

A PL/SQL eljárás sikeresen befejeződött.

*/
```

A SELECT, INSERT, DELETE, UPDATE, FETCH utasítások INTO utasításrészében használható a BULK_COLLECT előírás, amely az SQL motortól az együttes hozzárendelést kéri. Alakja:

```
BULK COLLECT INTO kollekciónév [,kollekciónév]…
```

A kollekciók elemtípusainak rendre meg kell egyezniük az eredmény oszlopainak típusaival. Több oszlop tartalmát egy megfelelő *rekord* elemtípusú kollekcióba is össze lehet gyűjteni.

7. példa

```
CREATE OR REPLACE FUNCTION aktiv_kolcsonzok
RETURN T_Id_lista IS
v_Id_lista T_Id_lista; — BULK COLLECT-nél nem kell inicializálni
BEGIN
SELECT DISTINCT kolcsonzo
BULK COLLECT INTO v_Id_lista
FROM kolcsonzes
ORDER BY kolcsonzo;
RETURN v_Id_lista;
END aktiv_kolcsonzok;
/
SELECT * FROM TABLE(aktiv_kolcsonzok);
/*
COLUMN_VALUE
------------
10
15
20
25
30
35
6 sor kijelölve.
*/
DECLARE
/* Növeljük a kölcsönzött könyvek példányszámait 1-gyel.
A módosított könyvek azonosítóit összegyűjtjük. */
v_Konyvek T_Id_lista;
BEGIN
UPDATE konyv k
SET keszlet = keszlet + 1,
szabad = szabad + 1
WHERE EXISTS (SELECT 1 FROM kolcsonzes
WHERE konyv = k.id)
RETURNING id
```

```
BULK COLLECT INTO v_Konyvek;

DBMS_OUTPUT.PUT_LINE('count: ' || v_Konyvek.COUNT);

END;

/

/*

Eredmény:

count: 10

A PL/SQL eljárás sikeresen befejeződött.

*/
```

A BULK COLLECT mind az implicit, mind az explicit kurzorok esetén használható. Az adatokat a kollekcióban az 1-es indextől kezdve helyezi el folyamatosan, felülírva az esetleges korábbi elemeket.

Az együttes hozzárendelést tartalmazó FETCH utasításnak lehet egy olyan utasításrésze, amely a betöltendő sorok számát korlátozza. Ennek alakja:

```
FETCH … BULK COLLECT INTO … LIMIT sorok;
```

ahol a *sorok* pozitív számértéket szolgáltató kifejezés. A kifejezés értéke egészre kerekítődik, ha szükséges. Ha értéke negatív, akkor az INVALID_NUMBER kivétel váltódik ki. Ezzel korlátozhatjuk a betöltendő sorok számát.

8. példa

```
DECLARE

CURSOR cur_ugyfel_konyvek IS

SELECT nev, konyvek FROM ugyfel;

-- Kollekció rekord típusú elemekkel

TYPE t_ugyfel_adatok IS

VARRAY(4) OF cur_ugyfel_konyvek%ROWTYPE;

v_Buffer t_ugyfel_adatok;

BEGIN

OPEN cur_ugyfel_konyvek;

LOOP

FETCH cur_ugyfel_konyvek

BULK COLLECT INTO v_Buffer

LIMIT v_Buffer.LIMIT; -- Ennyi fér bele

EXIT WHEN v_Buffer.COUNT = 0;

FOR i IN 1..v_Buffer.COUNT LOOP

DBMS_OUTPUT.NEW_LINE;

DBMS_OUTPUT.PUT_LINE(v_Buffer(i).nev);

FOR j IN 1..v_Buffer(i).konyvek.COUNT LOOP

DBMS_OUTPUT.PUT_LINE(' '
```

```
|| LPAD(v_Buffer(i).konyvek(j).konyv_id, 3) || ' '

|| TO_CHAR(v_Buffer(i).konyvek(j).datum, 'YYYY-MON-DD'));

END LOOP;

END LOOP;

END LOOP;

CLOSE cur_ugyfel_konyvek;

END;

/

/*

Eredmény:

Kovács János

Szabó Máté István

30 2002-ÁPR. -21

45 2002-ÁPR. -21

50 2002-ÁPR. -21

József István

15 2002-JAN. -22

20 2002-JAN. -22

25 2002-ÁPR. -10

45 2002-ÁPR. -10

50 2002-ÁPR. -10

Tóth László

30 2002-FEBR. -24

Erdei Anita

35 2002-ÁPR. -15

Komor Ágnes

5 2002-ÁPR. -12

10 2002-MÁRC. -12

Jaripekka Hämälainen

35 2002-MÁRC. -18

40 2002-MÁRC. -18

A PL/SQL eljárás sikeresen befejeződött.

*/
```

A BULK COLLECT csak szerveroldali programokban alkalmazható.

A FORALL utasítás tartalmazhat olyan INSERT, DELETE és UPDATE utasítást, amelynek van BULK COLLECT utasításrésze, de nem tartalmazhat ilyen SELECT utasítást. A FORALL működése közben a BULK

COLLECT által visszaadott eredményeket az egyes iterációk a megelőző iteráció eredményei után fűzik (ezzel szemben ha egy FOR ciklusban szerepelne a BULK COLLECT-et tartalmazó utasítás, akkor az minden iterációban felülírná az előző eredményeket).

9. példa

```
/*
Hasonlítsa össze az ugyfel_visszahoz eljárás
előzőleg megadott implementációját a mostanival.
*/
CREATE OR REPLACE PROCEDURE ugyfel_visszahoz(
p_Ugyfelek T_Id_lista
) IS
/* Több ügyfél minden könyvének visszahozatalát adminisztrálja
a függvény. Kiírja, hogy ki hány könyvet hozott vissza. */
v_Konyvek T_Id_lista;
BEGIN
/* A kölcsönzések törlése a törölt könyvek azonosítóinak
összegyűjtése mellett, egy könyv azonosítója többször
is szerepelhet a visszaadott kollekcióban. */
FORALL i IN 1..p_Ugyfelek.COUNT
DELETE FROM kolcsonzes WHERE kolcsonzo = p_Ugyfelek(i)
RETURNING konyv BULK COLLECT INTO v_Konyvek;
/* A %BULK_ROWCOUNT segítségével jelentés készítése */
FOR i IN 1..p_Ugyfelek.COUNT LOOP
DBMS_OUTPUT.PUT_LINE('Ügyfél: ' || p_Ugyfelek(i)
|| ', visszahozott könyvek: ' || SQL%BULK_ROWCOUNT(i));
END LOOP;
/* Könyvek szabad példányainak adminisztrálása */
FORALL i IN 1..v_Konyvek.COUNT
UPDATE konyv SET szabad = szabad + 1
WHERE id = v_Konyvek(i);
/* Az ügyfelek konyvek tábláinak üresre állítása. */
FORALL i IN 1..p_Ugyfelek.COUNT
UPDATE ugyfel SET konyvek = T_Konyvek()
WHERE id = p_Ugyfelek(i);
END ugyfel_visszahoz;
/
```

```
show errors
```

13. fejezet - Triggerek

A *trigger* olyan tevékenységet definiál, amely automatikusan végbemegy, ha egy tábla vagy nézet módosul vagy ha egyéb felhasználói vagy rendszeresemények következnek be. A trigger adatbázis-objektum. A tevékenység kódját megírhatjuk PL/SQL, Java vagy C nyelven. A triggerek működése a felhasználó számára átlátszó módon történik. Egy trigger működését a következő események válthatják ki:

- egy táblán vagy nézeten végrehajtott INSERT, DELETE vagy UPDATE utasítás;
- egyes DDL utasítások;
- szerverhibák;
- felhasználói be- és kijelentkezés;
- adatbázis elindítása és leállítása.

A triggereket elsősorban az alábbi esetekben használjuk:

- származtatott oszlopértékek generálása;
- érvénytelen tranzakciók megelőzése;
- védelem;
- hivatkozási integritási megszorítások definiálása;
- komplex üzleti szabályok kezelése;
- eseménynaplózás;
- követés;
- táblastatisztikák gyűjtése;
- adattöbbszörözés.

1. Triggerek típusai

A triggereket többféle szempont szerint osztályozhatjuk. Meg kell határozni, hogy egy trigger a bekövetkezett eseményhez viszonyítva *mikor* (például előtte) és *hányszor* fusson le, illetve külön kell kezelni bizonyos események triggereit. Vegyük sorra a triggerek típusait:

- sor szintű és utasítás szintű trigger;
- BEFORE és AFTER trigger;
- INSTEAD OF trigger;
- rendszertriggerek.

Sor szintű trigger

Egy sor szintű trigger mindannyiszor lefut, ahányszor a tábla adatai módosulnak. Például egy DELETE utasítás esetén minden törölt sor újból aktiválja a triggert. Ha egyetlen sor sem módosul, a trigger egyszer sem fut le.

Utasítás szintű trigger

Az utasítás szintű trigger egyszer fut le, függetlenül a kezelt sorok számától. Ez a trigger akkor is lefut, ha egyetlen sort sem kezeltünk.

BEFORE és AFTER triggerek

A BEFORE és AFTER triggerek egyaránt lehetnek sor és utasítás szintűek. Csak táblához kapcsoltan hozhatók létre, nézetre nem, ám egy alaptáblához kapcsolt trigger lefut a nézeten végrehajtott DML utasítás esetén is.

DDL utasításhoz kapcsolt trigger is létrehozható, de csak adatbázison és sémán, táblán nem. A BEFORE trigger azelőtt fut le, mielőtt a hozzákapcsolt utasítás lefutna. Az AFTER trigger a hozzákapcsolt utasítás lefutása után fut le.

Ugyanahhoz a táblához, ugyanazon utasításhoz ugyanazon típusból akárhány trigger megadható.

INSTEAD OF trigger

Ez a triggerfajta a hozzákapcsolt utasítás helyett fut le. Az INSTEAD OF trigger csak sor szintű lehet és csak nézeteken definiálható. Akkor használjuk, ha egy nézetet módosítani akarunk, de azt nem tehetjük meg közvetlenül DML utasítások segítségével.

Rendszertriggerek

A triggerek felhasználhatók arra is, hogy adatbázis-eseményekről információkat adjunk az „előfizetőknek”. Az alkalmazások feliratkozhatnak az adatbázis-események, illetve más alkalmazások üzeneteinek előfizetői listájára, és akkor ezeket automatikusan megkapják. Az adatbázis-események a következők:

• rendszeresemények:

– adatbázis elindítása és leállítása,

– szerverhiba;

• felhasználói események:

– bejelentkezés és kijelentkezés,

– DDL utasítás (CREATE, ALTER, DROP) kiadása,

– DML utasítás (INSERT, DELETE, UPDATE) kiadása.

A rendszereseményekhez és a felhasználói be- és kijelentkezéshez, illetve a DDL-utasításokhoz kapcsolt triggerek séma vagy adatbázis szinten hozhatók létre.

A DML utasításokhoz kapcsolt triggerek táblákon és nézeteken definiálhatók.

Az események publikálásához az Oracle Advanced Queuing mechanizmusa használható. A triggerek a DBMS_AQ csomag eszközeire hivatkozhatnak (*lásd* [18]).

2. Trigger létrehozása

Egy trigger létrehozásához saját sémában a CREATE TRIGGER, másik felhasználó sémájában a CREATE ANY TRIGGER, az adatbázison létrehozandóhoz pedig az ADMINISTER DATABASE TRIGGER jogosultság szükséges. A létrehozó utasítás formája:

```
CREATE [OR REPLACE] TRIGGER [séma.]triggernév

{BEFORE|AFTER|INSTEAD OF}

{dml_trigger|{ddl_esemény [OR ddl_esemény]…|

ab_esemény [OR ab_esemény]…}

ON {DATEBASE|[séma.]SCHEMA}

[WHEN (feltétel)] {plsql_blokk|eljáráshívás}
```

ahol

```
dml_trigger::=

{INSERT|DELETE|UPDATE [OF oszlop [,oszlop]…}

[OR {INSERT|DELETE|UPDATE [OF oszlop [,oszlop]…}]…

ON {[séma.]tábla |

[NESTED TABLE bát_oszlop OF] [séma.]nézet}

[REFERENCING {OLD [AS] régi | NEW [AS] új | PARENT [AS] szülő}

[{OLD [AS] régi | NEW [AS] új | PARENT [AS] szülő}]…

[FOR EACH ROW]
```

Az OR REPLACE esetén a már létező trigger újradefiniálása történik, annak előzetes megszüntetése nélkül. A *séma* a triggert tartalmazó séma neve. Ha hiányzik, a parancsot kiadó felhasználó sémájában jön létre a trigger.

A *triggernév* a most létrehozandó trigger neve lesz.

A BEFORE, AFTER, INSTEAD OF a trigger típusát adja meg. BEFORE és AFTER trigger csak táblán, INSTEAD OF csak nézeten hozható létre.

Az INSERT, DELETE, UPDATE definiálja azt az SQL utasítást, amelynek hatására a trigger lefut. UPDATE trigger esetén ha megadunk oszlopokat az OF kulcsszó után, akkor a trigger csak az olyan UPDATE utasítás hatására fut le, amely SET utasításrészében legalább az egyik megadott oszlop szerepel.

Az ON utasításrész azt az adatbázis-objektumot adja meg, amelyen a triggert létrehozzuk. Ez a megadott *séma* (vagy a létrehozó sémája) *tábla, nézet* vagy *beágyazott tábla típusú oszlop* (*bát_oszlop*) lehet.

A REFERENCING utasításrész korrelációs neveket *(régi, új, szülő)* határoz meg. Ezek a trigger törzsében és a WHEN utasításrészben használhatók sorszintű trigger esetén. Az OLD az aktuális sor módosítása előtti, a NEW a módosítása utáni neveket adja meg. Alapértelmezett korrelációs nevek :OLD és :NEW. Beágyazott tábla esetén az OLD és a NEW a beágyazott tábla sorait, a PARENT a szülő tábla aktuális sorát adja. Objektumtábla és objektumnézet esetén az OLD és NEW az objektumpéldányt hivatkozza.

A FOR EACH ROW sor szintű triggert hoz létre. Ha nem adjuk meg, akkor az INSTEAD OF trigger (amelynél ez az alapértelmezés) kivételével utasítás szintű trigger jön létre. A *ddl_esemény* egy olyan DDL utasítást, az *ab_esemény* egy olyan adatbázis-eseményt határoz meg, amelyek a triggert aktiválják. Ezek az adatbázishoz (DATABASE) vagy a *séma* nevű (ennek hiányában a saját) sémához (SCHEMA) köthetők. BEFORE vagy AFTER triggerek esetén adhatók meg.

A *ddl_esemény* a következők valamelyike lehet: ALTER, ANALYZE, ASSOCIATE STATISTICS, AUDIT, COMMENT, CREATE, DISASSOCIATE STATISTICS, DROP, GRANT, NOAUDIT, RENAME, REVOKE, TRUNCATE. Megadásuk esetén az általuk megnevezett parancs végrehajtása jelenti a triggert aktivizáló eseményt.

Ha azt akarjuk, hogy a fenti parancsok mindegyikére reagáljon a trigger, akkor adjuk meg a DDL opciót.

Az ALTER DATABASE, CREATE DATABASE és CREATE CONTROLFILE parancsok nem aktivizálják a triggert.

Az *ab_esemény* az alábbi opciókkal rendelkezik: SERVERERROR: egy szerverhiba bekövetkezte aktiválja a triggert. A következő hibák esetén a trigger nem fog lefutni:

- ORA-01403: nem talált adatot,
- ORA-01422: a pontos lehívás (FETCH) a kívántnál több sorral tér vissza,
- ORA-01423: hiba a többlet sorok ellenőrzésénél a pontos lehívásban (FETCH),
- ORA-01034: az ORACLE nem érhető el,

- ORA-04030: a feldolgozás kifutott a memóriából (,) lefoglalása közben.

LOGON: egy kliensalkalmazás bejelentkezik az adatbázisba.

LOGOFF: egy kliensalkalmazás kilép az adatbázisból.

STARTUP: az adatbázis megnyitásra kerül.

SHUTDOWN: az adatbázis leállításra kerül.

SUSPEND: szerverhiba miatt egy tranzakció működése felfüggesztődik.

AFTER trigger esetén csak a LOGON, STARTUP, SERVERERROR, SUSPEND; BEFORE triggernél a LOGOFF és SHUTDOWN megadása lehetséges. Az AFTER STARTUP és a BEFORE SHUTDOWN triggerek csak adatbázison értelmezhetők.

A trigger csak a WHEN utasításrészben megadott *feltétel* teljesülése esetén fut le. A feltételben nem szerepelhet lekérdezés vagy PL/SQL függvény hívása. INSTEAD OF és utasításszintű trigger esetén nem adható meg.

A *plsql_blokk* vagy az *eljáráshívás* a trigger *törzsét* alkotják. Ez tehát vagy egy név nélküli PL/SQL blokk, vagy egy tárolt eljárás meghívása.

A következőkben példákat adunk különböző triggerekre.

DML triggerek

1. példa

```
/*
A következő trigger segítségével a kölcsönzés
adminisztrációja automatizálható.
Ugyanis egyetlen sor beszúrása a kolcsonzes táblába
maga után vonja az ugyfel és a konyv táblák megfelelő
bejegyzéseinek változtatását.
A trigger sor szintű. Ha nem lehet a kölcsönzést
végrehajtani, kivételt dobunk.
*/
CREATE OR REPLACE TRIGGER tr_insert_kolcsonzes
AFTER INSERT ON kolcsonzes
FOR EACH ROW
DECLARE
v_Ugyfel ugyfel%ROWTYPE;
BEGIN
SELECT * INTO v_Ugyfel
FROM ugyfel WHERE id = :NEW.kolcsonzo;
IF v_Ugyfel.max_konyv = v_Ugyfel.konyvek.COUNT THEN
RAISE_APPLICATION_ERROR(-20010,
v_Ugyfel.nev || ' ügyfél nem kölcsönözhet több könyvet.');
```

```
END IF;
INSERT INTO TABLE(SELECT konyvek FROM ugyfel
WHERE id = :NEW.kolcsonzo)
VALUES (:NEW.konyv, :NEW.datum);
BEGIN
UPDATE konyv SET szabad = szabad -1
WHERE id = :NEW.konyv;
EXCEPTION
WHEN OTHERS THEN
RAISE_APPLICATION_ERROR(-20020,
'Nincs a könyvből több példány.');
END;
END tr_insert_kolcsonzes;
/
show errors
/*
Nézzünk néhány példát.
A példák az inicializált adatbázis adatain futnak.
/*
József István és az 'SQL:1999 ...' könyv
Sajnos az ügyfél nem kölcsönözhet többet.
*/
INSERT INTO kolcsonzes (kolcsonzo, konyv, datum)
VALUES (15, 30, SYSDATE);
/*
INSERT INTO kolcsonzes (kolcsonzo, konyv, datum)
*
Hiba a(z) 1. sorban:
ORA-20010: József István ügyfél nem kölcsönözhet több könyvet.
ORA-06512: a(z) "PLSQL.TR_INSERT_KOLCSONZES", helyen a(z) 8. sornál
ORA-04088: hiba a(z) 'PLSQL.TR_INSERT_KOLCSONZES' trigger futása közben
*/
/*
Komor Ágnes és az 'A critical introduction… ' könyv
Sajnos a könyvből nincs szabad példány.
*/
```

```
INSERT INTO kolcsonzes (kolcsonzo, konyv, datum)
VALUES (30, 35, SYSDATE);
/*
INSERT INTO kolcsonzes (kolcsonzo, konyv, datum)
*
Hiba a(z) 1. sorban:
ORA-20020: Nincs a könyvből több példány.
ORA-06512: a(z) "PLSQL.TR_INSERT_KOLCSONZES", helyen a(z) 21. sornál
ORA-04088: hiba a(z) 'PLSQL.TR_INSERT_KOLCSONZES' trigger futása közben
*/
/*
Komor Ágnes és a 'The Norton Anthology… ' könyv
Minden rendben lesz.
*/
INSERT INTO kolcsonzes (kolcsonzo, konyv, datum)
VALUES (30, 40, SYSDATE);
SELECT * FROM TABLE(SELECT konyvek FROM ugyfel
WHERE id = 30);
/*
1 sor létrejött.
KONYV_ID DATUM
---------- -----------
5 02-ÁPR. 12
10 02-MÁRC. 12
40 02-MÁJ. 12
*/
```

2. példa

```
/*
A következő trigger segítségével naplózzuk
a törölt kölcsönzéseket a kolcsonzes_naplo táblában.
*/
CREATE OR REPLACE TRIGGER tr_kolcsonzes_naploz
AFTER DELETE ON kolcsonzes
REFERENCING OLD AS kolcsonzes
FOR EACH ROW
```

```
DECLARE
v_Konyv konyv%ROWTYPE;
v_Ugyfel ugyfel%ROWTYPE;
BEGIN
SELECT * INTO v_Ugyfel
FROM ugyfel WHERE id = :kolcsonzes.kolcsonzo;
SELECT * INTO v_Konyv
FROM konyv WHERE id = :kolcsonzes.konyv;
INSERT INTO kolcsonzes_naplo VALUES(
v_Konyv.ISBN, v_Konyv.cim,
v_Ugyfel.nev, v_Ugyfel.anyja_neve,
:kolcsonzes.datum, SYSDATE,
:kolcsonzes.megjegyzes);
END tr_kolcsonzes_naploz;
/
show errors
```

3. példa

```
/*
A következő trigger megakadályozza, hogy
egy könyvet 2-nél többször meghosszabítsanak.
Megjegyzés:
- A WHEN feltételében nem kell ':' az OLD,
NEW, PARENT elé.
- Ugyanez a hatás elérhető egy CHECK megszorítással.
*/
CREATE OR REPLACE TRIGGER tr_kolcsonzes_hosszabbit
BEFORE INSERT OR UPDATE ON kolcsonzes
FOR EACH ROW
WHEN (NEW.hosszabbitva > 2 OR NEW.hosszabbitva < 0)
BEGIN
RAISE_APPLICATION_ERROR(-20005,
'Nem megengedett a hosszabbítások száma');
END tr_kolcsonzes_hosszabbit;
/
show errors
```

```
UPDATE kolcsonzes SET hosszabbitva = 10;
/*
UPDATE kolcsonzes SET hosszabbitva = 10
*
Hiba a(z) 1. sorban:
ORA-20005: Nem megengedett a hosszabbítások száma
ORA-06512: a(z) "PLSQL.TR_KOLCSONZES_HOSSZABBIT", helyen a(z) 2. sornál
ORA-04088: hiba a(z) 'PLSQL.TR_KOLCSONZES_HOSSZABBIT' trigger futása közben
*/
```

4. példa

```
/*Egy triggerrel megakadályozzuk azt, hogy valaki
a kolcsonzes_naplo táblából töröljön, vagy
az ott levő adatokat módosítsa.
Nincs szükség sor szintű triggerre, elég
utasítás szintű trigger.
*/
CREATE OR REPLACE TRIGGER tr_kolcsonzes_naplo_del_upd
BEFORE DELETE OR UPDATE ON kolcsonzes_naplo
BEGIN
RAISE_APPLICATION_ERROR(-20100,
'Nem megengedett művelet a kolcsonzes_naplo táblán');
END tr_kolcsonzes_naplo_del_upd;
/
```

5. példa

```
/*A NEW pszeudováltozó értéke megváltoztatható
BEFORE triggerben, és akkor az új érték kerül
be a táblába.
*/
CREATE TABLE szam_tabla(a NUMBER);
CREATE OR REPLACE TRIGGER tr_duplaz
BEFORE INSERT ON szam_tabla
FOR EACH ROW
BEGIN
:NEW.a := :NEW.a * 2;
END tr_duplaz;
```

```
INSERT INTO szam_tabla VALUES(5);

SELECT * FROM szam_tabla;

/*

A

----------

10

*/

DROP TRIGGER tr_duplaz;

DROP TABLE szam_tabla;
```

6. példa

```
/*

Egy könyv azonosítóját kell megváltoztatni.

Ez nem lehetséges automatikusan, mert

a kolcsonzes tábla konyv oszlopa külső

kulcs. Így azt is meg kell változtatni, hogy

az integritási megszorítás ne sérüljön.

A megszorítás a trigger futása után kerül ellenőrzésre.

*/

CREATE OR REPLACE TRIGGER tr_konyv_id

BEFORE UPDATE OF id ON konyv

FOR EACH ROW

BEGIN

-- Módosítjuk a kolcsonzes táblát

UPDATE kolcsonzes SET konyv = :NEW.id

WHERE konyv = :OLD.id;

END tr_konyv_id;

/

show errors;

UPDATE konyv SET id = 6 WHERE id = 5;
```

INSTEAD OF triggerek

1. példa

```
/*

Az INSTEAD OF triggerek lehetővé teszik nézetek módosítását.

Megpróbáljuk módosítani egy ügyfél nevét az ugyfel_konyv

nézet sorain keresztül.
```

```
*/
UPDATE ugyfel_konyv SET ugyfel = 'József István TRIGGERES'
WHERE ugyfel_id = 15;
/*
UPDATE ugyfel_konyv SET ugyfel = 'József István TRIGGERES'
*
Hiba a(z) 1. sorban:
ORA-01779: nem módosítható olyan oszlop, amely egy kulcsot nem megőrző
táblára utal
*/
/*
A következő trigger segítségével egy ügyfél nevét vagy
egy könyv címét módosíthatjuk az ugyfel_konyv nézeten
keresztül. Ha azonosítót is megpróbálnak változtatni,
azzal egyszerűen nem törődünk (nem váltunk ki kivételt).
*/
CREATE OR REPLACE TRIGGER tr_ugyfel_konyv_mod
INSTEAD OF UPDATE ON ugyfel_konyv
FOR EACH ROW
BEGIN
IF :NEW.ugyfel <> :OLD.ugyfel THEN
UPDATE ugyfel SET nev = :NEW.ugyfel
WHERE id = :OLD.ugyfel_id;
END IF;
IF :NEW.konyv <> :OLD.konyv THEN
UPDATE konyv SET cim = :NEW.konyv
WHERE id = :OLD.konyv_id;
END IF;
END tr_ugyfel_konyv_mod;
/
show errors
/*
Megpróbálunk módosítani megint.
*/
UPDATE ugyfel_konyv SET ugyfel = 'József István TRIGGERES'
WHERE ugyfel_id = 15;
```

```
/*
5 sor módosítva.
*/
```

2. példa

```
/*
Az INSTEAD OF triggerek használhatók NESTED TABLE
opcióval, egy nézet kollekcióoszlopának módosítására.
NESTED TABLE előírás csak nézetek esetében használható,
táblák kollekció típusú oszlopaira nem.
Megadunk egy nézetet, amely egy alkérdés eredményét
kollekció típusú oszlopként mutatja.
*/
CREATE VIEW ugyfel_kolcsonzes AS
SELECT u.id, u.nev, CAST( MULTISET( SELECT konyv, datum
FROM kolcsonzes
WHERE kolcsonzo = u.id)
AS T_Konyvek) AS konyvek
FROM ugyfel u;
/*
Töröljük József István 'ECOOP 2001...' kölcsönzését
*/
DELETE FROM TABLE(SELECT konyvek FROM ugyfel_kolcsonzes
WHERE id = 15)
WHERE konyv_id = 25;
/*
DELETE FROM TABLE(SELECT konyvek FROM ugyfel_kolcsonzes
*
Hiba a(z) 1. sorban:
ORA-25015: ezen a beágyazott táblanézet oszlopon nem hajtható végre DML
*/
/*
A következő trigger segítségével a visszahozatal
adminisztrációja automatizálható az ugyfel_kolcsonzes
tábla konyvek oszlopán keresztül.
Ugyanis egyetlen sor törlése a beágyazott táblából
```

```
előidézi a kolcsonzes tábla egy sorának törlését,
az ugyfel tábla konyvek oszlopának módosítását, valamint
a konyv tábla megfelelő bejegyzésének változtatását.
*/
CREATE OR REPLACE TRIGGER tr_ugyfel_kolcsonzes_del
INSTEAD OF DELETE ON NESTED TABLE konyvek OF ugyfel_kolcsonzes
FOR EACH ROW
BEGIN
DELETE FROM TABLE(SELECT konyvek FROM ugyfel
WHERE id = :PARENT.id)
WHERE konyv_id = :OLD.konyv_id
AND datum = :OLD.datum;
DELETE FROM kolcsonzes
WHERE kolcsonzo = :PARENT.id
AND konyv = :OLD.konyv_id
AND datum = :OLD.datum;
UPDATE konyv SET szabad = szabad + 1
WHERE id = :OLD.konyv_id;
END tr_ugyfel_kolcsonzes_del;
/
show errors
/*
Töröljük József István 'ECOOP 2001...' kölcsönzését
*/
DELETE FROM TABLE(SELECT konyvek FROM ugyfel_kolcsonzes
WHERE id = 15)
WHERE konyv_id = 25;
/*
1 sor törölve.
*/
/*
A trigger végrehajtott egy DELETE utasítást a kolcsonzes
táblán. Ellenőrizzük, hogy a korábban létrehozott
tr_kolcsonzes_naploz trigger is lefutott-e?
*/
SELECT * FROM kolcsonzes_naplo;
```

```
/*
KONYV_ISBN KONYV_CIM
---------------------------- ----------------------------
UGYFEL_NEV UGYFEL_ANYJANEVE
---------------------------- ----------------------------
ELVITTE VISSZAHOZTA MEGJEGYZES
---------- ---------- --------------------
ISBN 963 03 9005 1 Java - start!
József István Ábrók Katalin
02-ÁPR. -10 06-JÚN. -23
(Megj.: Az eredmény formátumát kisebb méretűvé szabtuk át
az olvashatóság kedvéért.)
*/
```

3. példa (Korrelációs nevek használatára)

```
/*
Triggereink olvashatóbbak lehetnek, ha az
alapértelmezett NEW, OLD, illetve PARENT nevekre
azok szemantikájának megfelelő névvel hivatkozunk.
Ezt a REFERENCING előírással adhatjuk meg.
Lássuk egy előző példa módosított változatát:
*/
CREATE OR REPLACE TRIGGER tr_ugyfel_kolcsonzes_del
INSTEAD OF DELETE ON NESTED TABLE konyvek OF ugyfel_kolcsonzes
REFERENCING OLD AS v_Kolcsonzes
PARENT AS v_Ugyfel
FOR EACH ROW
BEGIN
DELETE FROM TABLE(SELECT konyvek FROM ugyfel
WHERE id = :v_Ugyfel.id)
WHERE konyv_id = :v_Kolcsonzes.konyv_id
AND datum = :v_Kolcsonzes.datum;
DELETE FROM kolcsonzes
WHERE kolcsonzo = :v_Ugyfel.id
AND konyv = :v_Kolcsonzes.konyv_id
AND datum = :v_Kolcsonzes.datum;
```

```
UPDATE konyv SET szabad = szabad + 1
WHERE id = :v_Kolcsonzes.konyv_id;
END tr_ugyfel_kolcsonzes_del;
/
show errors
/*
Döntse el, melyik forma tetszik jobban,
és használja azt következetesen!
A könyv további részeiben maradunk a
standard nevek mellett.
*/
```

4. példa (Trigger, ahol a törzs egyetlen eljáráshívásból áll)

```
/*
Az előzőek során létrehozott
tr_ugyfel_kolcsonzes_del egy könyv visszahozatalát
adminisztrálta. Ugyanezt a funkciót már megírtuk
egy csomagbeli eljárással.
Célszerűbb lenne azt használni, a kód
újrahasználása végett.
*/
CREATE OR REPLACE TRIGGER tr_ugyfel_kolcsonzes_del
INSTEAD OF DELETE ON NESTED TABLE konyvek OF ugyfel_kolcsonzes
FOR EACH ROW
CALL konyvtar_csomag.visszahoz(:PARENT.id, :OLD.konyv_id)
/
```

DDL trigger a PLSQL sémára

Példa

```
/* Egy csomagváltozóban számláljuk a munkamenetben
végrehajtott sikeres és sikertelen CREATE és DROP
utasításokat.
*/
CREATE OR REPLACE PACKAGE ddl_szamlalo
IS
v_Sikeres_create NUMBER := 0;
v_Sikertelen_create NUMBER := 0;
```

```
v_Sikeres_drop NUMBER := 0;
v_Sikertelen_drop NUMBER := 0;
PROCEDURE kiir;
END ddl_szamlalo;
/
CREATE OR REPLACE PACKAGE BODY ddl_szamlalo
IS
PROCEDURE kiir IS
BEGIN
DBMS_OUTPUT.PUT_LINE(RPAD('v_Sikeres_create: ', 25)
|| v_Sikeres_create);
DBMS_OUTPUT.PUT_LINE(RPAD('v_Sikertelen_create: ', 25)
|| v_Sikertelen_create);
DBMS_OUTPUT.PUT_LINE(RPAD('v_Sikeres_drop: ', 25)
|| v_Sikeres_drop);
DBMS_OUTPUT.PUT_LINE(RPAD('v_Sikertelen_drop: ', 25)
|| v_Sikertelen_drop);
END kiir;
END ddl_szamlalo;
/
CREATE OR REPLACE TRIGGER tr_ddl_szamlalo_bef
BEFORE CREATE OR DROP
ON plsql.SCHEMA
BEGIN
/* Pesszimistán feltételezzük, hogy a
kiváltó utasítás nem lesz sikeres */
IF ORA_SYSEVENT = 'CREATE' THEN
ddl_szamlalo.v_Sikertelen_create :=
ddl_szamlalo.v_Sikertelen_create + 1;
ELSE
ddl_szamlalo.v_Sikertelen_drop :=
ddl_szamlalo.v_Sikertelen_drop + 1;
END IF;
END tr_ddl_szamlalo_bef;
/
CREATE OR REPLACE TRIGGER tr_ddl_szamlalo_aft
```

```
AFTER CREATE OR DROP
ON plsql.SCHEMA
BEGIN
/* Nem kellett volna pesszimistának lenni:) */
IF ORA_SYSEVENT = 'CREATE' THEN
ddl_szamlalo.v_Sikertelen_create :=
ddl_szamlalo.v_Sikertelen_create - 1;
ddl_szamlalo.v_Sikeres_create :=
ddl_szamlalo.v_Sikeres_create + 1;
ELSE
ddl_szamlalo.v_Sikertelen_drop :=
ddl_szamlalo.v_Sikertelen_drop - 1;
ddl_szamlalo.v_Sikeres_drop :=
ddl_szamlalo.v_Sikeres_drop + 1;
END IF;
END tr_ddl_szamlalo_aft;
/
-- Egy sikertelen CREATE
CREATE TRIGGER tr_ddl_szamlalo_aft
/
-- És egy sikeres CREATE
CREATE TABLE abcdefghijklm (a NUMBER)
/
-- És egy sikeres DROP
DROP TABLE abcdefghijklm
/
CALL ddl_szamlalo.kiir();
/*
v_Sikeres_create: 1
v_Sikertelen_create: 1
v_Sikeres_drop: 1
v_Sikertelen_drop: 0
*/
/*
Mi lesz az eredmény, ha újra létrehozzuk
a csomag specifikációját?
```

```
Magyarázza meg az eredményt!
*/
CREATE OR REPLACE PACKAGE ddl_szamlalo
IS
v_Sikeres_create NUMBER := 0;
v_Sikertelen_create NUMBER := 0;
v_Sikeres_drop NUMBER := 0;
v_Sikertelen_drop NUMBER := 0;
-- ez a megjegyzés itt megváltoztatja a csomagot,
-- ezért tényleg újra létre kell hozni
PROCEDURE kiir;
END ddl_szamlalo;
/
CALL ddl_szamlalo.kiir();
/*
v_Sikeres_create: 1
v_Sikertelen_create: -1
v_Sikeres_drop: 0
v_Sikertelen_drop: 0
*/
```

Adatbázistrigger

Példa

```
/*
A DBA naplózza a felhasználók
be- és kijelentkezéseit a következő
tábla és triggerek segítségével.
*/
CREATE TABLE felhasznalok_log (
Felhasznalo VARCHAR2(30),
Esemeny VARCHAR2(30),
Hely VARCHAR2(30),
Idopont TIMESTAMP
)
CREATE OR REPLACE PROCEDURE felhasznalok_log_bejegyez(
p_Esemeny felhasznalok_log.esemeny%TYPE
```

```
) IS
BEGIN
INSERT INTO felhasznalok_log VALUES (
ORA_LOGIN_USER, p_Esemeny,
ORA_CLIENT_IP_ADDRESS, SYSTIMESTAMP
);
END felhasznalok_log_bejegyez;
/
CREATE OR REPLACE TRIGGER tr_felhasznalok_log_be
AFTER LOGON
ON DATABASE
CALL felhasznalok_log_bejegyez('Bejelentkezés')
/
CREATE OR REPLACE TRIGGER tr_felhasznalok_log_ki
BEFORE LOGOFF
ON DATABASE
CALL felhasznalok_log_bejegyez('Kijelentkezés')
/
/*
Próbáljon meg be- és kijelentkezni néhány
felhasználóval, ha teheti különböző kliensekről,
majd ellenőrizze a tábla tartalmát.
*/
```

3. A triggerek működése

Egy triggernek két állapota lehet: *engedélyezett* és *letiltott*. A letiltott trigger nem indul el, ha a kiváltó esemény bekövetkezik. Az engedélyezett trigger esetén az Oracle automatikusan a következő tevékenységeket hajtja végre:

- Lefuttatja a triggert. Ha ugyanarra az utasításra több azonos típusú trigger van definiálva, akkor ezek sorrendje határozatlan.
- Ellenőrzi az integritási megszorításokat és biztosítja, hogy a trigger ne sértse meg azokat.
- Olvasási konzisztenciát biztosít a lekérdezésekhez.
- Kezeli a trigger és a sémaobjektumok közötti függőségeket.
- Osztott adatbázis esetén, ha a trigger távoli táblát módosított, kétfázisú véglegesítést alkalmaz.

A CREATE utasítás automatikusan engedélyezi a triggert. Triggert letiltani és engedélyezni az ALTER TRIGGER (*lásd* később ebben a fejezetben) és az ALTER TABLE (*lásd* [8]) utasítással lehet.

A DML triggerek futtatási konzisztenciájának biztosítása érdekében az Oracle a következő végrehajtási modellt követi, ha ugyanazon utasításon különböző típusú triggerek vannak értelmezve:

1. Végrehajtja az összes utasításszintű BEFORE triggert.

2. A DML utasítás által érintett minden sorra ciklikusan:

a) végrehajtja a sorszintű BEFORE triggereket;

b) zárolja és megváltoztatja a sort és ellenőrzi az integritási megszorításokat. A zár csak a tranzakció végeztével oldódik;

c) végrehajtja a sorszintű AFTER triggereket.

3. Ellenőrzi a késleltetett integritási megszorításokat.

4. Végrehajtja az utasítás szintű AFTER triggereket.

A végrehajtási modell rekurzív. Egy trigger működése közben újabb triggerek indulhatnak el, azok végrehajtása ugyanezt a modellt követi. A modell igen lényeges tulajdonsága, hogy az összes tevékenység és ellenőrzés befolyásolja a DML utasítás sikerességét. Ha egy trigger futása közben kivétel következik be, és azt nem kezeljük le, akkor az összes tevékenység (az SQL utasítás és a trigger hatását is beleértve) visszagörgetésre kerül. Így egy trigger működése nem sérthet integritási megszorítást.

1. példa

```
/*

A következő példa demonstrálja

a triggerek végrehajtási sorrendjét.

*/

CREATE TABLE tabla (a NUMBER);

DELETE FROM tabla;

INSERT INTO tabla VALUES(1);

INSERT INTO tabla VALUES(2);

INSERT INTO tabla VALUES(3);

CREATE TABLE tabla_log(s VARCHAR2(30));

DELETE FROM tabla_log;

CREATE OR REPLACE PROCEDURE tabla_insert(

p tabla_log.s%TYPE

) IS

BEGIN

INSERT INTO tabla_log VALUES(p);

END tabla_insert;

/

CREATE OR REPLACE TRIGGER tr_utasitas_before

BEFORE INSERT OR UPDATE OR DELETE ON tabla

CALL tabla_insert('UTASITAS BEFORE')
```

```
/
CREATE OR REPLACE TRIGGER tr_utasitas_after
AFTER INSERT OR UPDATE OR DELETE ON tabla
CALL tabla_insert('UTASITAS AFTER')
/
CREATE OR REPLACE TRIGGER tr_sor_before
BEFORE INSERT OR UPDATE OR DELETE ON tabla
FOR EACH ROW
CALL tabla_insert('sor before ' || :OLD.a || ', ' || :NEW.a)
/
CREATE OR REPLACE TRIGGER tr_sor_after
AFTER INSERT OR UPDATE OR DELETE ON tabla
FOR EACH ROW
CALL tabla_insert('sor after ' || :OLD.a || ', ' || :NEW.a)
/
UPDATE tabla SET a = a+10;
SELECT * FROM tabla_log;
/*
S
------------------------------
UTASITAS BEFORE
sor before 1, 11
sor after 1, 11
sor before 2, 12
sor after 2, 12
sor before 3, 13
sor after 3, 13
UTASITAS AFTER
*/
```

2. példa

```
/*
Rekurzív triggerek.
*/
CREATE TABLE tab_1 (a NUMBER);
INSERT INTO tab_1 VALUES(1);
```

```
CREATE TABLE tab_2 (a NUMBER);

INSERT INTO tab_2 VALUES(1);

CREATE OR REPLACE TRIGGER tr_tab1

BEFORE DELETE ON tab_1

BEGIN

DELETE FROM tab_2;

END tr_tab1;

/

CREATE OR REPLACE TRIGGER tr_tab2

BEFORE DELETE ON tab_2

BEGIN

DELETE FROM tab_1;

END tr_tab2;

/

DELETE FROM tab_1;

/*

Hiba a(z) 1. sorban:

ORA-00036: a rekurzív SQL szintek maximális számának (50) túllépése

...

*/

/*

Mi lesz az eredmény sorszintű triggerek esetén?

*/
```

4. A trigger törzse

A trigger törzse állhat egyetlen CALL utasításból, amely egy tárolt eljárást hív meg. Ez az eljárás lehet PL/SQL eljárás, de lehet egy C vagy Java nyelven megírt eljárás is.

Másik lehetőség, hogy a trigger törzse egy PL/SQL blokk. Minden, amit a blokkról írtunk, itt is igaz, de van néhány korlátozás. A trigger törzsét alkotó blokkban nem lehetnek tranzakcióvezérlő utasítások (COMMIT, ROLLBACK, SAVEPOINT, SET TRANSACTION). Természetesen a törzsből hívott alprogramok sem tartalmazhatják ezeket az utasításokat. A PL/SQL fordító megengedi ezeket, hibát csak a trigger futása közben okoznak.

Olyan triggerben, amelynek törzse autonóm tranzakciót tartalmaz, használhatók a tranzakcióvezérlő utasítások.

Autonóm tranzakciót tartalmazó kölcsönösen rekurzív triggerek használata az erőforrások zárolása miatt holtpontot eredményezhet.

1. példa

```
/*

Holtpont az AUTONOMOUS_TRANSACTION miatt rekurzív triggerekben.
```

```
*/
CREATE TABLE tab_1 (a NUMBER);
INSERT INTO tab_1 VALUES(1);
CREATE TABLE tab_2 (a NUMBER);
INSERT INTO tab_2 VALUES(1);
CREATE OR REPLACE TRIGGER tr_tab1
AFTER DELETE ON tab_1
FOR EACH ROW
DECLARE
PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
DELETE FROM tab_2;
END tr_tab1;
/
CREATE OR REPLACE TRIGGER tr_tab2
AFTER DELETE ON tab_2
FOR EACH ROW
DECLARE
PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
DELETE FROM tab_1;
END tr_tab2;
/
DELETE FROM tab_1;
/*
Eredmény:
DELETE FROM tab_1
*
Hiba a(z) 1. sorban:
ORA-00060: erőforrásra várakozás közben holtpont jött létre
ORA-06512: a(z) "PLSQL.TR_TAB2", helyen a(z) 4. sornál
ORA-04088: hiba a(z) 'PLSQL.TR_TAB2' trigger futása közben
ORA-06512: a(z) "PLSQL.TR_TAB1", helyen a(z) 4. sornál
ORA-04088: hiba a(z) 'PLSQL.TR_TAB1' trigger futása közben
*/
```

A DML triggerek törzsében használható három paraméter nélküli logikai függvény, amelyek a triggert aktivizáló utasításról adnak információt (miután egy triggert több DML utasítás is aktivizálhat). Ezek a függvények az INSERTING, DELETING, UPDATING függvények, amelyek rendre akkor térnek vissza igaz értékkel, ha az aktivizáló utasítás az INSERT, DELETE, UPDATE volt.

2. példa

```
/*

A szükséges táblák és az eljárás az előző példában voltak definiálva.

*/

CREATE OR REPLACE TRIGGER tr_utasitas_before

BEFORE INSERT OR UPDATE OR DELETE ON tabla

BEGIN

tabla_insert('UTASITAS BEFORE '

|| CASE

WHEN INSERTING THEN 'INSERT'

WHEN UPDATING THEN 'UPDATE'

WHEN DELETING THEN 'DELETE'

END);

END tr_utasitas_before;

/

CREATE OR REPLACE TRIGGER tr_utasitas_after

AFTER INSERT OR UPDATE OR DELETE ON tabla

BEGIN

tabla_insert('UTASITAS AFTER '

|| CASE

WHEN INSERTING THEN 'INSERT'

WHEN UPDATING THEN 'UPDATE'

WHEN DELETING THEN 'DELETE'

END);

END tr_utasitas_after;

/

show errors;

CREATE OR REPLACE TRIGGER tr_sor_before

BEFORE INSERT OR UPDATE OR DELETE ON tabla

FOR EACH ROW

BEGIN

tabla_insert('sor before ' || :OLD.a || ', ' || :NEW.a

|| ' ' || CASE
```

```
WHEN INSERTING THEN 'insert'
WHEN UPDATING THEN 'update'
WHEN DELETING THEN 'delete'
END);
END tr_sor_before;
/
show errors;
CREATE OR REPLACE TRIGGER tr_sor_after
AFTER INSERT OR UPDATE OR DELETE ON tabla
FOR EACH ROW
BEGIN
tabla_insert('sor after ' || :OLD.a || ', ' || :NEW.a
|| ' ' || CASE
WHEN INSERTING THEN 'insert'
WHEN UPDATING THEN 'update'
WHEN DELETING THEN 'delete'
END);
END tr_sor_after;
/
DELETE FROM tabla;
DELETE FROM tabla_log;
-- Biztosan létezik 2 tábla a sémában. */
INSERT INTO tabla
(SELECT ROWNUM FROM tab WHERE ROWNUM <= 2);
UPDATE tabla SET a = a+10;
DELETE FROM tabla;
SELECT * FROM tabla_log;
/*
S
------------------------------
UTASITAS BEFORE INSERT
sor before , 1 insert
sor after , 1 insert
sor before , 2 insert
sor after , 2 insert
UTASITAS AFTER INSERT
```

```
UTASITAS BEFORE UPDATE

sor before 1, 11 update

sor after 1, 11 update

sor before 2, 12 update

sor after 2, 12 update

UTASITAS AFTER UPDATE

UTASITAS BEFORE DELETE

sor before 11, delete

sor after 11, delete

sor before 12, delete

sor after 12, delete

UTASITAS AFTER DELETE

*/
```

Rendszertrigger létrhozásához ADMINISTER DATABASE TRIGGER rendszerjogosultság szükséges.

A rendszertriggerek törzsében az ún. eseményattribútum függvények használhatók, melyek a bekövetkezett rendszer- és felhasználói események jellemzőit adják meg. Ezek a függvények ugyan bármely PL/SQL blokkból meghívhatók, de valós eredményt csak egy trigger törzsében szolgáltatnak. Ezek a függvények a SYS tulajdonában lévő függvények, nyilvános szinonimáik *ora_* prefixszel kezdődnek. A szinonimák ismertetését a 13.1. táblázat tartalmazza.

13.1. táblázat - Eseményattribútum függvények

Szinonima	Típus	Leírás	Példa

ora_client_ip_address	VARCHAR2	TCP/IP protokoll esetén megadja a kliens IP-címét LOGON esemény bekövetkeztekor.	IF ora_sysevent = 'LOGON' THEN
			cim := ora_client_ip_address;
			END IF;

ora_database_name	VARCHAR2(50)	Az adatbázis nevét adja meg.	DECLARE
			db_nev VARCHAR2(50);
			BEGIN
			db_nev := ora_database_name;
			END;

ora_des_encrypted_password	VARCHAR2	Az éppen létrehozás vagy módosítás alatt álló felhasználó jelszava DES titkosítással.	IF ora_dict_obj_type = 'USER' THEN INSERT INTO esemenyek VALUES

			(ora_des_encrypted_password); END IF;
ora_dict_obj_name	VARCHAR2(30)	Azon adatszótárbeli objektum neve, amelyen a DDL művelet végrehajtódik.	INSERT INTO esemenyek VALUES ('Változtatott objektum: ' || ora_dict_obj_name);
ora_dict_obj_name_list (nev_lista OUT ora_name_list_t)	BINARY_INTEGER	Az éppen módosítás alatt álló objektumok számát és neveinek listáját határozza meg.	IF ora_sysevent = 'ASSOCIATE STATISTICS' THEN valt_obj_szama := ora_dict_obj_name_list (nev_lista); END IF;
ora_dict_obj_owner	VARCHAR2(30)	Azon adatszótárbeli objektum tulajdonosa, amelyen a DDL művelet végrehajtódik.	INSERT INTO esemenyek VALUES ('Objektum tulajdonosa: ' || ora_dict_obj_owner);
ora_dict_obj_owner_list (tulaj_lista OUT ora_name_list_t)	BINARY_INTEGER	Az éppen módosítás alatt álló objektumok számát és tulajdonosaik nevének listáját határozza meg.	IF ora_sysevent = 'ASSOCIATE STATISTICS' THEN valt_obj_szama := ora_dict_obj_owner_list(tulaj_lista); END IF;
ora_dict_obj_type	VARCHAR(20)	Azon adatszótárbeli objektum típusa, amelyen a DDL művelet végrehajtódik.	INSERT INTO esemenyek VALUES ('Objektum típusa: ' || ora_dict_obj_type);
ora_grantee(felh_lista OUT ora_name_list_t)	BINARY_INTEGER	A feljogosítottak számát és listáját határozza meg.	IF ora_sysevent = 'GRANT' THEN felh_szama := ora_grantee(felh_lista); END IF;

ora_instance_num	NUMBER	Az adatbázispéldány számát adja meg.	IF ora_instance_num = 1 THEN
			INSERT INTO esemenyek VALUES
			('1');
			END IF;

ora_is_alter_column(
oszlop_nev
IN VARCHAR2)
	BOOLEAN	Igazzal tér vissza, ha a megadott oszlop megváltozott.	IF ora_sysevent = 'ALTER' AND
			ora_dict_obj_type = 'TABLE'
			THEN oszlop_valtozott :=
			ora_is_alter_column('FOO');
			END IF;

ora_is_creating_nested_table	BOOLEAN	Igazzal tér vissza, ha beágyazott tábla jött létre.	IF ora_sysevent = 'CREATE' AND ora_dict_obj_type = 'TABLE' AND ora_is_creating_nested_table THEN INSERT INTO esemenyek VALUES ('Egy beágyazott tábla jött létre.'); END IF;

ora_is_drop_column(
oszlop_nev IN
VARCHAR2)
	BOOLEAN	Igazzal tér vissza, ha a megadott oszlop törlődött.	IF ora_sysevent = 'ALTER'
			AND ora_dict_obj_type =
			'TABLE'
			THEN
			drop_column :=
			ora_is_drop_column('FOO');
			END IF;

ora_is_servererror(hiba_szam IN NUMBER)	BOOLEAN	Igazzal tér vissza, ha a megadott hiba a hibaveremben van.	IF ora_is_servererror(hiba_szam) THEN INSERT INTO esemenyek VALUES

			('Szerverhiba!!'); END IF;
ora_login_user	VARCHAR2(30)	A bejelentkezett felhasználó nevét adja meg.	SELECT ora_login_user FROM dual;
ora_partition_pos	BINARY_INTEGER	Egy CREATE TABLE-re vonatkozó INSTEAD OF triggerben megadja, hogy a táblát létrehozó utasítás mely pozícióján lehet PARTITION utasításrészt elhelyezni.	-- Már elmentettük az -- ora_sql_txt értékét az -- sql_text változóba. n := ora_partition_pos; uj_utasitas := substr(sql_text, 1, n-1) || ' ' || sajat_part_resz || ' ' || substr(sql_text, n));

ora_privileges(
privilegium_lista
OUT
ora_name_list_t)
	BINARY_INTEGER	Az éppen kapott vagy visszavont jogosultságok számát és listáját határozza meg.	IF ora_sysevent = 'GRANT'
			OR ora_sysevent = 'REVOKE'
			THEN
			priv_szama :=
			ora_privileges(priv_lista);
			END IF;

ora_revokee(felh_list
OUT
ora_name_list_t)
	BINARY_INTEGER	Azon felhasználók számát és neveinek listáját határozza meg, akiktől jogosultságokat vontak vissza.	IF (ora_sysevent =
			'REVOKE') THEN felh_szama
			:= ora_revokee(felh_list);
			END IF;

ora_server_error(pozicio IN NUMBER)	NUMBER	A hibaverem megadott pozícióján elhelyezett hibaszámot adja vissza. A verem tetejének pozíciója 1.	INSERT INTO esemenyek VALUES ('A legutolsó hiba száma: ' || ora_server_error(1));
ora_server_error_depth	BINARY_INTEGER	A hibaveremben levő összes hiba darabszámát	n := ora_server_error_depth;

		adja vissza.	
ora_server_error_msg(pozicio IN BINARY_INTEGER)	VARCHAR2	A hibaverem megadott pozícióján elhelyezett hibához tartozó üzenetet adja vissza. A verem tetejének pozíciója 1.	INSERT INTO esemenyek VALUES ('A legutolsó hiba: ' || ora_server_error_msg(1));
ora_server_error_num_ params(pozicio IN BINARY_INTEGER)	BINARY_IN TEGER	A hibaverem megadott pozícióján elhelyezett hibához tartozó üzenetben lecserélt "%s" formájú hibaparaméterek számát adja meg.	n := ora_server_error_num_params (1);
ora_server_error_param(pozicio IN BINARY_INTEGER, param IN BINARY_INTEGER)	VARCHAR2	A hibaverem megadott pozícióján elhelyezett hibához tartozó üzenetben lecserélt "%s", "%d" formájú hibaparaméterek számát adja meg.	param := ora_server_error_param(1, 2);
ora_sql_txt(sql_text OUT ora_name_list_t)	BINARY_IN TEGER	Meghatározza a triggert aktivizáló utasítás szövegét. Ha az utasítás túl hosszú, akkor több táblasorra tördelődik. A visszatérési érték a tábla sorainak száma.	DECLARE sql_text ora_name_list_t; stmt VARCHAR2(2000); ... n := ora_sql_txt(sql_text); FOR i IN 1..n LOOP stmt := stmt || sql_text(i); END LOOP; INSERT INTO esemenyek VALUES ('Az aktiváló

			utasítás: ' || stmt); ...
ora_sysevent	VARCHAR2(20)	A triggert aktivizáló redszeresemény.	INSERT INTO esemenyek VALUES (ora_sysevent);
ora_with_grant_option	BOOLEAN	Igazzal tér vissza, ha a GRANT utasításban szerepel a WITH GRANT OPTION utasításrész.	IF ora_sysevent = 'GRANT' AND ora_with_grant_option THEN INSERT INTO esemenyek VALUES ('WITH GRANT OPTION'); END IF;
space_error_info(hiba_szam OUT NUMBER, hiba_tipus OUT VARCHAR2, objektum_tulaj OUT VARCHAR2, tablespace_nev OUT VARCHAR2, objektum_nev OUT VARCHAR2, alobjektum_nev OUT VARCHAR2)	BOOLEAN	Igazzal tér vissza, ha valamely objektum számára elfogyott a hely. Az OUT paraméterek adnak információt az objektumról	IF space_error_info(hsz, htip, tul, ts, obj, alobj) THEN DBMS_OUTPUT.PUT_LINE('Elfogyott a hely a(z) ' || obj || ' objektum számára, melynek tulajdonosa ' || tul); END IF;

5. Triggerek tárolása

Az Oracle a triggert lefordítva, p-kódban tárolja az adatszótárban. A triggerekkel kapcsolatos információk az adatszótárnézetekben érhetők el. A USER_TRIGGERS nézet tartalmazza a törzs és a WHEN feltétel kódját, a táblát és a trigger típusát. Az ALL_TRIGGERS az aktuális felhasználó triggereiről, a DBA_TRIGGERS minden triggerről tartatalmaz információkat.

A trigger és más objektumok közötti függőségekre is igazak a 10. fejezetben leírtak. Egy érvénytelen trigger explicit módon fordítható újra, vagy pedig a következő aktivizálása során automatikusan újrafordítódik.

1. példa

```
/*
```

```
Emlékezzünk vissza a tr_ugyfel_kolcsonzes_del

trigger legutóbbi definíciójára;

CREATE OR REPLACE TRIGGER tr_ugyfel_kolcsonzes_del

INSTEAD OF DELETE ON NESTED TABLE konyvek OF ugyfel_kolcsonzes

FOR EACH ROW

CALL konyvtar_csomag.visszahoz(:PARENT.id, :OLD.konyv_id)

/

*/

/*

Fordítsuk újra a csomagot, amiben az eljárás,

a trigger törzse van.

*/

ALTER PACKAGE konyvtar_csomag COMPILE;

/*

A következő DELETE hatására újra lefordul a trigger.

*/

DELETE FROM TABLE(SELECT konyvek FROM ugyfel_kolcsonzes

WHERE id = 15)

WHERE konyv_id = 15;
```

Mint az adatbázis-objektumokra általában, a triggerre is alkalmazható az ALTER és DROP utasítás. Az ALTER alakja:

```
ALTER TRIGGER [séma.]triggernév

{ENABLE | DISABLE | RENAME TO új_név|

COMPILE [DEBUG] [REUSE SETTINGS]];
```

A *séma* megadja azt a sémát, ahol a trigger van (ha nincs megadva, akkor feltételezi a saját sémát). A *triggernév* a trigger nevét határozza meg.

Az ENABLE engedélyezi, a DISABLE letiltja a triggert. A RENAME átnevezi azt *új_név*-re.

A COMPILE újrafordítja a triggert, függetlenül attól, hogy érvénytelen-e vagy sem.

A DEBUG arra utasítja a PL/SQL fordítót, hogy kódgenerálás közben használja a PL/SQL nyomkövetőjét.

Az Oracle először újrafordít minden olyan objektumot, amelytől a trigger függ, ha azok érvénytelenek. Ezután fordítja újra a triggert. Az újrafordítás közben töröl minden fordítási kapcsolót, ezeket újraszármaztatja a munkamenetből, majd tárolja őket a fordítás végén. Ezt elkerülendő adjuk meg a REUSE SETTINGS utasításrészt.

Egy trigger törlése az adatszótárból a következő utasításal történik:

```
DROP TRIGGER triggernév;
```

2. példa

```
/*
```

```
Letiltjuk azt a triggert, amelyik lehetővé teszi

az ugyfel_kolcsonzes nézet konyvek oszlopának módosítását.

*/

ALTER TRIGGER tr_ugyfel_kolcsonzes_del DISABLE;

DELETE FROM TABLE(SELECT konyvek FROM ugyfel_kolcsonzes

WHERE id = 15)

WHERE konyv_id = 45;

/*

DELETE FROM TABLE(SELECT konyvek FROM ugyfel_kolcsonzes

*

Hiba a(z) 1. sorban:

ORA-25015: ezen a beágyazott táblanézet oszlopon nem hajtható végre DML

*/

-- Újra engedélyezzük a triggert.

ALTER TRIGGER tr_ugyfel_kolcsonzes_del ENABLE;

-- Explicit módon újrafordítjuk a régi beállításokkal

ALTER TRIGGER tr_ugyfel_kolcsonzes_del COMPILE REUSE SETTINGS;

-- Át is nevezhetjük:

ALTER TRIGGER tr_ugyfel_kolcsonzes_del RENAME TO tr_ugyf_kolcs_del;
```

6. Módosítás alatt álló táblák

Vannak bizonyos megszorítások arra vonatkozóan, hogy egy trigger törzsében mely táblák mely oszlopai érhetők el. Egy DML utasítás által éppen változtatott táblát *módosítás alatt álló táblának* hívunk. A DML utasításhoz rendelt trigger ezen a táblán van definiálva. Az adott táblához a DELETE CASCADE hivatkozási integritási megszorítás által hozzárendelt táblákat is módosítás alatt álló tábláknak tekintjük. Azokat a táblákat, amelyeket egy adott hivatkozási integritási megszorítás ellenőrzéséhez olvasni kell, *megszorítással kapcsolt táblának* hívjuk.

A trigger törzsében elhelyezett SQL utasítás nem olvashatja és módosíthatja a triggert aktivizáló utasítás által éppen módosítás alatt álló táblákat.

Ezek a megszorítások sor szintű triggerekre mindig érvényesek, utasítás szintűekre csak akkor, ha a trigger egy DELETE CASCADE eredményeként aktivizálódott.

Az INSERT INTO … SELECT utasítás kivételével az INSERT utasítás abban az esetben, ha csak egyetlen sort szúr be, a bővítendő táblát nem tekinti módosítás alatt állónak.

A következő példában egy fát tárolunk hierarchikusan az adatbázisban. Minden elem tartalmazza a szülő elem azonosítóját. A gyökér elemeket tartalmazó sorokban a szulo oszlop értéke NULL.

Példa

```
CREATE TABLE fa (

id NUMBER PRIMARY KEY,

szulo NUMBER,
```

```
adat NUMBER,
CONSTRAINT fa_fk FOREIGN KEY (szulo)
REFERENCES fa(id)
);
INSERT INTO fa VALUES (1, NULL, 10);
INSERT INTO fa VALUES (2, 1, 20);
INSERT INTO fa VALUES (3, 2, 30);
INSERT INTO fa VALUES (4, 2, 40);
INSERT INTO fa VALUES (5, 3, 50);
INSERT INTO fa VALUES (6, 1, 60);
INSERT INTO fa VALUES (7, 1, 70);
INSERT INTO fa VALUES (8, NULL, 80);
INSERT INTO fa VALUES (9, 8, 90);
SELECT LPAD(' ', (LEVEL-1)*3, '| ') || '+--'
|| '(' || id || ', ' || adat || ')' AS elem
FROM fa
CONNECT BY PRIOR id = szulo
START WITH szulo IS NULL;
/*
ELEM
---------------------
+--(1, 10)
| +--(2, 20)
| | +--(3, 30)
| | | +--(5, 50)
| | +--(4, 40)
| +--(6, 60)
| +--(7, 70)
+--(8, 80)
| +--(9, 90)
9 sor kijelölve.
*/
```

Készítsünk triggert, ami lehetővé teszi DML segítségével elemek törlését a fából úgy, hogy minden törölt elem gyerekeinek szülőjét átállítja a törölt elem szülőjére. Ha a DML azt a szülőt is törli, akkor annak a szülőjére stb.

A következő trigger logikus megoldás lenne, használata mégsem megengedett, mert egy módosítás alatt álló táblát nem módosíthatunk.

```
CREATE OR REPLACE TRIGGER tr_fa
BEFORE DELETE ON fa
FOR EACH ROW
BEGIN
/* Ezt kellene csinálni, ha lehetne: */
UPDATE fa SET szulo = :OLD.szulo
WHERE szulo = :OLD.id;
END;
/
show errors
DELETE FROM fa WHERE id = 2;
/*
DELETE FROM fa WHERE id = 2;
*
Hiba a(z) 1. sorban:
ORA-04091: PLSQL.FA tábla változtatás alatt áll, trigger/funkció számára
nem látható
ORA-06512: a(z) "PLSQL.TR_FA", helyen a(z) 3. sornál
ORA-04088: hiba a(z) 'PLSQL.TR_FA' trigger futása közben
*/
```

A megoldás megkerüli ezt a megszorítást:

```
/*
Egy lehetséges megoldás.
Úgy törlünk, hogy az azonosítót -1-re módosítjuk,
triggereken keresztül pedig elvégezzük a tényleges
törlést.
Szükségünk van egy temporális táblára,
ezt egy csomagban tároljuk majd.
Első menetben ebben összegyűjtjük a törlendő elemeket.
Második menetben módosítjuk a gyerekeket,
aztán végül elvégezzük a tényleges törlést.
*/
CREATE OR REPLACE PACKAGE pkg_fa
IS
TYPE t_torlendo IS TABLE OF fa%ROWTYPE
```

```
INDEX BY BINARY_INTEGER;
v_Torlendo t_torlendo;
v_Torles BOOLEAN := FALSE;
/* Megadja egy törlésre kerülő elem
végső szülőjét, hiszen az ő szülőjét is
lehet, hogy törlik. */
FUNCTION szulo_torles_utan(p_Id NUMBER)
RETURN NUMBER;
END pkg_fa;
/
CREATE OR REPLACE PACKAGE BODY pkg_fa
IS
FUNCTION szulo_torles_utan(p_Id NUMBER)
RETURN NUMBER IS
v_Id NUMBER := p_Id;
v_Szulo NUMBER;
BEGIN
WHILE v_Torlendo.EXISTS(v_Id) LOOP
v_Id := v_Torlendo(v_Id).szulo;
END LOOP;
RETURN v_Id;
END szulo_torles_utan;
END pkg_fa;
/
-- utasítás szintű BEFORE trigger inicializálja
-- a csomagváltozót
CREATE OR REPLACE TRIGGER tr_fa1
BEFORE UPDATE OF id ON fa
BEGIN
IF NOT pkg_fa.v_Torles THEN
pkg_fa.v_Torlendo.DELETE;
END IF;
END tr_fa1;
/
-- sor szintű trigger tárolja a törlendő elemeket
CREATE OR REPLACE TRIGGER tr_fa
```

```
BEFORE UPDATE OF id ON fa
FOR EACH ROW
WHEN (NEW.id = -1)
BEGIN
IF NOT pkg_fa.v_Torles
THEN
pkg_fa.v_Torlendo(:OLD.id).id := :OLD.id;
pkg_fa.v_Torlendo(:OLD.id).szulo := :OLD.szulo;
pkg_fa.v_Torlendo(:OLD.id).adat := :OLD.adat;
-- nem módosítunk, hogy az integritás rendben legyen
:NEW.id := :OLD.id;
END IF;
END tr_fa;
/
show errors
-- utasítás szintű AFTER trigger végzi el a munka
-- oroszlánrészét
CREATE OR REPLACE TRIGGER tr_fa2
AFTER UPDATE OF id ON fa
DECLARE
v_Id NUMBER;
BEGIN
IF NOT pkg_fa.v_Torles AND pkg_fa.v_Torlendo.COUNT > 0
THEN
pkg_fa.v_Torles := TRUE;
-- Gyerekek átállítása
v_Id := pkg_fa.v_Torlendo.FIRST;
WHILE v_Id IS NOT NULL LOOP
UPDATE fa SET szulo = pkg_fa.szulo_torles_utan(v_Id)
WHERE szulo = v_Id;
v_Id := pkg_fa.v_Torlendo.NEXT(v_Id);
END LOOP;
-- Törlés
v_Id := pkg_fa.v_Torlendo.FIRST;
WHILE v_Id IS NOT NULL LOOP
DELETE FROM fa
```

```
WHERE id = v_Id;
v_Id := pkg_fa.v_Torlendo.NEXT(v_Id);
END LOOP;
pkg_fa.v_Torles := FALSE;
END IF;
END tr_fa2;
/
/*
Emlékeztetőül a fa:
ELEM
---------------------
+--(1, 10)
| +--(2, 20)
| | +--(3, 30)
| | | +--(5, 50)
| | +--(4, 40)
| +--(6, 60)
| +--(7, 70)
+--(8, 80)
| +--(9, 90)
*/
UPDATE fa SET id = -1
WHERE id IN (2, 3, 8);
/*
3 sor módosítva.
*/
SELECT LPAD(' ', (LEVEL-1)*3, '| ') || '+--'
|| '(' || id || ', ' || adat || ')' AS elem
FROM fa
CONNECT BY PRIOR id = szulo
START WITH szulo IS NULL;
/*
ELEM
--------------
+--(1, 10)
| +--(4, 40)
```

```
| +--(5, 50)

| +--(6, 60)

| +--(7, 70)

+--(9, 90)

6 sor kijelölve.

*/
```

14. fejezet - Objektumrelációs eszközök

Az Oracle 10*g* objektumrelációs adatbázis-kezelő rendszer. A relációs adatbázisok szokásos eszközrendszerét objektumorientált eszközrendszerrel bővíti ki. Az objektumorientált nyelvek osztályfogalmának az Oracle-ben az *objektumtípus* felel meg. Egy objektumtípus példányai az *objektumok*. Az objektumtípus *absztrakt adattípus*, az adatstruktúrát *attribútumok*, a viselkedésmódot *metódusok* realizálják.

Az objektumtípus adatbázis-objektum, és a CREATE, ALTER, DROP SQL utasításokkal kezelhető. Objektumtípus PL/SQL programban nem hozható létre, ilyen típusú eszközök viszont kezelhetők.

Az Oracle az *egyszeres öröklődés* elvét vallja. Az Oracle-ben nincs bezárást szabályozó eszköz, az attribútumok és metódusok *nyilvánosak*. Az Oracle objektumorientált eszközrendszerének használata tehát a programozóktól kellő önfegyelmet követel meg, hogy az attribútumokat csak az adott objektumtípus metódusaival és ne közvetlenül manipulálják.

1. Objektumtípusok és objektumok

Egy objektumtípus *specifikációjának* létrehozását az alábbi módon tehetjük meg:

```
CREATE [OR REPLACE] TYPE típusnév

[AUTHID {CURRENT_USER|DEFINER}]

{{IS|AS} OBJECT|UNDER szupertípus_név}

({attribútum adattípus|metódus_spec}

[,(attribútum adattípus|metódus_spec}]…)

[[NOT] FINAL] [[NOT] INSTANTIABLE];

metódus_spec:

[[NOT] OVERRIDING] [[NOT] FINAL] [[NOT] INSTANTIABLE]

[{MAP|ORDER} MEMBER fv_spec]

{MEMBER|STATIC|CONSTRUCTOR} alprogram_spec]
```

Az objektumtípus *törzsét* a következő módon adhatjuk meg:

```
CREATE [OR REPLACE] TYPE BODY típusnév

{IS|AS}

{MEMBER|STATIC|CONSTRUCTOR} alprogramdeklaráció

[{MEMBER|STATIC|CONSTRUCTOR} alprogramdeklaráció]…

[{MAP|ORDER} MEMBER fv_deklaráció]

END;
```

Az OR REPLACE újra létrehozza az objektumtípust, ha az már létezett, anélkül hogy azt előzőleg töröltük volna. Az előzőleg az objektumtípushoz rendelt jogosultságok érvényben maradnak.

A *típusnév* a most létrehozott objektumtípus neve lesz.

Az AUTHID az objektumtípus metódusainak hívásakor alkalmazandó jogosultságokat adja meg. A CURRENT_USER esetén a hívó, DEFINER (ez az alapértelmezés) esetén a tulajdonos jogosultságai érvényesek. Egy altípus örökli szupertípusának jogosultsági szabályozását, azt nem írhatja felül.

Az OBJECT olyan objektumtípust definiál, amelynek nincs szupertípusa. Ez a típus egy önálló objektumtípus-hierarchia gyökértípusa lehet.

Az UNDER a megadott szupertípus altípusaként származtatja a típust, tehát beilleszti abba az objektumtípus-hierarchiába, amelyben a szupertípus is van.

Az *attribútum* az objektumtípus attribútumának neve. Az *adattípus* bármilyen beépített Oracle-típus vagy felhasználói típus lehet, az alábbiak kivételével: ROWID, UROWID, LONG, LONG RAW, BINARY_INTEGER, PLS_INTEGER, BOOLEAN, RECORD, REF CURSOR, %TYPE és %ROWTYPE és PL/SQL csomagban deklarált típus. Minden objektumtípusnak rendelkeznie kell legalább egy attribútummal (ez lehet örökölt is), az attribútumok maximális száma 1000. Az altípusként létrehozott objektumtípus attribútumainak neve nem egyezhet meg a szupertípus attribútumainak nevével. Az attribútumoknak nem adható kezdőérték és nem adható meg rájuk a NOT NULL megszorítás. Egy objektumtípus adatstruktúrája tetszőlegesen bonyolult lehet. Az attribútumok típusa lehet objektumtípus is, ekkor *beágyazott* objektumtípusról beszélünk. Az Oracle megengedi *rekurzív* objektumtípusok létrehozását.

A szintaktikus leírás végén (a kerek zárójeleken kívül) szereplő előírások az objektumtípus öröklődését és példányosítását szabályozzák. A FINAL (ez az alapértelmezés) azt adja meg, hogy az adott objektumtípusból nem származtatható altípus. Az ilyen típus az öröklődési fa levéleleme lesz. NOT FINAL esetén az adott objektumtípusból öröklődéssel származtathatunk altípust. Az INSTANTIABLE (ez az alapértelmezés) esetén az objektumtípus példányosítható, NOT INSTANTIABLE esetén *absztrakt objektumtípus* jön létre, ez nem példányosítható, de örökölthető. Ez utóbbi előírás kötelező, ha az objektumtípus absztrakt metódust tartalmaz. Ekkor az adott objektumtípusnak nincs konstruktora. NOT INSTANTIABLE esetén kötelező a NOT FINAL megadása is.

A metódusok lehetnek eljárások vagy függvények. Ezeket mindig az attribútumok után kell megadni. Az objektumtípus specifikációjában csak az alprogramok specifikációja szerepelhet. A formális paraméterek típusa ugyanaz lehet, mint az attribútumé. Objektumtípushoz nem kötelező metódust megadni, ilyenkor az objektumtípusnak nem lesz törzse sem. A metódusok teljes definícióját az objektumtípus törzsében kell megadni, kivéve ha egy metódusspecifikáció előtt megadjuk a NOT INSTANTIABLE előírást. Ez azt jelenti, hogy a metódus teljes deklarációját valamelyik altípus fogja megadni *(absztrakt metódus)*. Az INSTANTIABLE esetén a törzsbeli implementáció kötelező (ez az alapértelmezés). Az OVERRIDING polimorf metódusoknál kötelező. Azt jelöli, hogy az adott metódus újraimplementálja a szupertípus megfelelő metódusát. Az alapértelmezés NOT OVERRIDING. A FINAL azt határozza meg, hogy az adott metódus nem polimorf, és az altípusokban nem implementálható újra. Az alapértelmezés a NOT FINAL.

A MEMBER metódusok az Oracle-ben példány szintű metódusok. Ezek mindig az *aktuális példányon* operálnak. Első paraméterük implicit módon mindig a SELF, amely az aktuális példányt hivatkozza. Hívási módjuk:

```
objektum_kifejezés.metódusnév(aktuális_paraméter_lista)
```

A SELF explicit módon is deklarálható, de mindig csak első formális paraméterként. Alapértelmezett paraméterátadási módja IN OUT, ha a metódus eljárás, és IN, ha függvény. De a SELF függvény esetén is definiálható explicit módon IN OUT átadási móddal, így a függvény megváltoztathatja a SELF attribútumainak értékét. Ha egy SQL utasításban szereplő metódushívásnál a SELF értéke NULL, akkor a metódus NULL értékkel tér vissza. Procedurális utasításban viszont a SELF_IS_NULL kivétel váltódik ki.

A STATIC metódusok az Oracle-ben osztály szintű metódusok. Nincs implicit paraméterük. Tipikus hívási módjuk:

```
típusnév.metódusnév(aktuális_paraméter_lista)
```

A CONSTRUCTOR metódus segítségével adhatunk az objektumtípushoz konstruktort. Ez egy függvény, amelynek neve kötelezően azonos a típus nevével. A konstruktor is rendelkezik az implicit SELF paraméterrel, amely ekkor IN OUT módú és így értéke akár meg is változtatható a konstruktorban. A konstruktor visszatérési típusának specifikációja kötelezően a következő:

```
RETURN SELF AS RESULT
```

A visszatérési érték mindig a SELF értéke lesz, ezért a RETURN *kifejezés* forma helyett üres RETURN utasítást kell használni. Az Oracle minden egyes objektumtípushoz automatikusan felépít egy alapértelmezett

konstruktort. Ez egy olyan függvény, amelynek neve megegyezik az objektumtípus nevével, paraméterei pedig az attribútumok felsorolásuk sorrendjében, a megadott típussal. A konstruktor meghívása létrehoz egy új példányt, a paraméterek értékét megkapják az attribútumok és a konstruktor visszatér a példánnyal. Az alapértelmezett konstruktor felülbírálható olyan explicit konstruktor megadásával, amelynek formálisparaméter-listája megegyezik az alapértelmezett konstruktoréval, azaz a formális paraméterek neve, típusa és módja is meg kell egyezzen. Egy konstruktort mindig explicit módon kell meghívni olyan helyen, ahol függvényhívás állhat. A konstruktor hívást megelőzheti a NEW operátor, ennek használata nem kötelező, de ajánlott. A konstruktorok nem öröklődnek.

Egy objektumtípus metódusnevei túlterhelhetők. Ekkor a metódusok specifikációjában a formális paraméterek számának, sorrendjének vagy típusának különböznie kell. A konstruktor is túlterhelhető.

Egy objektumtípushoz megadható egy MAP vagy egy ORDER metódus (egyszerre mindkettő nem). Ezek függvények és az adott objektumtípus példányainak rendezésére szolgálnak. Ha sem MAP, sem ORDER metódust nem adunk meg, akkor a példányok csak egyenlőségre és nem egyenlőségre hasonlíthatók össze. Két példány egyenlő, ha a megfelelő attribútumaik értéke páronként egyenlő. Különböző típusú objektumok nem hasonlíthatók össze.

Egy altípus nem adhat meg MAP vagy ORDER metódust, ha ezek a szupertípusban szerepelnek. Viszont a MAP metódus újraimplementálható. Az ORDER metódus viszont nem.

Egy MAP metódus visszatérési típusának valamilyen skalártípusnak kell lennie. A metódust úgy kell megírni, hogy minden példányhoz a skalártípus tartományának egy elemét rendelje. Ez a leképezés hash függvényként működik, és a skalártípus elemeinek sorrendje adja a példányok sorrendjét. A MAP metódus implicit módon meghívódik két megfelelő típusú objektum összehasonlításakor, ugyanakkor explicit módon is hívható. A MAP (miután MEMBER metódus) egyetlen implicit paramétere a SELF. Ha a MAP metódust egy NULL értékű példányra hívjuk meg, akkor NULL-t ad vissza és nem fog lefutni.

Az ORDER metódus rendelkezik az implicit SELF paraméterrel és egy explicit paraméterrel, amely az adott objektumtípus egy példánya. A metódus visszatérési típusa NUMBER (vagy valamelyik altípusa), értéke negatív, nulla vagy pozitív attól függően, hogy a SELF kisebb, egyenlő vagy nagyobb az explicit módon megadott példánynál.

Ha az ORDER metódus paramétere NULL, akkor visszatérési értéke is NULL és nem fog lefutni. Az ORDER metódus is implicit módon meghívódik, ha két objektumot hasonlítunk össze, de meghívható explicit módon is.

1. példa (Verem)

```
CREATE OR REPLACE TYPE T_Egesz_szamok IS TABLE OF INTEGER;

/

CREATE OR REPLACE TYPE T_Verem AS OBJECT (

max_meret INTEGER,

top INTEGER,

elemek T_Egesz_szamok,

-- Az alapértelmezett konstruktort elfedjük

CONSTRUCTOR FUNCTION T_Verem(

max_meret INTEGER,

top INTEGER,

elemek T_Egesz_szamok

) RETURN SELF AS RESULT,

-- Saját konstruktor használata

CONSTRUCTOR FUNCTION T_Verem(
```

```
p_Max_meret INTEGER
) RETURN SELF AS RESULT,
-- Metódusok
MEMBER FUNCTION tele RETURN BOOLEAN,
MEMBER FUNCTION ures RETURN BOOLEAN,
MEMBER PROCEDURE push (n IN INTEGER),
MEMBER PROCEDURE pop (n OUT INTEGER)
);
/
show errors
CREATE OR REPLACE TYPE BODY T_Verem AS
-- Kivétel dobásával meggátoljuk az alapértelmezett konstruktor hívását
CONSTRUCTOR FUNCTION T_Verem(
max_meret INTEGER,
top INTEGER,
elemek T_Egesz_szamok
) RETURN SELF AS RESULT
IS
BEGIN
RAISE_APPLICATION_ERROR(-20001, 'Nem használható ez a konstruktor. '
|| 'Ajánlott konstruktor: T_Verem(p_Max_meret)');
END T_Verem;
CONSTRUCTOR FUNCTION T_Verem(
p_Max_meret INTEGER
) RETURN SELF AS RESULT
IS
BEGIN
top := 0;
/* Inicializáljuk az elemek tömböt a maximális elemszámra. */
max_meret := p_Max_meret;
elemek := NEW T_Egesz_szamok(); -- Ajánlott a NEW használata
elemek.EXTEND(max_meret); -- Előre lefoglaljuk a helyet az elemeknek
RETURN;
MEMBER FUNCTION tele RETURN BOOLEAN IS
BEGIN
-- Igazat adunk vissza, ha tele van a verem
```

```
RETURN (top = max_meret);
END tele;
MEMBER FUNCTION ures RETURN BOOLEAN IS
BEGIN
-- Igazat adunk vissza, ha üres a verem
RETURN (top = 0);
END ures;
MEMBER PROCEDURE push(n IN INTEGER) IS
BEGIN
IF NOT tele THEN
top := top + 1; -- írunk a verembe
elemek(top) := n;
ELSE -- a verem megtelt
RAISE_APPLICATION_ERROR(-20101, 'A verem már megtelt');
END IF;
END push;
MEMBER PROCEDURE pop (n OUT INTEGER) IS
BEGIN
IF NOT ures THEN
n := elemek(top);
top := top - 1; -- olvasunk a veremből
ELSE -- a verem üres
RAISE_APPLICATION_ERROR(-20102, 'A verem üres');
END IF;
END pop;
END;
/
show errors
/* Használata */
DECLARE
v_Verem T_Verem;
i INTEGER;
BEGIN
-- Az alapértelmezett konstruktor már nem használható itt
BEGIN
v_Verem := NEW T_Verem(5, 0, NULL);
```

```
EXCEPTION
WHEN OTHERS THEN
DBMS_OUTPUT.PUT_LINE('Kivétel - 1: ' || SQLERRM);
END;
v_Verem := NEW T_Verem(p_Max_meret => 5);
i := 1;
BEGIN
LOOP
v_Verem.push(i);
i := i + 1;
END LOOP;
EXCEPTION
WHEN OTHERS THEN
DBMS_OUTPUT.PUT_LINE('Kivétel - 2: ' || SQLERRM);
END;
BEGIN
LOOP
v_Verem.pop(i);
DBMS_OUTPUT.PUT_LINE(i);
END LOOP;
EXCEPTION
WHEN OTHERS THEN
DBMS_OUTPUT.PUT_LINE('Kivétel - 3: ' || SQLERRM);
END;
END;
/
/*
Kivétel - 1: ORA-20001: Nem használható ez a konstruktor. Ajánlott
konstruktor: T_Verem(p_Max_meret)
Kivétel - 2: ORA-20101: A verem már megtelt
5
4
3
2
1
Kivétel - 3: ORA-20102: A verem üres
```

```
A PL/SQL eljárás sikeresen befejeződött.

*/
```

2. példa (Racionális számok (példát ad STATIC és MAP metódusokra))

```
CREATE OR REPLACE TYPE T_Racionalis_szam AS OBJECT (

szamlalo INTEGER,

nevezo INTEGER,

STATIC FUNCTION lnko(x INTEGER, y INTEGER) RETURN INTEGER,

-- Megj: van alapértelmezett konstruktor is.

CONSTRUCTOR FUNCTION T_Racionalis_szam(p_Egesz INTEGER)

RETURN SELF AS RESULT,

CONSTRUCTOR FUNCTION T_Racionalis_szam(p_Tort VARCHAR2)

RETURN SELF AS RESULT,

MAP MEMBER FUNCTION konvertal RETURN REAL,

MEMBER PROCEDURE egyszerusit,

MEMBER FUNCTION reciprok RETURN T_Racionalis_szam,

MEMBER FUNCTION to_char RETURN VARCHAR2,

MEMBER FUNCTION plusz(x T_Racionalis_szam) RETURN T_Racionalis_szam,

MEMBER FUNCTION minusz(x T_Racionalis_szam) RETURN T_Racionalis_szam,

MEMBER FUNCTION szorozva(x T_Racionalis_szam) RETURN T_Racionalis_szam,

MEMBER FUNCTION osztva(x T_Racionalis_szam) RETURN T_Racionalis_szam,

PRAGMA RESTRICT_REFERENCES (DEFAULT, RNDS,WNDS,RNPS,WNPS)

);

/

show errors

CREATE OR REPLACE TYPE BODY T_Racionalis_szam AS

STATIC FUNCTION lnko(x INTEGER, y INTEGER) RETURN INTEGER IS

-- Megadja x és y legnagyobb közös osztóját, y előjelével.

rv INTEGER;

BEGIN

IF (y < 0) OR (x < 0) THEN

rv := lnko(ABS(x), ABS(y)) * SIGN(y);

ELSIF (y <= x) AND (x MOD y = 0) THEN

rv := y;

ELSIF x < y THEN

rv := lnko(y, x); -- rekurzív hívás
```

```
ELSE
rv := lnko(y, x MOD y); -- rekurzív hívás
END IF;
RETURN rv;
END;
CONSTRUCTOR FUNCTION T_Racionalis_szam(p_Egesz INTEGER)
RETURN SELF AS RESULT
IS
BEGIN
SELF := NEW T_Racionalis_szam(p_Egesz, 1);
RETURN;
END;
CONSTRUCTOR FUNCTION T_Racionalis_szam(p_Tort VARCHAR2)
RETURN SELF AS RESULT
IS
v_Perjel_poz PLS_INTEGER;
BEGIN
v_Perjel_poz := INSTR(p_Tort, '/');
SELF := NEW T_Racionalis_szam(
TO_NUMBER(SUBSTR(p_Tort,1,v_Perjel_poz-1)),
TO_NUMBER(SUBSTR(p_Tort,v_Perjel_poz+1))
);
RETURN;
EXCEPTION
WHEN OTHERS THEN
RAISE VALUE_ERROR;
END;
MAP MEMBER FUNCTION konvertal RETURN REAL IS
-- Valós számmá konvertálja a számpárral reprezentált racionális számot.
BEGIN
RETURN szamlalo / nevezo;
END konvertal;
MEMBER PROCEDURE egyszerusit IS
-- Egyszerűsíti a legegyszerübb alakra.
l INTEGER;
BEGIN
```

```
l := T_Racionalis_szam.lnko(szamlalo, nevezo);
szamlalo := szamlalo / l;
nevezo := nevezo / l;
END egyszerusit;
MEMBER FUNCTION reciprok RETURN T_Racionalis_szam IS
-- Megadja a reciprokot.
BEGIN
RETURN T_Racionalis_szam(nevezo, szamlalo); -- konstruktorhívás
END reciprok;
MEMBER FUNCTION to_char RETURN VARCHAR2 IS
BEGIN
RETURN szamlalo || '/' || nevezo;
END to_char;
MEMBER FUNCTION plusz(x T_Racionalis_szam) RETURN T_Racionalis_szam IS
-- Megadja a SELF + x összeget.
rv T_Racionalis_szam;
BEGIN
rv := T_Racionalis_szam(szamlalo * x.nevezo + x.szamlalo * nevezo,
nevezo * x.nevezo);
rv.egyszerusit;
RETURN rv;
END plusz;
MEMBER FUNCTION minusz(x T_Racionalis_szam) RETURN T_Racionalis_szam IS
-- Megadja a SELF - x értékét.
rv T_Racionalis_szam;
BEGIN
rv := T_Racionalis_szam(szamlalo * x.nevezo - x.szamlalo * nevezo,
nevezo * x.nevezo);
rv.egyszerusit;
RETURN rv;
END minusz;
MEMBER FUNCTION szorozva(x T_Racionalis_szam) RETURN T_Racionalis_szam IS
-- Megadja a SELF * x értékét.
rv T_Racionalis_szam;
BEGIN
rv := T_Racionalis_szam(szamlalo * x.szamlalo, nevezo * x.nevezo);
```

```
rv.egyszerusit;
RETURN rv;
END szorozva;
MEMBER FUNCTION osztva(x T_Racionalis_szam) RETURN T_Racionalis_szam IS
-- Megadja a SELF / x értékét.
rv T_Racionalis_szam;
BEGIN
rv := T_Racionalis_szam(szamlalo * x.nevezo, nevezo * x.szamlalo);
rv.egyszerusit;
RETURN rv;
END osztva;
END;
/
show errors
/* Egy apró példa: */
DECLARE
x T_Racionalis_szam;
y T_Racionalis_szam;
z T_Racionalis_szam;
BEGIN
x := NEW T_Racionalis_szam(80, -4);
x.egyszerusit;
y := NEW T_Racionalis_szam(-4, -3);
y.egyszerusit;
z := x.plusz(y);
DBMS_OUTPUT.PUT_LINE(x.to_char || ' + ' || y.to_char
|| ' = ' || z.to_char);
-- Alternatív konstrukotrok használata
x := NEW T_Racionalis_szam(10);
y := NEW T_Racionalis_szam('23 / 12');
/*
Hibás konstruktorhívások:
z := NEW T_Racionalis_szam('a/b');
z := NEW T_Racionalis_szam('1.2 / 3.12');
*/
END;
```

```
/
/*
Eredmény:
-20/1 + 4/3 = -56/3
A PL/SQL eljárás sikeresen befejeződött.
*/
```

3. példa (Téglalap (példa ORDER metódusra))

```
CREATE OR REPLACE TYPE T_Teglalap IS OBJECT (
a NUMBER,
b NUMBER,
MEMBER FUNCTION kerulet RETURN NUMBER,
MEMBER FUNCTION terulet RETURN NUMBER,
ORDER MEMBER FUNCTION meret(t T_Teglalap) RETURN NUMBER
)
/
show errors
CREATE OR REPLACE TYPE BODY T_Teglalap AS
MEMBER FUNCTION kerulet RETURN NUMBER IS
BEGIN
RETURN 2 * (a+b);
END kerulet;
MEMBER FUNCTION terulet RETURN NUMBER IS
BEGIN
RETURN a * b;
END terulet;
ORDER MEMBER FUNCTION meret(t T_Teglalap) RETURN NUMBER IS
BEGIN
RETURN CASE
WHEN terulet < t.terulet THEN -1 -- negatív szám
WHEN terulet > t.terulet THEN 1 -- pozitív szám
ELSE 0
END;
END meret;
END;
/
```

```
show errors
DECLARE
v_Teglalap1 T_Teglalap := T_Teglalap(10, 20);
v_Teglalap2 T_Teglalap := T_Teglalap(22, 10);
PROCEDURE kiir(s VARCHAR2, x T_Teglalap, y T_Teglalap) IS
BEGIN
DBMS_OUTPUT.PUT_LINE(s
|| CASE
WHEN x < y THEN 'x < y'
WHEN x > y THEN 'x > y'
ELSE 'x = y'
END);
END kiir;
BEGIN
kiir('1. ', v_Teglalap1, v_Teglalap2);
kiir('2. ', v_Teglalap2, v_Teglalap1);
kiir('3. ', v_Teglalap1, v_Teglalap1);
END;
/
/*
Eredmény:
1. x < y
2. x > y
3. x = y
A PL/SQL eljárás sikeresen befejeződött.
*/
```

4. példa (Öröklődés – T_Arucikk, T_Toll és T_Sorkihuzo típusok (példát ad típus származtatására, absztrakt típusra, ORDER metódusra, objektum típusú attribútumra, valamint polimorfizmusra).)

```
CREATE TYPE T_Arucikk IS OBJECT (
leiras VARCHAR2(200),
ar NUMBER,
mennyiseg NUMBER,
MEMBER PROCEDURE felvesz(p_mennyiseg NUMBER),
MEMBER PROCEDURE elad(p_Mennyiseg NUMBER, p_Teljes_ar OUT NUMBER),
MAP MEMBER FUNCTION osszertek RETURN NUMBER
) NOT FINAL NOT INSTANTIABLE
```

```
/
CREATE OR REPLACE TYPE BODY T_Arucikk AS
MEMBER PROCEDURE felvesz(p_Mennyiseg NUMBER) IS
-- Raktárra veszi a megadott mennyiségű árucikket
BEGIN
mennyiseg := mennyiseg + p_Mennyiseg;
END felvesz;
MEMBER PROCEDURE elad(p_Mennyiseg NUMBER, p_Teljes_ar OUT NUMBER) IS
/* Csökkenti az árucikkből a készletet a megadott mennyiséggel.
Ha nincs elég árucikk, akkor kivételt vált ki.
A p_Teljes_ar megadja az eladott cikkek (esetlegesen
kedvezményes) fizetendő árát. */
BEGIN
IF mennyiseg < p_Mennyiseg THEN
RAISE_APPLICATION_ERROR(-20101, 'Nincs elég árucikk');
END IF;
p_Teljes_ar := p_Mennyiseg * ar;
mennyiseg := mennyiseg - p_Mennyiseg;
END elad;
MAP MEMBER FUNCTION osszertek RETURN NUMBER IS
BEGIN
RETURN ar * mennyiseg;
END osszertek;
END;
/
show errors
CREATE OR REPLACE TYPE T_Kepeslap UNDER T_Arucikk (
meret T_Teglalap
) NOT FINAL
/
show errors
CREATE OR REPLACE TYPE T_Toll UNDER T_Arucikk (
szin VARCHAR2(20),
gyujto INTEGER, -- ennyi darab együtt kedvezményes
kedvezmeny REAL, -- a kedvezmény mértéke, szorzó
OVERRIDING MEMBER PROCEDURE elad(p_Mennyiseg NUMBER,
```

```
p_Teljes_ar OUT NUMBER),
OVERRIDING MAP MEMBER FUNCTION osszertek RETURN NUMBER
) NOT FINAL
/
show errors
CREATE OR REPLACE TYPE BODY T_Toll AS
OVERRIDING MEMBER PROCEDURE elad(p_Mennyiseg NUMBER,
p_Teljes_ar OUT NUMBER) IS
BEGIN
IF mennyiseg < p_Mennyiseg THEN
RAISE_APPLICATION_ERROR(-20101, 'Nincs elég árucikk');
END IF;
mennyiseg := mennyiseg - p_Mennyiseg;
p_Teljes_ar := p_Mennyiseg * ar;
IF p_Mennyiseg >= gyujto THEN
p_Teljes_ar := p_Teljes_ar * kedvezmeny;
END IF;
END elad;
OVERRIDING MAP MEMBER FUNCTION osszertek RETURN NUMBER IS
v_Osszertek NUMBER;
BEGIN
v_Osszertek := mennyiseg * ar;
IF mennyiseg >= gyujto THEN
v_Osszertek := v_Osszertek * kedvezmeny;
END IF;
return v_Osszertek;
END osszertek;
END;
/
show errors
CREATE OR REPLACE TYPE T_Sorkihuzo UNDER T_Toll (
vastagsag NUMBER
)
/
/* Egy apró példa */
DECLARE
```

```
v_Toll T_Toll;

v_Ar NUMBER;

BEGIN

v_Toll := T_Toll('Golyóstoll', 150, 55, 'kék', 8, 0.95);

v_Toll.elad(24, v_Ar);

DBMS_OUTPUT.PUT_LINE('megmaradt mennyiség: ' || v_Toll.mennyiseg

|| ', eladási ár: ' || v_Ar);

END;

/

/*

Eredmény:

megmaradt mennyiség: 31, eladási ár:3420

A PL/SQL eljárás sikeresen befejeződött.

*/
```

A beágyazott tábla és a dinamikus tömb típusok az Oracle-ben speciális objektumtípusnak tekinthetők. Egyetlen attribútumuk van, amelynek típusát megadjuk a létrehozásnál. Ezen kollekciótípusoknál nem adhatunk meg metódusokat, de az Oracle implicit módon felépít hozzájuk kezelő metódusokat. Öröklődés nem értelmezhető közöttük, de a példányosítás létezik és konstruktor is tartozik hozzájuk. Ezen típusok tárgyalását *lásd* a 12. fejezetben.

Egy létező objektumtípus csak akkor cserélhető le CREATE OR REPLACE TYPE utasítással, ha nincsenek típus- vagy táblaszármazékai. A következő utasítás viszont mindig használható egy létező objektumtípus definíciójának megváltoztatására:

```
ALTER TYPE típusnév

{COMPILE [DEBUG] [{BODY|SPECIFICATION}] [REUSE SETTINGS]|

REPLACE [AUTHID {CURRENT_USER|DEFINER}]

AS OBJECT({attribútum adattípus| metódus_spec}

[,{attribútum adattípus|metódus_spec}]…)|

{[NOT]{INSTANTIABLE|FINAL}|

{ADD|DROP} metódus_fej[,{ADD|DROP} metódus_fej]…|

{{ADD|MODIFY} ATTRIBUTE{attribútum[adattípus]|

(attribútum adattípus

[,attribútum adattípus]…)}|

DROP ATTRIBUTE {attribútum|(attribútum[,attribútum]…)}}

[{INVALIDATE|

CASCADE [[NOT] INCLUDING TABLE DATA]

[[FORCE] EXCEPTIONS INTO tábla]}]};
```

Az utasítás a *típusnév* nevű objektumtípust módosítja. A COMPILE újrafordítja a típus törzsét (BODY esetén), specifikációját (SPECIFICATION esetén), vagy mindkettőt. Az újrafordítás törli a fordító beállításait, majd a

fordítás végén újra beállítja azokat. A REUSE SETTINGS megadása esetén ez elmarad. A DEBUG megadása esetén a fordító használja a PL/SQL nyomkövetőjét.

A REPLACE utasításrész lecseréli az objektumtípus teljes attribútum- és metóduskészletét a megadott attribútum- és metóduskészletre. A részleteket a CREATE TYPE utasításnál találjuk.

Az ADD|DROP *metódus_fej* új metódust ad a típushoz, illetve egy meglévőt töröl. Szuper típustól örökölt metódus nem törölhető. Hozzáadásnál csak a metódus specifikációját szerepeltetjük, a metódus implementációját egy CREATE OR REPLACE TYPE BODY utasításban kell megadni.

Az ADD ATTRIBUTE egy új attribútumot ad az objektumtípushoz, ez az eddigi attribútumok után fog elhelyezkedni. A MODIFY ATTRIBUTE módosít egy létező skalár típusú attribútumot (például megváltoztatja valamely korlátozását). A DROP ATTRIBUTE töröl egy attribútumot.

Az INVALIDATE azt írja elő, hogy az Oracle mindenféle ellenőrzés nélkül érvénytelenítsen minden, a módosított objektumtípustól függő adatbázis-objektumot.

A CASCADE megadása esetén az objektumtípus változásainak következményei automatikusan átvezetésre kerülnek az összes függő típusra és táblára. Az INCLUDING TABLE DATA (ez az alapértelmezés) hatására a függő táblák megfelelő oszlopainak értékei az új típusra konvertálódnak. Ha az objektumtípushoz új attribútumot adtunk, akkor az a táblában NULL értéket kap. Ha töröltünk egy attribútumot, akkor az a tábla minden sorából törlődik. NOT INCLUDING TABLE DATA esetén az oszlop értékei változatlanok maradnak, sőt le is kérdezhetők.

Az Oracle a CASCADE hatására hibajelzéssel félbeszakítja az utasítást, ha a függő típusok vagy táblák módosítása során hiba keletkezik. Ez elkerülhető a FORCE megadásával, ekkor az Oracle ignorálja a hibákat és tárolja azokat a megadott *tábla* soraiban.

5. példa

```
CREATE TYPE T_Obj IS OBJECT (

mezo1 NUMBER,

MEMBER PROCEDURE proc1

)

/

CREATE TYPE BODY T_Obj AS

MEMBER PROCEDURE proc1 IS

BEGIN

DBMS_OUTPUT.PUT_LINE('proc1');

END proc1;

END;

/

/* Nem lehet úgy törölni, hogy egy attribútum se maradjon. */

ALTER TYPE T_Obj DROP ATTRIBUTE mezo1;

/*

ALTER TYPE T_Obj DROP ATTRIBUTE mezo1

*

Hiba a(z) 1. sorban:

ORA-22324: a módosított típus fordítási hibákat tartalmaz.
```

```
ORA-22328: A(z) "PLSQL"."T_OBJ" objektum hibákat tartalmaz.

PLS-00589: nem található attribútum a következő objektumtípusban: "T_OBJ"

*/

ALTER TYPE T_Obj ADD ATTRIBUTE mezo2 DATE;

/* A törzs validálása */

ALTER TYPE T_Obj COMPILE DEBUG BODY REUSE SETTINGS;

ALTER TYPE T_Obj ADD

FINAL MEMBER PROCEDURE proc2;

/* A törzset nem elég újrafordítani… */

ALTER TYPE T_Obj COMPILE REUSE SETTINGS;

/*

Figyelmeztetés: A típus módosítása fordítási hibákkal fejeződött be.

*/

/* .. ezért cseréljük a törzs implementációját */

CREATE OR REPLACE TYPE BODY T_Obj AS

MEMBER PROCEDURE proc1 IS

BEGIN

DBMS_OUTPUT.PUT_LINE('proc1');

END proc1;

FINAL MEMBER PROCEDURE proc2 IS

BEGIN

DBMS_OUTPUT.PUT_LINE('proc2');

END proc2;

END;

/
```

Egy objektumtípus megváltoztatása a következő lépésekből áll:

1. Adjunk ki ALTER TYPE utasítást a típus megváltoztatására. Az ALTER TYPE alapértelmezett működésében – ha nem adunk meg semmilyen opciót – ellenőrzi, hogy létezik-e a típustól függő objektum. Ha létezik ilyen, akkor az utasítás sikertelen lesz. Az opcionális alapszavak segítségével megadhatjuk, hogy a változtatások érvényesítése megtörténjen a típustól függő objektumokon és táblákon.

2. Használjuk a CREATE OR REPLACE TYPE BODY utasítást, hogy aktualizáljuk a típus törzsét.

3. Adjunk ki a függő táblákra ALTER TABLE UPGRADE INCLUDING DATA utasítást, ha a táblában szereplő adatokat nem aktualizáltuk az ALTER TYPE utasítással.

4. Változtassuk meg a függő PL/SQL programegységeinket, hogy azok konzisztensek legyenek a változtatott típussal.

5. Aktualizáljuk az adatbázison kívüli, a típustól függő alkalmazásainkat.

Objektumtípus törlésére szolgál a következő utasítás:

```
DROP TYPE típusnév [{FORCE|VALIDATE}];
```

A FORCE megadása lehetővé teszi szupertípus törlését. Ekkor az összes altípus érvénytelenné válik. A FORCE UNUSED minősítéssel látja el azokat az oszlopokat, amelyek *típusnév* típusúak, így ezek elérhetetlenné válnak.

VALIDATE esetén az Oracle csak akkor hajtja végre a törlést, ha az adott objektumtípus példányai nincsenek elhelyezve valamely szupertípusának megfelelő típusú oszlopban.

6. példa

```
DROP TYPE T_Obj;
```

Egy objektumtípus létrehozása után az a szokásos módon használható deklarációs részben bármely programozási eszköz típusának megadásához. Egy objektum típusú programozási eszköz értéke a konstruktorral történő inicializálás előtt NULL.

7. példa

```
DECLARE

x T_Racionalis_szam;

BEGIN

IF x IS NULL THEN

DBMS_OUTPUT.PUT_LINE('NULL');

END IF;

END;

/

/*

NULL

A PL/SQL eljárás sikeresen befejeződött.

*/
```

Egy nem inicializált objektum attribútumainak az értéke NULL, ha azokra egy kifejezésben hivatkozunk. Az ilyen attribútumnak történő értékadás viszont az ACCESS_INTO_NULL kivételt váltja ki. Ha paraméterként adunk át ilyen objektumot, akkor IN mód esetén attribútumainak értéke NULL lesz, OUT és IN OUT estén pedig az ACCESS_INTO_NULL kivétel következik be, ha megpróbálunk az attribútumoknak értéket adni. Bekapcsolt PL/SQL optimalizáló mellett a kivétel kiváltása elmaradhat.

8. példa

```
DECLARE

x T_Teglalap;

BEGIN

DBMS_OUTPUT.PUT_LINE(x.terulet);

EXCEPTION

WHEN SELF_IS_NULL THEN

DBMS_OUTPUT.PUT_LINE('SELF_IS_NULL');

END;

/
```

```
/*
SELF_IS_NULL
A PL/SQL eljárás sikeresen befejeződött.
*/
```

9. példa

```
DECLARE
x T_Racionalis_szam;
BEGIN
DBMS_OUTPUT.PUT_LINE('.' || x.szamlalo || '.');
EXCEPTION
WHEN ACCESS_INTO_NULL THEN
DBMS_OUTPUT.PUT_LINE('ACCESS_INTO_NULL');
END;
/
/*
..
A PL/SQL eljárás sikeresen befejeződött.
*/
```

10. példa

```
-- Kikapcsoljuk az optimalizálót.
ALTER SESSION SET PLSQL_OPTIMIZE_LEVEL=0;
CREATE OR REPLACE PROCEDURE proc_access_into_null_teszt
IS
x T_Racionalis_szam := NEW T_Racionalis_szam(5, 6);
y T_Racionalis_szam;
BEGIN
x.szamlalo := 10;
DBMS_OUTPUT.PUT_LINE('.' || x.szamlalo || '.');
y.szamlalo := 11;
DBMS_OUTPUT.PUT_LINE('.' || y.szamlalo || '.');
EXCEPTION
WHEN ACCESS_INTO_NULL THEN
DBMS_OUTPUT.PUT_LINE('ACCESS_INTO_NULL');
END proc_access_into_null_teszt;
/
```

```
EXEC proc_access_into_null_teszt;
/*
.10.
ACCESS_INTO_NULL
A PL/SQL eljárás sikeresen befejeződött.
*/
-- Visszakapcsoljuk az optimalizálót.
ALTER SESSION SET PLSQL_OPTIMIZE_LEVEL=2;
ALTER PROCEDURE proc_access_into_null_teszt COMPILE;
-- Ha így futtatjuk újra, az értékadás megtörténik
-- egy az optimalizáló által bevezetett ideiglenes memóriabeli
-- változóban, mintha y.szamláló egy külön INTEGER változó lenne.
EXEC proc_access_into_null_teszt;
/*
.10.
.11.
A PL/SQL eljárás sikeresen befejeződött.
*/
```

11. példa

```
DECLARE
x T_Racionalis_szam;
y T_Racionalis_szam;
PROCEDURE kiir(z T_Racionalis_szam) IS
BEGIN
DBMS_OUTPUT.PUT_LINE('.' || z.szamlalo || '.');
END kiir;
PROCEDURE init(
p_Szamlalo INTEGER,
p_Nevezo INTEGER,
z OUT T_Racionalis_szam) IS
BEGIN
z := NEW T_Racionalis_szam(p_Szamlalo, p_Nevezo);
z.egyszerusit;
END init;
PROCEDURE modosit(
```

```
p_Szamlalo INTEGER,
p_Nevezo INTEGER,
z IN OUT T_Racionalis_szam) IS
BEGIN
z.szamlalo := p_Szamlalo;
z.nevezo := p_Nevezo;
z.egyszerusit;
END modosit;
BEGIN
x := NEW T_Racionalis_szam(1, 2);
kiir(x);
init(2, 3, x);
kiir(x);
kiir(y);
init(3, 4, y);
kiir(y);
modosit(4, 5, x);
kiir(x);
y := NULL;
modosit(5, 6, y);
kiir(y);
EXCEPTION
WHEN ACCESS_INTO_NULL THEN
DBMS_OUTPUT.PUT_LINE('Error: ACCESS_INTO_NULL');
DBMS_OUTPUT.PUT_LINE(' SQLCODE:' || SQLCODE);
DBMS_OUTPUT.PUT_LINE(' SQLERRM:' || SQLERRM);
DBMS_OUTPUT.PUT_LINE('Error backtrace:');
DBMS_OUTPUT.PUT_LINE(DBMS_UTILITY.FORMAT_ERROR_BACKTRACE);
END;
/
/*
.1.
.2.
..
.3.
.4.
```

```
Error: ACCESS_INTO_NULL

SQLCODE:-6530

SQLERRM:ORA-06530: Inicializálatlan összetett objektumra való hivatkozás

Error backtrace:

ORA-06512: a(z) helyen a(z) 24. sornál

ORA-06512: a(z) helyen a(z) 43. sornál

A PL/SQL eljárás sikeresen befejeződött.

*/
```

Egy objektumtípus attribútumaira az adott típus példányainak nevével történő minősítéssel hivatkozhatunk.

12. példa

```
DECLARE

v_Arucikk T_Arucikk;

v_Kepeslap T_Kepeslap;

v_Terulet NUMBER;

BEGIN

v_Toll := NEW T_Toll('Golyóstoll', 150, 55, 'kék', 8, 0.95);

DBMS_OUTPUT.PUT_LINE('Tollak mennyisége: ' || v_Toll.mennyiseg);

-- Megváltoztatjuk az árat:

v_Toll.ar := 200;

v_Kepeslap := NEW T_Kepeslap('Boldog szülinapot lap',

80, 200, NEW T_Teglalap(10, 15));

.

.

.
```

Az attribútumnevek láncolata beágyazott objektum esetén megengedett.

13. példa

```
.

.

.

-- Hivatkozás kollekció attribútumára

v_Terulet := v_Kepeslap.meret.terulet;

DBMS_OUTPUT.PUT_LINE('Egy képeslap területe: ' || v_Terulet);

.

.

.
```

Egy konstruktor hívása kifejezésben megengedett, a konstruktor hívásánál a függvényekre vonatkozó szabályok állnak fönn. Ilyenkor azonban a NEW operátor nem használható.

14. példa (A blokk eredménye)

```
.

.

.

-- Konstruktor kifejezésben

DBMS_OUTPUT.PUT_LINE('Egy légből kapott tétel összértékének a fele: '

|| (T_Kepeslap('Üdvözlet Debrecenből',

120, 50, NEW T_Teglalap(15, 10)).osszertek / 2) );

END;

/

/*

Tollak mennyisége: 55

Egy képeslap területe: 150

Egy légből kapott tétel összértékének a fele: 3000

A PL/SQL eljárás sikeresen befejeződött.

*/
```

2. Objektumtáblák

Az objektumtábla olyan speciális tábla, ahol minden sor egy-egy objektumot tartalmaz. A következő utasítások például létrehozzák a T_Szemely objektumtípust, majd egy objektumtáblát a T_Szemely objektumok számára:

```
CREATE TYPE T_Szemely AS OBJECT(

nev VARCHAR2(35),

telefon VARCHAR2(12))

),

/

CREATE TABLE szemelyek OF T_Szemely;
```

Egy objektumtábla megközelítése kétféle módon történhet:

- tekinthetjük olyan táblának, amelynek egyetlen oszlopa van, és ennek minden sorában egy-egy objektum áll, amelyen objektumorientált műveletek végezhetők;
- tekinthetjük olyan táblának, amelynek oszlopai az objektumtípus attribútumai, és így a relációs műveletek hajthatók végre rajta.

A REF adattípus

A REF beépített típus sorobjektumok kezelését teszi lehetővé. A REF egy pointer, amely egy sorobjektumot címez. A REF típus használható oszlop, változó, formális paraméter, rekordmező és attribútum típusának megadásánál:

```
REF objektumtípus [SCOPE IS objektumtábla]
```

alakban. Ha megadjuk a SCOPE IS előírást, akkor csak az adott *objektumtábla* sorai, egyébként bármely, az adott *objektumtípussal* rendelkező objektumtábla sorai hivatkozhatók. A SCOPE IS előírással rendelkező REF típust *korlátozott* REF típusnak hívjuk.

1. példa

```
CREATE TABLE hitel_ugyfelek OF T_Szemely;

CREATE TABLE hitelek (

azonosito VARCHAR2(30) PRIMARY KEY,

osszeg NUMBER,

lejarat DATE,

ugyfel_ref REF T_Szemely SCOPE IS hitel_ugyfelek,

kezes_ref REF T_Szemely -- Nem feltétlen ügyfél

)

/
```

Elképzelhető, hogy egy REF által hivatkozott objektum elérhetetlenné válik (például töröltük vagy megváltozott a jogosultság), ekkor az ilyen hivatkozást *érvénytelen* hivatkozásnak hívjuk. Az SQL az IS DANGLING logikai értékű operátorral tudja vizsgálni egy REF érvényességét. Az IS DANGLING értéke igaz, ha a hivatkozás érvénytelen, egyébként

hamis.

Egy objektumtáblát a relációs DML utasításokkal hagyományos relációs táblaként kezelhetünk.

2. példa

```
INSERT INTO szemelyek VALUES ('Kiss Aranka', '123456');

INSERT INTO szemelyek VALUES (T_Szemely('József István', '454545'));

INSERT INTO hitel_ugyfelek VALUES ('Nagy István', '789878');

INSERT INTO hitel_ugyfelek VALUES ('Kocsis Sándor', '789878');

SELECT nev, telefon FROM hitel_ugyfelek;

SELECT * FROM szemelyek;

UPDATE szemelyek SET telefon = '111111'

WHERE nev = 'Kiss Aranka';

UPDATE szemelyek sz SET sz = T_Szemely('Kiss Aranka', '222222')

WHERE sz.nev = 'Kiss Aranka';
```

Az objektumtáblák sorait mint objektumokat a következő függvényekkel kezelhetjük: VALUE, REF, DEREF, TREAT. Ezek PL/SQL kódban csak SQL utasításokban szerepelhetnek.

VALUE

A VALUE függvény aktuális paramétere egy objektumtábla vagy objektumnézet másodlagos neve lehet és a tábla vagy nézet sorait mint objektumpéldányokat adja vissza. Például a következő lekérdezés azon objektumok halmazát adja, ahol a *nev* attribútum értéke Kiss Aranka:

```
SELECT VALUE(sz) FROM szemelyek sz

WHERE sz.nev = 'Kiss Aranka';
```

A VALUE függvény visszaadja az objektumtábla típusának és minden leszármazott típusának

példányait. Ha azt akarjuk, hogy csak azokat a példányokat kapjuk meg, amelyek

legszűkebb típusa a tábla típusa, használjuk az IS OF operátort ONLY kulcsszóval (*lásd*

később a fejezetben).

REF

A REF függvény aktuális paramétere egy objektumtábla vagy objektumnézet másodlagos neve lehet, és a visszatérési értéke az adott tábla vagy nézet egy objektumpéldányának a referenciája. A REF függvény által visszaadott referencia hivatkozhatja az adott tábla vagy nézet típusának összes leszármazott típusához tartozó példányt is.

Példa

```
INSERT INTO hitelek

SELECT '767-8967-6' AS azonosito, 65000 AS osszeg,

SYSDATE+365 AS lejarat,

REF(u) AS ugyfel_ref, REF(sz) AS kezes_ref

FROM hitel_ugyfelek u, szemelyek sz

WHERE u.nev = 'Nagy István'

AND sz.nev = 'Kiss Aranka'

;

SELECT * FROM hitelek

DELETE FROM hitel_ugyfelek WHERE nev = 'Nagy István';

SELECT * FROM hitelek WHERE ugyfel_ref IS DANGLING;
```

DEREF

PL/SQL kódban a referenciák nem használhatók navigálásra. Egy referencia által hivatkozott objektumhoz a DEREF függvény segítségével férhetünk hozzá. A DEREF aktuális paramétere egy referencia, visszatérési értéke az objektum. Érvénytelen referencia esetén a DEREF NULL-al tér vissza.

Példa

```
SELECT DEREF(ugyfel_ref) AS ugyfel, DEREF(kezes_ref) AS kezes

FROM hitelek;

DELETE FROM hitelek

WHERE DEREF(ugyfel_ref) IS NULL;
```

TREAT

A TREAT függvény az explicit típuskonverziót valósítja meg. Alakja:

```
TREAT(kifejezés AS típus)
```

A *kifejezés* típusát módosítja *típusra*. A *típus* a kifejezés típusának leszármazott típusa. Tehát a TREAT függvény lehetővé teszi az őstípus egy objektumának valamely leszármazott típus példányaként való kezelését (például azért, hogy a leszármazott típus attribútumait vagy metódusait használni tudjuk). Ha az adott objektum nem példánya a leszármazott típusnak, a TREAT NULL-lal tér vissza.

1. példa

```
CREATE TABLE arucikkek OF T_Arucikk;

INSERT INTO arucikkek VALUES(

T_Kepeslap('Boldog névnapot!', 120, 40, T_Teglalap(15, 10))

);

INSERT INTO arucikkek VALUES(

T_Toll('Filctoll', 40, 140, 'piros', 10, 0.9)

);

INSERT INTO arucikkek VALUES(

T_Toll('Filctoll', 40, 140, 'kék', 10, 0.9)

);

INSERT INTO arucikkek VALUES(

T_Sorkihuzo('Jo vastag sorkihuzo', 40, 140, 'zöld', 10, 0.9, 9)

);

SELECT ROWNUM, TREAT(VALUE(a) AS T_Toll).szin AS szin

FROM arucikkek a;
```

Az objektumok kezelésénél alkalmazható az IS OF logikai értékét szolgáltató operátor, alakja:

objektum IS OF ([ONLY] *típus*[,[ONLY] *típus*]...)

ahol a *típus* az objektum típusának leszármazott típusa. Az *objektumot* egy kifejezés szolgáltatja. Az IS OF operátor értéke igaz, ha az objektum az adott *típus* vagy a *típus leszármazott típusainak* példánya. Az ONLY megadása esetén csak akkor ad igaz értéket, ha az adott *típus* példányáról van szó (a leszármazott típusokra már hamis az értéke).

2. példa

```
SELECT ROWNUM, TREAT(VALUE(a) AS T_Toll).szin AS szin

FROM arucikkek a

WHERE VALUE(a) IS OF (T_Toll);

SELECT ROWNUM, TREAT(VALUE(a) AS T_Toll).szin AS szin

FROM arucikkek a

WHERE VALUE(a) IS OF (ONLY T_Toll);
```

A TREAT *kifejezése* és *típusa* is lehet REF típus.

3. példa

```
CREATE TABLE cikk_refek (

toll_ref REF T_Toll,

kihuzo_ref REF T_Sorkihuzo

);

INSERT INTO cikk_refek (toll_ref)

SELECT TREAT(REF(a) AS REF T_Toll) FROM arucikkek a;
```

```
UPDATE cikk_refek SET kihuzo_ref = TREAT(toll_ref AS REF T_Sorkihuzo);

SELECT ROWNUM, DEREF(toll_ref), DEREF(kihuzo_ref) FROM cikk_refek;
```

3. Objektumnézetek

Mint ahogy a relációs nézet egy virtuális tábla, ugyanúgy az *objektumnézet* egy virtuális objektumtábla. Az objektumnézet minden sora egy objektum, amelyre a típusának megfelelő attribútumok és módszerek alkalmazhatók.

Az objektumnézetek jelentősége abban van, hogy a létező relációs táblák adatai objektumorientált alkalmazásokban úgy használhatók fel, hogy azokat nem kell egy másik fizikai struktúrába konvertálni. Egy objektumnézet létrehozásának a lépései a következők:

1. Létre kell hozni egy olyan objektumtípust, ahol minden attribútum megfelel egy létező relációs tábla egy oszlopának.

2. Egy lekérdezés segítségével le kell válogatni a relációs táblából a kezelni kívánt adatokat. Az oszlopokat az objektumtípus attribútumainak sorrendjében kell megadni.

3. Az attribútumok segítségével meg kell adni egy egyedi értéket, amely objektumazonosítóként szolgál, ez teszi lehetővé az objektumnézet objektumainak hivatkozását a REF segítségével. Gyakran a létező elsődleges kulcsot alkalmazzuk.

4. A CREATE VIEW OF utasítás segítségével létre kell hozni az objektumnézetet (az utasítás részletes ismertetését *lásd* [21]).

Példa

```
/*

Kezeljük a könyvtár ügyfeleit T_Szemely típusú objektumként!

*/

CREATE VIEW konyvtar_ugyfelek OF T_Szemely

WITH OBJECT IDENTIFIER(nev) AS

SELECT SUBSTR(id || ' ' || nev, 1, 35) AS nev,

SUBSTR(tel_szam, 1, 12)

FROM ugyfel;

SELECT VALUE(u) FROM konyvtar_ugyfelek u;

/*

Azonban a REF típusnál megadott hitelek táblában kezesként

nem használhatjuk az új T_Szemely-eket.

*/

INSERT INTO hitelek

SELECT 'hitelazon1212' AS azonosito, 50000 AS osszeg,

SYSDATE + 60 AS lejarat,

REF(u) AS ugyfel_ref, REF(k) AS kezes_ref

FROM hitel_ugyfelek u, konyvtar_ugyfelek k

WHERE u.nev = 'Kocsis Sándor'
```

```
AND k.nev like '%József István%'
;
/*
INSERT INTO hitelek
*
Hiba a(z) 1. sorban:
ORA-22979: objektumnézet REF vagy felhasználói REF nem szúrható be (INSERT)
*/
```

15. fejezet - A natív dinamikus SQL

Egy PL/SQL kódba (például egy tárolt alprogramba) beágyazhatók SQL utasítások. Ezek ugyanúgy lefordításra kerülnek, mint a procedurális utasítások és futásról futásra ugyanazt a tevékenységet hajtják végre. Ezen utasítások alakja, szövege fordítási időben ismert és utána nem változik meg. Az ilyen utasításokat *statikus* SQL utasításoknak hívjuk.

A PL/SQL viszont lehetővé teszi azt is, hogy egy SQL utasítás teljes szövege csak futási időben határozódjon meg, tehát futásról futásra változhasson. Ezeket hívjuk *dinamikus* SQL utasításoknak.

A dinamikus SQL utasítások *sztringekben* tárolódnak, melyek futási időben kezelhetők. A sztringek tartalma érvényes SQL utasítás vagy PL/SQL blokk lehet, amelyekbe beágyazhatunk ún. *helykijelölőket*. Ezek nem deklarált azonosítók, melyek első karaktere a kettőspont és a sztring paraméterezésére szolgálnak.

A dinamikus SQL elsősorban akkor használatos, ha

- DDL utasítást (például CREATE), DCL utasítást (például GRANT) vagy munkamenetvezérlő utasítást (például ALTER SESSION) akarunk végrehajtani. Ezek az utasítások statikusan nem hajthatók végre a PL/SQL-ben.
- hatékonyabbá akarjuk tenni a programunkat. Például egy sémaobjektumot csak futási időben szeretnénk meghatározni, vagy WHERE utasításrész feltételét futásról futásra változtatni akarjuk.

A dinamikus SQL megvalósítására a következő utasítás szolgál:

```
EXECUTE IMMEDIATE dinamikus_sztring

[INTO {változó [,változó]…|rekord}]

[USING [IN|OUT|IN OUT] kapcsoló_argumentum

[,[IN|OUT|IN OUT] kapcsoló_argumentum]…]

[{RETURNING|RETURN} INTO kapcsoló_argumentum

[,kapcsoló_argumentum]…];
```

A *dinamikus_sztring* tartalmazza az SQL utasítást (záró pontosvessző nélkül) vagy a PL/SQL blokkot. A *változó* egy olyan változó, amely egy leválogatott oszlopot, a *rekord* pedig egy olyan rekord, amely egy teljes sort tárol. Az INTO utasításrész csak akkor használható, ha a *dinamikus_sztring* olyan SELECT utasítást tartalmaz, amely egyetlen sort ad vissza. A szolgáltatott sor minden oszlopához meg kell adni típuskompatibilis változót, vagy pedig a rekordban léteznie kell egy ilyen mezőnek.

A *dinamikus_sztring* az utasítás fenti formájában nem tartalmazhat több sort visszaadó kérdést. Az ilyen kérdés dinamikus végrehajtására az OPEN–FOR, FETCH, CLOSE utasításokat vagy együttes hozzárendelést használunk. Ezeket a fejezet későbbi részében tárgyaljuk.

A USING utasításrész input (IN – ez az alapértelmezés), output (OUT) vagy input-output (IN OUT) *kapcsoló_argumentumokat* határozhat meg. Ezek IN esetén olyan kifejezések lehetnek, amelyek értéke átkerül a helykijelölőbe. OUT és IN OUT esetén pedig ezek változók, amelyek tárolják az SQL utasítás vagy blokk által szolgáltatott értékeket. A helykijelölők és a kapcsoló argumentumok számának meg kell egyezni (kivéve ha a DML utasításnál szerepel a RETURNING utasításrész), egymáshoz rendelésük a sorrend alapján történik. A *kapcsoló_argumentum* típusa nem lehet speciális PL/SQL típus (például BOOLEAN). NULL érték a *kapcsoló_argumentummal* nem adható át, de egy NULL értékű változóval

igen.

A RETURNING INTO utasításrész olyan dinamikus DML utasítások esetén használható, amelyek RETURNING utasításrészt tartalmaznak (BULK COLLECT utasításrész nélkül). Ez az utasításrész határozza meg azokat a kapcsoló változókat, amelyek a DML utasítás által visszaadott értékeket tárolják. A DML utasítás által szolgáltatott minden értékhez meg kell adni egy típuskompatibilis *kapcsoló_argumentumot*. Ebben az esetben a USING utasításrészben nem lehet OUT vagy IN OUT *kapcsoló_argumentum*.

Az EXECUTE IMMEDIATE utasítás minden végrehajtásánál a *dinamikus_sztring* tartalma elemzésre, majd végrehajtásra kerül.

A következő példák a fejezet végén szereplő *notesz* csomagból valók, és speciálisan az EXECUTE IMMEDIATE utasítás egy-egy tulajdonságát emelik ki:

1. példa (DDL használata és a sémaobjektum nevének dinamikus meghatározása)

```
PROCEDURE tabla_letrehoz

IS

BEGIN

EXECUTE IMMEDIATE 'CREATE TABLE ' || v_Sema || '.notesz_feljegyzesek ('

|| 'idopont DATE NOT NULL,'

|| 'szemely VARCHAR2(20) NOT NULL,'

|| 'szoveg VARCHAR2(3000),'

|| 'torles_ido DATE'

|| ')';

END tabla_letrehoz;
```

2. példa (Az implicit kurzorattribútumok használhatók a dinamikus SQL-lel)

```
FUNCTION torol_lejart

RETURN NUMBER IS

BEGIN

EXECUTE IMMEDIATE 'DELETE FROM ' || v_Sema || '.notesz_feljegyzesek '

|| 'WHERE torles_ido < SYSDATE';

RETURN SQL%ROWCOUNT;

END torol_lejart;
```

3. példa (A USING és RETURNING használata)

```
EXECUTE IMMEDIATE 'INSERT INTO ' || v_Sema || '.notesz_feljegyzesek '

|| 'VALUES (:1, :2, :3, :4) '

|| 'RETURNING idopont, szemely, szoveg, torles_ido '

|| 'INTO :5, :6, :7, :8 '

USING p_Idopont, p_Szemely, p_Szoveg, p_Torles_ido

RETURNING INTO rv.idopont, rv.szemely, rv.szoveg, rv.torles_ido;
```

4. példa (Az INTO használata)

```
EXECUTE IMMEDIATE 'SELECT MIN(idopont) '

|| 'FROM ' || v_Sema || '.notesz_feljegyzesek '

|| 'WHERE idopont > :idopont '

INTO p_Datum

USING p_Datum;
```

Több sort szolgáltató kérdések dinamikus feldolgozása

A dinamikus SQL használatánál is kurzorváltozót kell alkalmazni, ha több sort visszaadó SELECT utasítást akarunk kezelni (a kurzorokkal kapcsolatban *lásd* 8. fejezet).

A dinamikus OPEN–FOR utasítás egy kurzorváltozóhoz hozzárendel egy dinamikus lekérdezést, lefuttatja azt, meghatározza az aktív halmazt, a kurzort az első sorra állítja és nullázza a %ROWCOUNT attribútumot. Alakja:

```
OPEN kurzorváltozó FOR dinamikus_sztring

[USING kapcsoló_argumentum [,kapcsoló_argumentum]…];
```

Az utasításrészek jelentése ugyanaz, mint amit fentebb tárgyaltunk.

1. példa

```
FUNCTION feljegyzesek(

p_Idopont v_Feljegyzes.idopont%tyPE DEfAULT SYSDATE,

p_Szemely v_Feljegyzes.szemely%TYPE DEFAULT NULL

) RETURN t_refcursor IS

rv t_refcursor;

BEGIN

OPEN rv FOR

'SELECT * FROM ' || v_Sema || '.notesz_feljegyzesek '

|| 'WHERE idopont BETWEEN :idopont AND :idopont + 1 '

|| 'AND (:szemely IS NULL OR szemely = :szemely) '

|| 'ORDER BY idopont'

USING TRUNC(p_Idopont), TRUNC(p_Idopont), p_Szemely, p_Szemely;

RETURN rv;

END feljegyzesek;
```

A betöltés és lezárás ugyanúgy történik, mint statikus esetben.

2. példa

```
-- az előző példa függvényével megnyitjuk a kurzort

v_Feljegyzesek_cur := feljegyzesek(p_Idopont, p_Szemely);

LOOP

FETCH v_Feljegyzesek_cur INTO v_Feljegyzes;

EXIT WHEN v_Feljegyzesek_cur%NOTFOUND;

DBMS_OUTPUT.PUT_LINE(v_Feljegyzesek_cur%ROWCOUNT || '. '

|| TO_CHAR(v_Feljegyzes.idopont, 'HH24:MI') || ', '

|| v_Feljegyzes.szemely || ', ' || v_Feljegyzes.szoveg || ', '

|| TO_CHAR(v_Feljegyzes.torles_ido, 'YYYY-MM-DD HH24:MI'));

END LOOP;

CLOSE v_Feljegyzesek_cur;
```

Dinamikus együttes hozzárendelés

Az együttes hozzárendelés statikus verzióját a 12. fejezetben tárgyaltuk. Nézzük, hogyan alkalmazható dinamikus esetben. Az együttes EXECUTE IMMEDIATE utasítás alakja:

```
EXECUTE IMMEDIATE dinamikus_sztring

[BULK COLLECT INTO változó[,változó]…]

[USING kapcsoló_argumentum[,kapcsoló_argumentum]…]

[{RETURNING|RETURN}

BULK COLLECT INTO kapcsoló_argumentum[,kapcsoló_argumentum]…];
```

Egy dinamikus, több sort szolgáltató lekérdezés eredménye a BULK COLLECT INTO utasításrész segítségével úgy kezelhető, hogy az oszlopok értékei egy-egy kollekcióban vagy egy megfelelő *rekord* elemtípusú kollekcióban kerülnek tárolásra. A több sort visszaadó dinamikus INSERT, DELETE, UPDATE esetén RETURNING BULK COLLECT INTO definiálja az outputot fogadó kollekciókat.

Az együttes FETCH alakja:

```
FETCH dinamikus_kurzor

BULK COLLECT INTO változó[,változó]…;
```

Lehetőség van arra, hogy a FORALL (*lásd* 12. fejezet) ciklus belsejében alkalmazzuk az EXECUTE IMMEDIATE utasítást, ha az nem tartalmaz SELECT utasítást. Ennek alakja:

```
FORALL index IN alsó_határ..felső_határ

EXECUTE IMMEDIATE dinamikus_sztring

USING kapcsoló_argumentum[(index)]

[,kapcsoló_argumentum[(index)]]…

[{RETURNING|RETURN} BULK COLLECT INTO

kapcsoló_argumentum[,kapcsoló_argumentum]…];
```

1. példa (BULK COLLECT használata)

```
EXECUTE IMMEDIATE

'SELECT szemely, COUNT(1) '

|| 'FROM ' || v_Sema || '.notesz_feljegyzesek '

|| 'WHERE idopont BETWEEN '

|| 'TO_DATE(''' || v_Ido || ''', ''YYYY-MM-DD'') '

|| 'AND TO_DATE(''' || v_Ido || ''', ''YYYY-MM-DD'') + 1 '

|| v_Szemely_pred

|| 'GROUP BY szemely'

BULK COLLECT INTO p_Nevek, p_Szamok;
```

2. példa (FORALL és %BULK_ROWCOUNT)

```
FORALL i IN 1..p_Szemely_lista.COUNT

EXECUTE IMMEDIATE 'DELETE FROM ' || v_Sema || '.notesz_feljegyzesek '

|| 'WHERE szemely = :szemely'
```

```
USING p_Szemely_lista(i);

p_Szam_lista := t_szam_lista();

p_Szam_lista.EXTEND(p_Szemely_lista.COUNT);

FOR i IN 1..p_Szam_lista.COUNT LOOP

p_Szam_lista(i) := SQL%BULK_ROWCOUNT(i);

END LOOP;
```

A dinamikus SQL használata – esettanulmány

A következő esettanulmány egy noteszt létrehozó és kezelő csomag, mely a PL/SQL számos eszközét felvonultatja, de talán a legfontosabb azt kiemelni, hogy a csomag az aktuális hívó jogosultságaival fut és natív dinamikus SQL-t használ. A csomag specifikációja a következő:

```
CREATE OR REPLACE PACKAGE notesz

AUTHID CURRENT_USER

/*

A csomag segítségével időpontokhoz és

személyekhez kötött feljegyzéseket

készíthetünk és kezelhetünk.

A csomag képes létrehozni a szükséges táblákat

natív dinamikus SQL segítségével, mivel a

csomag eljárásai a hívó jogosultságaival futnak le.

*/

AS

/*********************************************/

/* A csomag által nyújtott típusok, deklarációk */

/*********************************************/

-- Egy feljegyzés formátuma

TYPE t_feljegyzes IS RECORD(

idopont DATE,

szemely VARCHAR2(20),

szoveg VARCHAR2(3000),

torles_ido DATE

);

-- Egy általános REF CURSOR

TYPE t_refcursor IS REF CURSOR;

-- Egy feljegyzes rekord, a %TYPE-okhoz

v_Feljegyzes t_feljegyzes;

-- Kollekciótípusok
```

```
TYPE t_feljegyzes_lista IS TABLE OF t_feljegyzes;
TYPE t_szemely_lista IS TABLE OF v_Feljegyzes.szemely%TYPE;
TYPE t_szam_lista IS TABLE OF NUMBER;
-- A hibás argumentum esetén az alprogramok kiválthatják
-- a hibas_argumentum kivételt.
hibas_argumentum EXCEPTION;
PRAGMA EXCEPTION_INIT(hibas_argumentum, -20200);
/*****************************************/
/* A csomag által nyújtott szolgáltatások */
/*****************************************/
-- A használt séma nevét állíthatjuk át
PROCEDURE init_sema(p_Sema VARCHAR2 DEFAULT USER);
-- Létrehozza a táblát az aktuális sémában
PROCEDURE tabla_letrehoz;
-- Törli a táblát az aktuális sémában
PROCEDURE tabla_torol;
-- Bejegyez egy feljegyzést és visszaadja egy rekordban
FUNCTION feljegyez(
p_Idopont v_Feljegyzes.idopont%TYPE,
p_Szoveg v_Feljegyzes.szoveg%TYPE,
p_Torles_ido v_Feljegyzes.torles_ido%TYPE DEFAULT NULL,
p_Szemely v_Feljegyzes.szemely%TYPE DEFAULT USER
) RETURN t_feljegyzes;
-- Megnyitja és visszaadja az adott személy feljegyzéseit a
-- napra. Ha a szemely NULL, akkor az összes feljegyzést megadja.
FUNCTION feljegyzesek(
p_Idopont v_Feljegyzes.idopont%tyPE DEfAULT SYSDATE,
p_Szemely v_Feljegyzes.szemely%TYPE DEFAULT NULL
) RETURN t_refcursor;
-- Visszaadja egy táblában az adott személy feljegyzéseinek számát a
-- napra. Ekkor p_Nevek és p_Szamok egy-egy elemet tartalmaznak.
-- Ha a p_Szemely NULL, akkor az összes feljegyzést visszadja a napra,
-- p_Nevek és p_Szamok megegyező elemszámúak lesznek.
PROCEDURE feljegyzes_lista(
p_Nevek OUT NOCOPY t_szemely_lista,
p_Szamok OUT NOCOPY t_szam_lista,
```

```
p_Idopont v_Feljegyzes.idopont%TYPE DEFAULT SYSDATE,
p_Szemely v_Feljegyzes.szemely%TYPE DEFAULT NULL
);
-- Kilistázza az adott személy feljegyzéseit a
-- napra. Ha a szemely NULL, akkor az összes feljegyzést megadja.
PROCEDURE feljegyzesek_listaz (
p_Idopont v_Feljegyzes.idopont%TYPE DEFAULT SYSDATE,
p_Szemely v_Feljegyzes.szemely%TYPE DEFAULT NULL
);
-- Megadja az időben első feljegyzés időpontját
PROCEDURE elso_feljegyzes(
p_Datum OUT DATE
);
-- A paraméterben megadja az első olyan feljegyzés időpontját,
-- amely időpontja nagyobb, mint a p_Datum.
-- NULL-t ad vissza, ha nincs ilyen.
PROCEDURE kovetkezo_feljegyzes(
p_Datum IN OUT DATE
);
-- Megadja az időben utolsó feljegyzés időpontját
PROCEDURE utolso_feljegyzes(
p_Datum OUT DATE
);
-- A paraméterben megadja az első olyan feljegyzés időpontját,
-- amely időpontja kisebb, mint a p_Datum.
-- NULL-t ad vissza, ha nincs ilyen.
PROCEDURE elozo_feljegyzes(
p_Datum IN OUT DATE
);
-- Törli a törlési idővel megjelölt és lejárt feljegyzéseket és
-- visszaadja a törölt elemek számát.
FUNCTION torol_lejart RETURN NUMBER;
-- Törli a megadott személyek összes feljegyzéseit és visszaadja
-- az egyes ügyfelekhez a törölt elemek számát.
PROCEDURE feljegyzesek_torol (
p_Szemely_lista t_szemely_lista,
```

```
p_Szam_lista OUT NOCOPY t_szam_lista
);
END notesz;
/
show errors
```

A csomag törzse:

```
CREATE OR REPLACE PACKAGE BODY notesz
AS
/**********************/
/* Privát deklarációk */
/**********************/
-- A séma nevét tárolja
v_Sema VARCHAR2(30);
/*************************************/
/* Publikus deklarációk implementációi */
/*************************************/
-- A használt séma nevét állíthatjuk át
PROCEDURE init_sema(p_Sema VARCHAR2 DEFAULT USER) IS
BEGIN
v_Sema := p_Sema;
END init_sema;
-- Létrehozza a táblát az aktuális sémában
PROCEDURE tabla_letrehoz
IS
BEGIN
EXECUTE IMMEDIATE 'CREATE TABLE ' || v_Sema || '.notesz_feljegyzesek ('
|| 'idopont DATE NOT NULL,'
|| 'szemely VARCHAR2(20) NOT NULL,'
|| 'szoveg VARCHAR2(3000),'
|| 'torles_ido DATE'
|| ')';
END tabla_letrehoz;
-- Törli a táblát az aktuális sémában
PROCEDURE tabla_torol
IS
```

```
BEGIN
EXECUTE IMMEDIATE 'DROP TABLE ' || v_Sema || '.notesz_feljegyzesek';
END tabla_torol;
-- Bejegyez egy feljegyzést és visszaadja egy rekordban
FUNCTION feljegyez(
p_Idopont v_Feljegyzes.idopont%TYPE,
p_Szoveg v_Feljegyzes.szoveg%TYPE,
p_Torles_ido v_Feljegyzes.torles_ido%TYPE DEFAULT NULL,
p_Szemely v_Feljegyzes.szemely%TYPE DEFAULT USER
) RETURN t_feljegyzes
IS
rv t_feljegyzes;
BEGIN
IF p_Idopont IS NULL THEN
RAISE_APPLICATION_ERROR(-20200,
'Nem lehet NULL az időpont egy feljegyzésben');
END IF;
EXECUTE IMMEDIATE 'INSERT INTO ' || v_Sema || '.notesz_feljegyzesek '
|| 'VALUES (:1, :2, :3, :4) '
|| 'RETURNING idopont, szemely, szoveg, torles_ido '
|| 'INTO :5, :6, :7, :8 '
USING p_Idopont, p_Szemely, p_Szoveg, p_Torles_ido
RETURNING INTO rv.idopont, rv.szemely, rv.szoveg, rv.torles_ido;
/* A RETURNING helyett lehetne a USING-ot is használni:
USING ...,
OUT rv.idopont, OUT rv.szemely, OUT rv.szoveg, OUT rv.torles_ido;
Az Oracle azonban a RETURNING használatát javasolja.
*/
RETURN rv;
END feljegyez;
-- Megnyitja és visszaadja az adott személy feljegyzéseit a
-- napra. Ha a szemely NULL, akkor az összes feljegyzést megadja.
FUNCTION feljegyzesek(
p_Idopont v_Feljegyzes.idopont%tyPE DEfAULT SYSDATE,
p_Szemely v_Feljegyzes.szemely%TYPE DEFAULT NULL
) RETURN t_refcursor IS
```

```
rv t_refcursor;
BEGIN
OPEN rv FOR
'SELECT * FROM ' || v_Sema || '.notesz_feljegyzesek '
|| 'WHERE idopont BETWEEN :idopont AND :idopont + 1 '
|| 'AND (:szemely IS NULL OR szemely = :szemely) '
|| 'ORDER BY idopont'
USING TRUNC(p_Idopont), TRUNC(p_Idopont), p_Szemely, p_Szemely;
RETURN rv;
END feljegyzesek;
-- Visszaadja egy táblában az adott személy feljegyzéseinek számát a
-- napra. Ekkor p_Nevek és p_Szamok egy-egy elemet tartalmaznak.
-- Ha a p_Szemely NULL, akkor az összes feljegyzést visszadja a napra,
-- p_Nevek és p_Szamok megegyező elemszámúak lesznek.
PROCEDURE feljegyzes_lista(
p_Nevek OUT NOCOPY t_szemely_lista,
p_Szamok OUT NOCOPY t_szam_lista,
p_Idopont v_Feljegyzes.idopont%TYPE DEFAULT SYSDATE,
p_Szemely v_Feljegyzes.szemely%TYPE DEFAULT NULL
) IS
v_Ido VARCHAR2(10);
v_Szemely_pred VARCHAR2(100);
v_Datum DATE;
BEGIN
v_Ido := TO_CHAR(p_Idopont, 'YYYY-MM-DD');
v_Szemely_pred := CASE
WHEN p_Szemely IS NULL THEN ''
ELSE 'AND ''' || p_Szemely || ''' = szemely '
END;
EXECUTE IMMEDIATE
'SELECT szemely, COUNT(1) '
|| 'FROM ' || v_Sema || '.notesz_feljegyzesek '
|| 'WHERE idopont BETWEEN '
|| 'TO_DATE(''' || v_Ido || ''', ''YYYY-MM-DD'') '
|| 'AND TO_DATE(''' || v_Ido || ''', ''YYYY-MM-DD'') + 1 '
|| v_Szemely_pred
```

```
|| 'GROUP BY szemely'
BULK COLLECT INTO p_Nevek, p_Szamok;
END feljegyzes_lista;
-- Kilistázza az adott személy feljegyzéseit a
-- napra. Ha a szemely NULL, akkor az összes feljegyzést megadja.
PROCEDURE feljegyzesek_listaz(
p_Idopont v_Feljegyzes.idopont%TYPE DEFAULT SYSDATE,
p_Szemely v_Feljegyzes.szemely%TYPE DEFAULT NULL
) IS
v_Feljegyzesek_cur t_refcursor;
v_Feljegyzes t_feljegyzes;
BEGIN
v_Feljegyzesek_cur := feljegyzesek(p_Idopont, p_Szemely);
LOOP
FETCH v_Feljegyzesek_cur INTO v_Feljegyzes;
EXIT WHEN v_Feljegyzesek_cur%NOTFOUND;
DBMS_OUTPUT.PUT_LINE(v_Feljegyzesek_cur%ROWCOUNT || '. '
|| TO_CHAR(v_Feljegyzes.idopont, 'HH24:MI') || ', '
|| v_Feljegyzes.szemely || ', ' || v_Feljegyzes.szoveg || ', '
|| TO_CHAR(v_Feljegyzes.torles_ido, 'YYYY-MM-DD HH24:MI'));
END LOOP;
CLOSE v_Feljegyzesek_cur;
END feljegyzesek_listaz;
-- Megadja az időben első feljegyzés időpontját
PROCEDURE elso_feljegyzes(
p_Datum OUT DATE
) IS
BEGIN
EXECUTE IMMEDIATE 'SELECT MIN(idopont) '
|| 'FROM ' || v_Sema || '.notesz_feljegyzesek'
INTO p_Datum;
END elso_feljegyzes;
-- A paraméterben megadja az első olyan feljegyzés időpontját,
-- amely időpont nagyobb, mint a p_Datum.
-- NULL-t ad vissza, ha nincs ilyen.
PROCEDURE kovetkezo_feljegyzes(
```

```
p_Datum IN OUT DATE
) IS
BEGIN
EXECUTE IMMEDIATE 'SELECT MIN(idopont) '
|| 'FROM ' || v_Sema || '.notesz_feljegyzesek '
|| 'WHERE idopont > :idopont '
INTO p_Datum
USING p_Datum;
END kovetkezo_feljegyzes;
-- Megadja az időben utolsó feljegyzés időpontját
PROCEDURE utolso_feljegyzes(
p_Datum OUT DATE
) IS
BEGIN
EXECUTE IMMEDIATE 'SELECT MAX(idopont) '
|| 'FROM ' || v_Sema || '.notesz_feljegyzesek'
INTO p_Datum;
END utolso_feljegyzes;
-- A paraméterben megadja az első olyan feljegyzés időpontját,
-- amely időpontja kisebb, mint a p_Datum.
-- NULL-t ad vissza, ha nincs ilyen.
PROCEDURE elozo_feljegyzes(
p_Datum IN OUT DATE
) IS
BEGIN
EXECUTE IMMEDIATE 'SELECT MAX(idopont) '
|| 'FROM ' || v_Sema || '.notesz_feljegyzesek '
|| 'WHERE idopont < :idopont '
INTO p_Datum
USING p_Datum;
END elozo_feljegyzes;
-- Törli a törlési idővel megjelölt és lejárt feljegyzéseket és
-- visszaadja a törölt elemek számát.
FUNCTION torol_lejart
RETURN NUMBER IS
BEGIN
```

```
EXECUTE IMMEDIATE 'DELETE FROM ' || v_Sema || '.notesz_feljegyzesek '
|| 'WHERE torles_ido < SYSDATE';
RETURN SQL%ROWCOUNT;
END torol_lejart;
-- Törli a megadott személyek összes feljegyzéseit és visszaadja
-- az egyes ügyfelekhez a törölt elemek számát.
PROCEDURE feljegyzesek_torol(
p_Szemely_lista t_szemely_lista,
p_Szam_lista OUT NOCOPY t_szam_lista
) IS
BEGIN
FORALL i IN 1..p_Szemely_lista.COUNT
EXECUTE IMMEDIATE 'DELETE FROM ' || v_Sema || '.notesz_feljegyzesek '
|| 'WHERE szemely = :szemely'
USING p_Szemely_lista(i);
p_Szam_lista := t_szam_lista();
p_Szam_lista.EXTEND(p_Szemely_lista.COUNT);
FOR i IN 1..p_Szam_lista.COUNT LOOP
p_Szam_lista(i) := SQL%BULK_ROWCOUNT(i);
END LOOP;
END feljegyzesek_torol;
/*************************/
/* Csomag inicializációja */
/*************************/
BEGIN
init_sema;
END notesz;
/
show errors
```

Példaprogram a csomag használatára:

```
DECLARE
v_Datum DATE;
v_Szam NUMBER;
v_Feljegyzes notesz.t_feljegyzes;
v_Feljegyzes_lista notesz.t_feljegyzes_lista;
```

```
v_Feljegyzesek_cur notesz.t_refcursor;
v_Nevek notesz.t_szemely_lista;
v_Szamok notesz.t_szam_lista;
BEGIN
BEGIN
notesz.tabla_torol;
EXCEPTION
WHEN OTHERS THEN
DBMS_OUTPUT.PUT_LINE('Nem volt még ilyen tábla.');
END;
notesz.tabla_letrehoz;
v_Feljegyzes := notesz.feljegyez(SYSDATE - 1, 'Mi volt ?', SYSDATE - 0.0001);
v_Feljegyzes := notesz.feljegyez(SYSDATE, 'Mi van ?');
v_Feljegyzes := notesz.feljegyez(SYSDATE+0.02, 'Mi van haver?',
p_Szemely => 'HAVER');
v_Feljegyzes := notesz.feljegyez(SYSDATE+0.05, 'Mi van haver megint?',
SYSDATE-0.05, 'HAVER');
v_Feljegyzes := notesz.feljegyez(SYSDATE + 1, 'Mi lesz ?');
DBMS_OUTPUT.NEW_LINE;
DBMS_OUTPUT.PUT_LINE('Kurzor teszt:');
v_Feljegyzesek_cur := notesz.feljegyzesek(p_Szemely => 'HAVER');
LOOP
FETCH v_Feljegyzesek_cur INTO v_Feljegyzes;
EXIT WHEN v_Feljegyzesek_cur%NOTFOUND;
DBMS_OUTPUT.PUT_LINE(v_Feljegyzesek_cur%ROWCOUNT || ', '
|| v_Feljegyzes.szoveg);
END LOOP;
CLOSE v_Feljegyzesek_cur;
DBMS_OUTPUT.NEW_LINE;
DBMS_OUTPUT.PUT_LINE('Lista teszt:');
notesz.feljegyzes_lista(v_Nevek, v_Szamok);
FOR i IN 1..v_Nevek.COUNT LOOP
DBMS_OUTPUT.PUT_LINE(i || ', ' || v_Nevek(i) || ': ' || v_Szamok(i));
END LOOP;
DBMS_OUTPUT.NEW_LINE;
DBMS_OUTPUT.PUT_LINE('Listázó teszt:');
```

```
notesz.feljegyzesek_listaz;
DBMS_OUTPUT.NEW_LINE;
DBMS_OUTPUT.PUT_LINE('Idő növekvő:');
notesz.elso_feljegyzes(v_Datum);
WHILE v_Datum IS NOT NULL LOOP
DBMS_OUTPUT.PUT_LINE(TO_CHAR(v_Datum, 'YYYY-MON-DD HH24:MI'));
notesz.kovetkezo_feljegyzes(v_Datum);
END LOOP;
DBMS_OUTPUT.NEW_LINE;
DBMS_OUTPUT.PUT_LINE('Idő csökkenő:');
notesz.utolso_feljegyzes(v_Datum);
WHILE v_Datum IS NOT NULL LOOP
DBMS_OUTPUT.PUT_LINE(TO_CHAR(v_Datum, 'YYYY-MON-DD HH24:MI'));
notesz.elozo_feljegyzes(v_Datum);
END LOOP;
DBMS_OUTPUT.NEW_LINE;
DBMS_OUTPUT.PUT_LINE('Lejárt teszt:');
v_Szam := notesz.torol_lejart;
DBMS_OUTPUT.PUT_LINE('Törölt elemek száma: ' || v_Szam);
EXECUTE IMMEDIATE 'SELECT COUNT(1) FROM notesz_feljegyzesek'
INTO v_Szam;
DBMS_OUTPUT.PUT_LINE('Ennyi maradt: ' || v_Szam);
DBMS_OUTPUT.NEW_LINE;
DBMS_OUTPUT.PUT_LINE('Törlés teszt:');
v_Nevek := notesz.t_szemely_lista('HAVER', 'Haha', 'PLSQL');
notesz.feljegyzesek_torol(v_Nevek, v_Szamok);
FOR i IN 1..v_Nevek.COUNT LOOP
DBMS_OUTPUT.PUT_LINE(i || ', ' || v_Nevek(i) || ': ' || v_Szamok(i));
END LOOP;
EXECUTE IMMEDIATE 'SELECT COUNT(1) FROM notesz_feljegyzesek'
INTO v_Szam;
DBMS_OUTPUT.NEW_LINE;
DBMS_OUTPUT.PUT_LINE('Ennyi maradt: ' || v_Szam);
DBMS_OUTPUT.PUT_LINE('Vége.');
END;
/
```

```
/*
Eredmény:
Kurzorteszt:
1, Mi van haver?
2, Mi van haver megint?
Listateszt:
1, PLSQL: 1
2, HAVER: 2
Listázóteszt:
1, 02:27, PLSQL, Mi van?,
2, 02:56, HAVER, Mi van haver?,
3, 03:39, HAVER, Mi van haver megint?, 2006-06-23 01:15
Idő növekvő:
2006-JÚN. -22 02:27
2006-JÚN. -23 02:27
2006-JÚN. -23 02:56
2006-JÚN. -23 03:39
2006-JÚN. -24 02:27
Idő csökkenő:
2006-JÚN. -24 02:27
2006-JÚN. -23 03:39
2006-JÚN. -23 02:56
2006-JÚN. -23 02:27
2006-JÚN. -22 02:27
Lejárt teszt:
Törölt elemek száma: 2
Ennyi maradt: 3
Törlés teszt:
1, HAVER: 1
2, Haha: 0
3, PLSQL: 2
Ennyi maradt: 0
Vége.
A PL/SQL eljárás sikeresen befejeződött.
*/
```

16. fejezet - Hatékony PL/SQL programok írása

Ebben a fejezetben az Oracle10*g* PL/SQL fordítójával és a kód hatékonyságával kapcsolatos kérdéseket tárgyaljuk. Áttekintjük a fordító optimalizációs beállításait, a feltételes fordítást, a fordító figyelmeztetéseit és a natív nyelvű fordítást. A fejezet végén néhány teljesítményhangolási tanácsot adunk.

1. A fordító beállításai

A fordító viselkedését több inicializációs paraméter szabályozza. Ezek legfeljebb 3 különböző szinten állíthatók be. A legkülső szint az Oracle-példány szintjén történő beállítás (ALTER SYSTEM SET …), ezt felüldefiniálja a munkamenet szintű beállítás (ALTER SESSION SET …). Néhány paraméter kivételével használható a legerősebb mód, amikor a paramétereket az egyes programegységek fordításakor adjuk meg az ALTER … COMPILE utasításban.

Ily módon beállíthatók a következő paraméterek: PLSQL_OPTIMIZE_LEVEL, PLSQL_CODE_TYPE, PLSQL_DEBUG, PLSQL_WARNINGS, PLSQL_CCFLAGS, NLS_LENGTH_SEMANTICS. A már lefordított programegységek esetén érvényben levő beállításokról a metaadatbázis tartalmaz információkat a {DBA|ALL|USER}_PLSQL_OBJECT_SETTINGS adatszótárnézetek megegyező nevű oszlopaiban. Egy újrafordítás során a korábbi beállítások megtarthatók az aktuális munkamenet és példány szintű beállításoktól függetlenül az ALTER … COMPILE utasítás REUSE

SETTINGS utasításrészének használatával.

Példa a PROC nevű eljárás optimalizálás és nyomkövetési információ nélküli újrafordítására úgy, hogy a többi beállítás a korábbi fordításnál használtakkal egyezik meg:

```
ALTER PROCEDURE proc

COMPILE

PLSQL_OPTIMIZE_LEVEL=0

PLSQL_DEBUG=FALSE

REUSE SETTINGS;
```

Az ALTER … COMPILE utasítás DEBUG utasításrészének hatása megegyezik a PLSQL_DEBUG=TRUE beállításával.

2. Az optimalizáló fordító

Egy PL/SQL kódot az optimalizáló úgy tud gyorsabbá tenni, optimalizálni, hogy a programozó által megadott kódot részben átstrukturálja, illetve a kódban szereplő kifejezések kiszámításához minél hatékonyabb módot választ anélkül, hogy a kód értelmét megváltoztatná. A fordító garantálja, hogy egyetlen átalakítás sem eredményezheti explicit alprogramhívás kihagyását, elő nem írt új alprogramhívás bevezetését vagy olyan kivétel kiváltását, ami a kódban eredetileg is ki nem váltódna. Azonban az optimalizálás eredményezheti azt, hogy az optimalizált kód nem vált ki kivételt ott, ahol az optimalizálatlan kód kiváltana (*lásd* 14.1. Objektumtípusok és objektumok – 10. példa).

Az optimalizálás mértékét szabályozhatjuk a PLSQL_OPTIMIZE_LEVEL inicializációs

paraméterrel. Ennek lehetséges értékei:

- 2 – teljes körű optimalizálás, ez az alapértelmezett,
- 1 – lokális optimalizálás, a kódsorok a megadott sorrendben kerülnek végrehajtásra, ekkor lényegében csak a számítások és kifejezések optimalizálására van lehetőség,

- 0 – az optimalizáló kikapcsolása, a korábbi verzióknak megfelelő viselkedés.

Amennyiben a kódot az optimalizáló bekapcsolása mellett, de nyomkövetési információkkal fordítjuk le (ALTER ... COMPILE DEBUG), a kód tényleges optimalizálása nem lehetséges, mert az optimalizálás során a kódsorok sorszáma megváltozhat. Ilyenkor csak a lokális optimalizálás néhány elemét használja a fordító, ezt az esetet így valahová a 0 és 1 beállításértékek közé sorolhatjuk. Éppen ezért törekedjünk arra, hogy nyomkövetési információkat csakis fejlesztői környezetekben használjunk. Éles adatbázisok esetén ez a beállítás mind a teljesítményre, mind a kód méretére negatív hatással van.

Az alapértelmezett beállításhoz képest kevesebb optimalizálást előíró 1 és 0 beállítások egyetlen gyakorlati előnye a kisebb fordítási idő. Ezért azok használata csak akkor tanácsos, ha ez igazán számít, azaz fontosabb a kódok rövid fordítási ideje, mint a lefordított kódok futási ideje. Ilyen helyzet adódhat (de nem feltétlenül adódik) a következő esetekben:

- ejlesztői környezetekben, ahol a nagyon gyakori kódújrafordítások lényegesebbek, mint a futási idő;
- nagyméretű, kevés optimalizálható kódot (kevés számítást és ciklust) tartalmazó generált kód fordításakor;
- egyszer használatos generált kód esetén, ahol a fordítási idő nagyságrendileg is összemérhető a futási idővel.

A teljesség igénye nélkül felsoroljuk és demonstráljuk a fordító néhány legfontosabb optimalizáló műveletét. A példák egyszerű tesztek, ahol az időmérést a SET TIMING ON SQL*Plus parancs és a DBMS_UTILITY.GET_TIME eljárás egyikével végeztük és egy-egy tipikusnak mondható futási időt a példákban rögzítettünk. A teszt pontos körülményei nem lényegesek, céljuk csak az, hogy szemléltessék az egyes példákon belüli optimalizált és optimalizálatlan kód futási idejének arányát. Az optimalizáló fordító működéséről bővebben *lásd* [28/a] és [28/b].

2.1. Matematikai számítások optimalizálása

Ez az optimalizálás abba a kategóriába esik, amelynek más magas szintű nyelvekben nagy hagyományai vannak. Az ilyen kódok optimalizálására sok technika ismeretes, nézzünk ezek közül néhányat:

a) Fordítási időben kiértékelhető kifejezések kiértékelése. Ez többet jelent a konstanskifejezések kiértékelésénél, hiszen például egy X+NULL kifejezés mindig NULL lesz, az X értékétől függetlenül. Ugyanakkor az egyes kifejezések átírásakor is kialakulhatnak hasonló, előre kiszámítható kifejezések, amelyek eredetileg nem konstans kifejezések. Például a 3+X+2 kifejezés helyettesíthető az X+5 kifejezéssel, mivel a PL/SQL nem rögzíti az azonos precedenciájú műveletek kötési sorrendjét.

b) Egy kevésbé hatékony művelet helyett egy hatékonyabb használata. Tegyük fel, hogy valamely típusra a 2-vel való szorzás hatékonyabb, mint az összeadás. Legyen ilyen típusú kifejezés X. Ekkor X+X helyett 2*X használata hatékonyabb. Megfontolandó azonban az, hogy ha például X egy általunk írt függvény hívása, akkor a csere eliminál egy hívást. Lehet azonban, hogy számunkra X mellékhatása is fontos. Ilyen esetben az optimalizáló biztosítja, hogy X ugyanannyiszor kerüljön meghívásra, mint optimalizálás nélkül. Mindemellett azonban megteheti az átalakítást, ha a számítás így gyorsabb lesz. Azaz X+X helyett használhat X*2-t és egy olyan X hívást, amelynek az eredményét egyszerűen ignorálja.

A fordított eset még könnyebben elképzelhető, amikor X+X hatékonyabb, mint X*2. Ekkor azonban X többszöri meghívása következne be, ami nem elfogadható. Használható azonban egy segédváltozó, ha még így is gyorsabb kódot kapunk. Vagyis az

```
Y := X*2;
```

utasítás helyett használható a

```
T := X; Y := T+T;
```

utasítássorozat.

A valóságban ennél jóval bonyolultabb esetek fordulnak elő.

c) Kereszteredmények számítása. Ez akkor történhet, ha egy vagy több kifejezés tartalmaz olyan azonos részkifejezést, amely a részkifejezést tartalmazó kifejezések kiértékelése között nem változik. Ekkor az első számítás során keletkezett részeredmény a másodikban már számítás nélkül újrahasznosítható. Például

```
C := A+B+X; D := (A+B)*Y;
```

helyett állhat

```
T := A+B; C := T+X; D := T*Y;
```

Ha A vagy B értéke a két kifejezésben nem garantáltan ugyanaz, akkor az átalakítás nem végezhető el. Ezt az okozhatja, hogy a két kifejezés közé ékelődik egy olyan utasítás, amelynek hatására (értékadás) vagy mellékhatásaként (alprogramhívás) A vagy B értéke megváltozhat, illetve az, hogy A vagy B maga is függvény.

d) A precedencia által nem kötött műveletek kiértékelési sorrendjének tetszőleges megválasztása. Ebbe a kategóriába tartozik az alprogramok paramétereinek kiértékelési sorrendje is.

Ez a lista közel sem teljes, célja csupán a lehetőségek szemléltetése volt.

1. példa (A következő kódban egy ciklusban a ciklusváltozótól is függő számításokat végzünk, ez lehetőséget ad a számítások optimalizálására.)

```
CREATE OR REPLACE PROCEDURE proc_szamitas(

p_Iter PLS_INTEGER

) IS

a PLS_INTEGER;

b PLS_INTEGER;

c PLS_INTEGER;

d PLS_INTEGER;

BEGIN

FOR i IN 1..p_Iter

LOOP

a := i+1;

b := i-2;

c := b-a+1;

d := b-a-1; -- ismétlődő kifejezés előfordulása

END LOOP;

END proc_szamitas;

/

SHOW ERRORS;

-- Fordítás optimalizálással, majd futtatás

ALTER PROCEDURE proc_szamitas COMPILE PLSQL_OPTIMIZE_LEVEL=2

PLSQL_DEBUG=FALSE;

SET TIMING ON

EXEC proc_szamitas(10000000);
```

```
-- Eltelt: 00:00:03.06

SET TIMING OFF;

-- Fordítás optimalizálás nélkül, majd futtatás

ALTER PROCEDURE proc_szamitas COMPILE PLSQL_OPTIMIZE_LEVEL=0

PLSQL_DEBUG=FALSE;

SET TIMING ON

EXEC proc_szamitas(10000000);

-- Eltelt: 00:00:05.54

SET TIMING OFF;
```

Az optimalizált kód majdnem kétszer gyorsabb. Sőt ha figyelembe vesszük azt is, hogy a ciklus futási ideje mindkét esetben ugyanakkora „szükséges rossz” (ezt a fix időt a következő példáknál pontosan lehet látni), az arány tovább javul az optimalizált kód javára.

2.2. Ciklus magjából a ciklusváltozótól független számítás kiemelése

Az előző példában a ciklusmagban a számítás hivatkozik a ciklus változójára. Tekintsünk egy módosított számítást, ami a ciklusmagtól független és figyeljük meg a futási időket.

2. példa

```
⋮

FOR i IN 1..p_Iter

LOOP

a := 3+1;

b := 3-2;

c := b-a+1;

d := b-a-1; -- ismétlődő kifejezés előfordulása

END LOOP;

⋮

-- Fordítás optimalizálással, majd futtatás

⋮

-- Eltelt: 00:00:00.46

-- Fordítás optimalizálás nélkül, majd futtatás

⋮

-- Eltelt: 00:00:05.53
```

A futási idők között hatalmas lett a különbség. Míg az optimalizálatlan kód futási ideje nem változott, az optimalizált kód ideje radikálisan csökkent, mivel az optimalizáló kiemelte a ciklusmagból a ciklustól független számításokat, így azokat csak egyszer (!) kell elvégezni, nem pedig minden iterációban. Ez a kódolási stílus valójában programozói figyelmetlenség, ami azonban a gyakorlatban gyakran előfordul.

A következő példában mi magunk emeljük ki a független számítást.

3. példa

```
⋮

BEGIN

a := 3+1;

b := 3-2;

c := b-a+1;

d := b-a-1; -- ismétlődő kifejezés előfordulása

FOR i IN 1..p_Iter

LOOP

NULL; -- ez maradt a ciklusmagból

END LOOP;

END proc_szamitas;

⋮

-- Fordítás optimalizálással, majd futtatás

⋮

-- Eltelt: 00:00:00.45

-- Fordítás optimalizálás nélkül, majd futtatás

⋮

-- Eltelt: 00:00:00.45
```

Most nem tapasztalunk különbséget, a számítás csak egyszer fut le mindkét esetben. Tehát a futási idő lényegében a ciklus költsége.

Tipp

Megjegyzés: Valójában az így kialakult csak üres utasítást tartalmazó ciklus teljesen kihagyható lenne, ezt a lépést az optimalizáló azonban szemmel látható módon most még nem teszi meg.

2.3. A CONSTANT kulcsszó figyelembevétele

A nevesített konstansok a deklaráció során kapnak kezdőértéket, és ezután nem változtatják meg értéküket. Ezt kifejezések optimalizálásakor figyelembe veheti az optimalizáló és egyes részkifejezéseket előre kiszámíthat. Ha nevesített konstans helyett változót használunk, akkor a fordító úgy gondolja, hogy a változó értékének stabilitása nem garantált a program két pontja között, ha a két pont között van olyan utasítás, amely mellékhatásával képes megváltoztatni a változó értékét. Ez vagy értékadást, vagy olyan alprogramhívást jelent, amelynek hatáskörében benne van a változó, még akkor is, ha az eljárás nem is hivatkozik a változóra, és ténylegesen nem változtatja meg annak értékét. Tegyük hozzá, hogy csomagbeli alprogram esetén egy ilyen feltétel ellenőrzése csak körülményesen volna lehetséges. (*Vesd össze* PRAGMA RESTRICT_REFERENCES).

A következő példában három esetet vizsgálunk:

- Egy majdnem üres ciklust, ami csak egy eljáráshívást tartalmaz. Ez adja majd az összehasonlítás alapját.
- Egy olyan ciklust, ahol egy változó szerepel, amelynek az értéke azonban soha nem változik. Mivel itt is szerepel majd az eljáráshívás, a fordító nem tekintheti konstansnak a ciklusmagon belül a kifejezést.
- Egy olyan ciklust, ahol nevesített konstans szerepel az előző eset változója helyett.

Mindhárom eset futási idejét látjuk optimalizálással és anélkül, fordított kód esetén.

4. példa

```
CREATE OR REPLACE PROCEDURE proc_konstans(
p_Iter PLS_INTEGER
) IS
c_Konstans CONSTANT NUMBER := 98765;
v_Konstans NUMBER := 98765;
v1 NUMBER := 1;
v2 NUMBER;
-- Tesztelést segítő változók
t NUMBER;
t1 NUMBER;
t2 NUMBER;
v_Ures_ciklus_ideje NUMBER;
-- Eljárás esetleges mellékhatással
PROCEDURE proc_lehet_mellekhatasa IS BEGIN NULL; END;
-- Tesztelést segítő eljárások
PROCEDURE cimke(p_Cimke VARCHAR2) IS
BEGIN
DBMS_OUTPUT.PUT(RPAD(p_Cimke, 20));
END cimke;
PROCEDURE eltelt(
p_Ures_ciklus BOOLEAN DEFAULT FALSE
) IS
BEGIN
t := t2-t1;
IF p_Ures_ciklus THEN
v_Ures_ciklus_ideje := t;
END IF;
DBMS_OUTPUT.PUT_LINE('- eltelt: ' || LPAD(t, 5)
|| ', ciklusidő nélkül:' || LPAD((t-v_Ures_ciklus_ideje), 5));
END eltelt;
-- Az eljárás törzse
BEGIN
cimke('Üres ciklus');
```

```
t1 := DBMS_UTILITY.GET_TIME;
FOR i IN 1..p_Iter
LOOP
proc_lehet_mellekhatasa;
END LOOP;
t2 := DBMS_UTILITY.GET_TIME;
eltelt(p_Ures_ciklus => TRUE);
cimke('Változó használata');
t1 := DBMS_UTILITY.GET_TIME;
FOR i IN 1..p_Iter
LOOP
proc_lehet_mellekhatasa;
v2 := v1 + v_Konstans * 12345;
END LOOP;
t2 := DBMS_UTILITY.GET_TIME;
eltelt;
cimke('Konstans használata');
t1 := DBMS_UTILITY.GET_TIME;
FOR i IN 1..p_Iter
LOOP
proc_lehet_mellekhatasa;
v2 := v1 + c_Konstans * 12345;
END LOOP;
t2 := DBMS_UTILITY.GET_TIME;
eltelt;
END proc_konstans;
/
SHOW ERRORS;
SET SERVEROUTPUT ON FORMAT WRAPPED;
PROMPT 1. PLSQL_OPTIMIZE_LEVEL=2
-- Fordítás optimalizálással, majd futtatás
ALTER PROCEDURE proc_konstans COMPILE PLSQL_OPTIMIZE_LEVEL=2
PLSQL_DEBUG=FALSE;
EXEC proc_konstans(2000000);
PROMPT 2. PLSQL_OPTIMIZE_LEVEL=0
-- Fordítás optimalizálás nélkül, majd futtatás
```

```
ALTER PROCEDURE proc_konstans COMPILE PLSQL_OPTIMIZE_LEVEL=0

PLSQL_DEBUG=FALSE;

EXEC proc_konstans(2000000);

/*

Egy tipikusnak mondható kimenet:

...

1. PLSQL_OPTIMIZE_LEVEL=2

Az eljárás módosítva.

Üres ciklus - eltelt: 78, ciklusidő nélkül: 0

Változó használata - eltelt: 186, ciklusidő nélkül: 108

Konstans használata - eltelt: 117, ciklusidő nélkül: 39

A PL/SQL eljárás sikeresen befejeződött.

2. PLSQL_OPTIMIZE_LEVEL=0

Az eljárás módosítva.

Üres ciklus - eltelt: 79, ciklusidő nélkül: 0

Változó használata - eltelt: 246, ciklusidő nélkül: 167

Konstans használata - eltelt: 245, ciklusidő nélkül: 166

A PL/SQL eljárás sikeresen befejeződött.

*/
```

A futási időkből látszik, hogy a nevesített konstansok alkalmazása csak optimalizáló használata esetén van hatással a futási időre.

Úgy tapasztaltuk, hogy a fordító nem mindig használja ki teljes mértékben a nevesített

konstans adta lehetőségeket.

2.4. Csomaginicializálás késleltetése

Egy inicializáló blokkal ellátott csomagban az inicializáló blokk futása hagyományosan akkor történik meg, amikor a munkamenetben a csomag egy elemére először hivatkozunk. Vannak azonban olyan csomagbeli elemek, amelyek nem függnek a csomag inicializálásától. Az optimalizáló felismeri, ha ilyen elemre hivatkozunk és az ilyen hivatkozás nem váltja ki a csomag inicializálását. Ha egy munkamenet csak ilyen elemekre hivatkozik, akkor a csomag egyáltalán nem kerül inicializálásra, ezzel futási időt takarítunk meg.

Valójában a fordító az inicializáló blokkot egy rejtett eljárássá alakítja, így az attól való függőséget ugyanúgy tudja kezelni, mint más csomagbeli eljárások esetén.

Az optimalizáló azon elemek esetében képes felismerni az inicializáló blokktól való függetlenséget, melyek csak a csomag specifikációjában lévő deklarációjuktól függnek, nem hivatkoznak például alprogramra, és az inicializáló blokkban sem módosulhatnak. Hivatkozhatnak viszont csomagbeli nevesített konstansokra, típusokra, kivételekre és a csomagbeli elemek típusára %TYPE, %ROWTYPE attribútumokkal. Úgy tapasztaltuk, hogy az optimalizáló még ezekben az esetekben sem képes mindig felismerni és felhasználni a függetlenséget, például egy nevesített konstansnál a kezdeti értéket meghatározó kifejezés bonyolultsága is befolyásolja az optimalizálót.

5. példa

```
-- Csomag inicializálással és az inicializálástól nem függő tagokkal
```

```
CREATE OR REPLACE PACKAGE csomag_inittel
IS
-- Néhány csomagbeli elem
c_Konstans CONSTANT NUMBER := 10;
v_Valtozo NUMBER := 10;
TYPE t_rec IS RECORD (id NUMBER, nev VARCHAR2(10000));
END csomag_inittel;
/
CREATE OR REPLACE PACKAGE BODY csomag_inittel
IS
-- Csomaginicializáló blokk
BEGIN
DBMS_OUTPUT.PUT_LINE('Csomaginicializáló blokk.');
-- Egy kis várakozás, sokat dolgozik ez a kód...
DBMS_LOCK.SLEEP(10);
-- A DBMS_LOCK csomag a SYS sémában van,
-- használatához EXECUTE jog szükséges.
END csomag_inittel;
/
-- Csomagot hivatkozó eljárás
CREATE OR REPLACE PROCEDURE proc_csomag_inittel
IS
-- Csomagbeli típus hivatkozása
v_Rec csomag_inittel.t_rec;
v2 csomag_inittel.v_Valtozo%type;
BEGIN
-- Csomagbeli konstans hivatkozása
DBMS_OUTPUT.PUT_LINE('Csomag konstansa: ' || csomag_inittel.c_Konstans);
END proc_csomag_inittel;
/
-- Tesztek
-- Új munkamenet nyitása, hogy a csomag ne legyen inicializálva
CONNECT plsql/plsql
SET SERVEROUTPUT ON FORMAT WRAPPED;
-- Fordítás optimalizálással, majd futtatás
ALTER PROCEDURE proc_csomag_inittel COMPILE PLSQL_OPTIMIZE_LEVEL=2
```

```
PLSQL_DEBUG=FALSE;

SET TIMING ON

EXEC proc_csomag_inittel;

-- Eltelt: 00:00:00.01

SET TIMING OFF;

-- Új munkamenet nyitása, hogy a csomag ne legyen inicializálva

CONNECT plsql/plsql

SET SERVEROUTPUT ON FORMAT WRAPPED;

-- Fordítás optimalizálás nélkül, majd futtatás

ALTER PROCEDURE proc_csomag_inittel COMPILE PLSQL_OPTIMIZE_LEVEL=0

PLSQL_DEBUG=FALSE;

SET TIMING ON

EXEC proc_csomag_inittel;

-- Eltelt: 00:00:10.01

SET TIMING OFF;
```

A példából látszik, hogy a hivatkozó kódot és nem a csomagot kell újrafordítani.

Mivel a csomag specifikációja nem függ a törzstől, így nem lehet ellenőrizni, hogy egy alprogram egy változót vagy kurzort ténylegesen módosít-e vagy sem. Azt tudjuk, hogy megteheti. Hasonló módon egy nevesített konstans függvénnyel történő inicializálásakor nem tudhatjuk, hogy a függvény nem változtatja-e meg a csomag más elemeit. Ezért az így inicializált nevesített konstansok, változók, kurzorok valamint alprogramok hivatkozásakor a csomaginicializálás megtörténik.

2.5. Indexet tartalmazó kifejezések indexelésének gyorsítása

Az X(IND) indexet tartalmazó kifejezés többszöri használata esetén, ha a két használat között X és IND is garantáltan változatlan, az X(IND) által hivatkozott elem címének kiszámítását nem kell újra elvégezni.

6. példa

```
CREATE OR REPLACE PROCEDURE proc_indexes_kifejezes(

p_Iter PLS_INTEGER

) IS

TYPE t_tab IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;

v_Tab t_tab;

v_Index NUMBER;

v_Dummy NUMBER;

-- Tesztelést segítő változók

t NUMBER;

t1 NUMBER;

t2 NUMBER;

v_Ures_ciklus_ideje NUMBER;
```

```
-- Tesztelést segítő eljárások
PROCEDURE cimke(p_Cimke VARCHAR2) IS
BEGIN
DBMS_OUTPUT.PUT(RPAD(p_Cimke, 25));
END cimke;
PROCEDURE eltelt(
p_Ures_ciklus BOOLEAN DEFAULT FALSE
) IS
BEGIN
t := t2-t1;
IF p_Ures_ciklus THEN
v_Ures_ciklus_ideje := t;
END IF;
DBMS_OUTPUT.PUT_LINE('- eltelt: ' || LPAD(t, 5)
|| ', ciklusidő nélkül:' || LPAD((t-v_Ures_ciklus_ideje), 5));
END eltelt;
-- Az eljárás törzse
BEGIN
cimke('Üres ciklus értékadással');
t1 := DBMS_UTILITY.GET_TIME;
FOR i IN 1..p_Iter
LOOP
v_Index := i-i; -- Egy értékadás
END LOOP;
t2 := DBMS_UTILITY.GET_TIME;
eltelt(p_Ures_ciklus => true);
cimke('Változó index');
t1 := DBMS_UTILITY.GET_TIME;
FOR i IN 1..p_Iter
LOOP
v_Index := i-i; -- Egy értékadás
v_Tab(v_Index) := 10;
END LOOP;
t2 := DBMS_UTILITY.GET_TIME;
eltelt;
cimke('Változatlan index');
```

```
t1 := DBMS_UTILITY.GET_TIME;
v_Index := 0;
FOR i IN 1..p_Iter
LOOP
v_Dummy := i-i; -- Egy értékadás, hogy ugyanannyi kód legyen.
v_Tab(v_Index) := 10;
END LOOP;
t2 := DBMS_UTILITY.GET_TIME;
eltelt;
END proc_indexes_kifejezes;
/
SHOW ERRORS;
SET SERVEROUTPUT ON FORMAT WRAPPED;
PROMPT 1. PLSQL_OPTIMIZE_LEVEL=2
-- Fordítás optimalizálással, majd futtatás
ALTER PROCEDURE proc_indexes_kifejezes COMPILE PLSQL_OPTIMIZE_LEVEL=2
PLSQL_DEBUG=FALSE;
EXEC proc_indexes_kifejezes(2000000);
PROMPT 2. PLSQL_OPTIMIZE_LEVEL=0
-- Fordítás optimalizálás nélkül, majd futtatás
ALTER PROCEDURE proc_indexes_kifejezes COMPILE PLSQL_OPTIMIZE_LEVEL=0
PLSQL_DEBUG=FALSE;
EXEC proc_indexes_kifejezes(2000000);
/*
Egy tipikusnak mondható kimenet:
...
1. PLSQL_OPTIMIZE_LEVEL=2
Az eljárás módosítva.
Üres ciklus értékadással - eltelt: 40, ciklusidő nélkül: 0
Változó index - eltelt: 110, ciklusidő nélkül: 70
Változatlan index - eltelt: 83, ciklusidő nélkül: 43
A PL/SQL eljárás sikeresen befejeződött.
2. PLSQL_OPTIMIZE_LEVEL=0
Az eljárás módosítva.
Üres ciklus értékadással - eltelt: 50, ciklusidő nélkül: 0
Változó index - eltelt: 146, ciklusidő nélkül: 96
```

```
Változatlan index - eltelt: 148, ciklusidő nélkül: 98

A PL/SQL eljárás sikeresen befejeződött.

*/
```

2.6. Statikus kurzor FOR ciklusok együttes hozzárendeléssel történő helyettesítése

A fordító képes a statikus kurzor FOR ciklust átírni úgy, hogy 1-1 sor beolvasása helyett bizonyos mennyiségű sort korlátozott méretű kollekciókba együttes hozzárendeléssel olvas be, majd ezeken futtat ciklust és hajtja végre a ciklusmagot.

A következő példában az idő mérése mellett egy fontosabb jellemzőt, a logikai blokkolvasások számát is láthatjuk. Megfelelően nagy számú blokkolvasást igénylő lekérdezésre is szükség volt, ehhez a KONYV tábla Descartes-szorzatait képeztük és ebből megfelelő számú sort olvastunk ki.

7. példa

```
CREATE OR REPLACE PROCEDURE proc_ciklus_bulk

IS

TYPE t_num_tab IS TABLE OF NUMBER;

TYPE t_char_tab IS TABLE OF VARCHAR2(100);

v_Num_tab t_num_tab;

v_Char_tab t_char_tab;

cur SYS_REFCURSOR;

-- Tesztelést segítő változók

t1 NUMBER;

t2 NUMBER;

lio1 NUMBER;

lio2 NUMBER;

v_Session_tag VARCHAR2(100);

-- Tesztelést segítő eljárások

PROCEDURE cimke(p_Cimke VARCHAR2) IS

BEGIN

DBMS_OUTPUT.PUT(RPAD(p_Cimke, 20));

END cimke;

PROCEDURE tag_session

IS

BEGIN

v_Session_tag := 'teszt-' || SYSTIMESTAMP;

DBMS_APPLICATION_INFO.SET_CLIENT_INFO(v_Session_tag);

END tag_session;

FUNCTION get_logical_io RETURN NUMBER
```

```
IS
rv NUMBER;
BEGIN
/*
Itt SYS tulajdonú táblákat hivatkozunk.
A lekérdezéséhez megfelelő jogosultságok szükségesek.
Például SELECT ANY DICTIONARY rendszerjogosultság.
*/
SELECT st.value
INTO rv
FROM v$sesstat st, v$session se, v$statname n
WHERE n.name = 'consistent gets'
AND se.client_info = v_Session_tag
AND st.sid = se.sid
AND n.statistic# = st.statistic#
;
RETURN rv;
END get_logical_io;
PROCEDURE eltelt
IS
BEGIN
DBMS_OUTPUT.PUT_LINE('- eltelt: ' || LPAD(t2-t1, 5)
|| ', LIO (logical I/O) :' || LPAD((lio2-lio1), 7));
END eltelt;
-- Az eljárás törzse
BEGIN
tag_session;
cimke('Statikus kurzor FOR');
t1 := DBMS_UTILITY.GET_TIME;
lio1 := get_logical_io;
FOR i IN (
SELECT k1.id, k1.cim
FROM konyv k1, konyv, konyv, konyv, konyv, konyv
WHERE ROWNUM <= 100000
)
LOOP
```

```
NULL;
END LOOP;
t2 := DBMS_UTILITY.GET_TIME;
lio2 := get_logical_io;
eltelt;
cimke('Kurzor BULK FETCH');
t1 := DBMS_UTILITY.GET_TIME;
lio1 := get_logical_io;
OPEN cur FOR
SELECT k1.id, k1.cim
FROM konyv k1, konyv, konyv, konyv, konyv, konyv
WHERE ROWNUM <= 100000
;
LOOP
FETCH cur
BULK COLLECT INTO v_Num_tab, v_Char_tab
LIMIT 100;
EXIT WHEN v_Num_tab.COUNT = 0;
FOR i IN 1..v_Num_tab.COUNT
LOOP
NULL;
END LOOP;
END LOOP;
CLOSE cur;
t2 := DBMS_UTILITY.GET_TIME;
lio2 := get_logical_io;
eltelt;
EXCEPTION
WHEN OTHERS THEN
IF cur%ISOPEN THEN CLOSE cur; END IF;
RAISE;
ND proc_ciklus_bulk;
/
SHOW ERRORS;
SET SERVEROUTPUT ON FORMAT WRAPPED;
-- Fordítás optimalizálással, majd futtatás
```

```
PROMPT 1. PLSQL_OPTIMIZE_LEVEL=2

ALTER PROCEDURE proc_ciklus_bulk COMPILE PLSQL_OPTIMIZE_LEVEL=2

PLSQL_DEBUG=FALSE;

EXEC proc_ciklus_bulk;

PROMPT 2. PLSQL_OPTIMIZE_LEVEL=0

-- Fordítás optimalizálás nélkül, majd futtatás

ALTER PROCEDURE proc_ciklus_bulk COMPILE PLSQL_OPTIMIZE_LEVEL=0

PLSQL_DEBUG=FALSE;

EXEC proc_ciklus_bulk;

/*

Egy tipikusnak mondható kimenet:

...

1. PLSQL_OPTIMIZE_LEVEL=2

Az eljárás módosítva.

Statikus kurzor FOR - eltelt: 100, LIO (logical I/O) : 34336

Kurzor BULK FETCH - eltelt: 81, LIO (logical I/O) : 34336

A PL/SQL eljárás sikeresen befejeződött.

2. PLSQL_OPTIMIZE_LEVEL=0

Az eljárás módosítva.

Statikus kurzor FOR - eltelt: 636, LIO (logical I/O) : 133336

Kurzor BULK FETCH - eltelt: 80, LIO (logical I/O) : 34336

A PL/SQL eljárás sikeresen befejeződött.

*/
```

Az eredményből látszik, hogy az explicit együttes hozzárendelés volt még így is a leggyorsabb. Ehhez azonban jól el kellett találni a FETCH utasításban a LIMIT kulcsszó után megadott korlátot (100). A korlát változtatása ugyanis nagyban befolyásolja az eredményt, de ezen állítás ellenőrzését az Olvasóra bízzuk.

A logikai I/O azért kevesebb BULK COLLECT esetén, mert ilyenkor a szerver egy FETCH során beolvasott blokkokból több sort is kiolvas egyszerre, míg soronkénti betöltésnél, minden FETCH blokkolvasást eredményez akkor is, ha két egymás után beolvasott sor ugyanabban a blokkban tárolódik. Az együttes hozzárendelés hátránya, hogy több memóriát igényel. Azonban ha explicit módon használjuk, akkor a kollekciók méretét is tudjuk szabályozni a rendszer követelményeinek megfelelően.

3. Feltételes fordítás

Feltételes fordítás során a tényleges fordítást előfordítás előzi meg. Az előfordító a fordításkor fennálló körülményektől függően a megadott kód szövegéből egy előfordítói direktívákat már nem tartalmazó kódot állít elő, ez kerül azután lefordításra. A direktívák kis- és nagybetűre nem érzékenyek. A feltételes fordítás használatának tipikus esetei:

- Egy általános rutin megírásakor fel tudunk készülni arra, hogy a kód különböző adatbázis-kezelő verziókban is ki tudja használni a nyelv új eszközeit az adott környezetben rendelkezésre álló verziótól függően.

- Nyomkövetési kódot helyezhetünk el a programban, amit a feltételes fordítás segítségével csak fejlesztői környezetben használunk. Éles környezetben ez a kód nem kerül lefordításra, nem csökken a teljesítmény, még egy feltétel vizsgálatának erejéig sem.

Az előfordítói szerkezetekben használható kifejezések speciálisan csak fordítási időben kiértékelhető BOOLEAN, PLS_INTEGER és VARCHAR2 típusú kifejezések lehetnek. Ezek jelölésére a formális leírásban rövidítve rendre a *bkf, pkf, vkf* nemterminálisokat használjuk.

```
bkf:

{TRUE | FALSE | NULL |

bkf hasonlító_operátor bkf |

pkf hasonlító_operátor pkf |

NOT bkf | bkf AND bkf | bkf OR bkf |

{bkf|pkf|vkf} IS [NOT] NULL |

csomagbeli_boolean_statikus_nevesített_konstans |

boolean_értékdirektíva}

pkf:

{pls_integer_literál | NULL |

csomagbeli_pls_integer_statikus_nevesített_konstans |

pls_integer_értékdirektíva}

vkf:

{char_literál | NULL |

TO_CHAR(pkf) | TO_CHAR(pkf, vkf, vkf) |

{pkf|vkf} || {pkf|vkf} |

varchar2_értékdirektíva}
```

A használni kívánt csomagbeli nevesített konstansokat CONSTANT kulcsszóval kell a csomag specifikációjában deklarálni, típusuk BOOLEAN vagy PLS_INTEGER lehet, kezdőértékként pedig megfelelő típusú fenti konstanskifejezés használható. Az ilyen nevesített konstansok hivatkozása *csomag.név* formájú, $ jel nélkül. Az értékdirektívák alakja:

```
$$azonosító
```

Az értékdirektívák a PLSQL_CCFLAGS paraméterből kapnak értéket. A paraméterben vesszővel elválasztva több értéket is megadhatunk *azonosító:érték* formában. Az érték BOOLEAN vagy PLS_INTEGER literál vagy NULL lehet. Ha a direktíva a PLSQL_CCFLAGS paraméterben nem szereplő azonosítóra hivatkozik, akkor értéke NULL lesz PLW-6003 figyelmeztetés mellett.

Ha a PLSQL_CCFLAGS paraméterben nincsenek explicit módon megadva, akkor a következő azonosítók speciális jelentéssel bírnak egy értékdirektívában:

- A nevükkel lekérdezhetők a fordító működését szabályozó inicializációs paraméterek értékei (például $$PLSQL_CCFLAGS, $$PLSQL_OPTIMIZE_LEVEL stb.).
- $$PLSQL_LINE értéke a kódsor sorszáma a programegységen belül.
- $$PLSQL_UNIT értéke a forrás programegységének neve, névtelen blokk esetén NULL.

Az elágaztató direktíva formája:

```
$IF bkf $THEN kód
```

```
[ $ELSIF bkf $THEN kód ]…

[ $ELSE kód ]

$END
```

A szemantika megegyezik az IF…THEN utasításéval. Használható még a hibadirektíva, amely kiértékelése PLS-00179 fordítási idejű hibát generál:

```
$ERROR vkf $END
```

1. példa

```
ALTER SESSION SET PLSQL_CCFLAGS='debug:2';

CREATE OR REPLACE PROCEDURE proc_preproc

IS

BEGIN

$IF $$debug = 1 $THEN

DBMS_OUTPUT.PUT_LINE('Van DEBUG');

$ELSIF $$debug = 0 $THEN

DBMS_OUTPUT.PUT_LINE('Nincs DEBUG');

$ELSIF $$debug IS NOT NULL $THEN

$ERROR 'Érvénytelen DEBUG beállítás: ' ||

'PLSQL_CCFLGAS=' || $$PLSQL_CCFLAGS $END

$END

DBMS_OUTPUT.PUT_LINE('Kód');

END proc_preproc;

/

SHOW ERRORS;

/*

Figyelmeztetés: Az eljárás létrehozása fordítási hibákkal fejeződött be.

Hibák PROCEDURE PROC_PREPROC:

LINE/COL ERROR

-------- -----------------------------------------------------------------

9/5 PLS-00179: $ERROR: Érvénytelen DEBUG beállítás:

PLSQL_CCFLGAS=debug:2

*/
```

Az előfordító elérhető programból is a DBMS_PREPROCESSOR csomagon keresztül. Lehetőségünk van szöveggel adott vagy már tárolt kód előfordítására. Az előfordított kód lekérdezésére szolgál a túlterhelt GET_POST_PROCESSED_SOURCE függvény. A szintén túlterhelt PRINT_POST_PROCESSED_SOURCE az előfordított kódot írja a szerverkimenetre. Tárolt kód esetén akkor is működnek az eljárások, ha a kód utolsó fordítása nem volt sikeres fordítási hiba miatt, ilyenkor azonban az előfordító csak a hiba helyéig alakítja át a kódot, akkor is, ha a fordítási hibát nem hibadirektíva váltotta ki.

2. példa

```
-- Lefordítjuk sikeresen az eljárást, hogy szép kódot lássunk majd.

ALTER PROCEDURE proc_preproc COMPILE PLSQL_CCFLAGS='debug:1';

SET SERVEROUTPUT ON FORMAT WRAPPED;

BEGIN

DBMS_PREPROCESSOR.PRINT_POST_PROCESSED_SOURCE(

object_type => 'PROCEDURE'

, schema_name => 'PLSQL'

, object_name => 'PROC_PREPROC'

);

END;

/

/*

Az eljárás módosítva.

PROCEDURE proc_preproc

IS

BEGIN

DBMS_OUTPUT.PUT_LINE('Van DEBUG');

DBMS_OUTPUT.PUT_LINE('Kód');

END proc_preproc;

A PL/SQL eljárás sikeresen befejeződött.

*/
```

Ha az előfordítás előtti eredeti kód hivatkozik csomagbeli nevesített konstansra, a kód és a csomag specifikációja között függőség alakul ki, még akkor is ha az előfordítás utánikód már nem hivatkozik a csomagra. Egy hivatkozott programegység újrafordítása a függőprogramegységek teljes újrafordítását vonja maga után, beleértve az esetleges előfordításokat is.

3. példa

```
ALTER SESSION SET PLSQL_CCFLAGS='konstansom_erteke:TRUE';

CREATE OR REPLACE package pack_konstansok

IS

c_Konstans CONSTANT BOOLEAN := $$konstansom_erteke;

END;

/

CREATE OR REPLACE PROCEDURE proc_preproc_pack

IS

BEGIN

$IF pack_konstansok.c_Konstans $THEN

DBMS_OUTPUT.PUT_LINE('pack_konstansok.c_konstans TRUE');
```

```
$ELSE

DBMS_OUTPUT.PUT_LINE('pack_konstansok.c_konstans FALSE');

$END

END proc_preproc_pack;

/

SET SERVEROUTPUT ON FORMAT WRAPPED;

EXEC proc_preproc_pack;

-- pack_konstansok.c_konstans TRUE

-- A csomag explicit újrafordítása az eljárást érvényteleníti

ALTER PACKAGE pack_konstansok COMPILE

PLSQL_CCFLAGS='konstansom_erteke:FALSE';

-- Most fog az eljárás újrafordulni automatikusan

EXEC proc_preproc_pack;

-- pack_konstansok.c_konstans FALSE
```

Az adatbázis-kezelő verziója lekérdezhető a DBMS_DB_VERSION csomag PLS_INTEGER típusú VERSION és RELEASE nevesített konstansaival. A csomagban további nevesített konstansok is megtalálhatók (*lásd* [18]).

Az előfordítói direktívák használatára vonatkozóan a következő szabályok érvényesek:

- Nem befolyásolhatják tárolt adatbázistípusok szerkezetét, ezért nem használhatók objektumtípus specifikációjában és sémában tárolt kollekciótípusok megadásakor.
- Használhatók a paraméter nélküli programegységekben az IS/AS kulcsszó után.
- Használhatók a paraméterrel rendelkező tárolt függvényben és eljárásban a formálisparaméter-listát nyitó zárójel után.
- Használhatók triggerekben és névtelen blokkokban közvetlenül a kezdő DECLARE vagy BEGIN után.
- Környezeti gazdaváltozó helykijelölője nem szerepelhet elágaztató direktíván belül. Ez névtelen blokkban fordulhatna elő.

4. A fordító figyelmeztetései

Figyelmeztetéseket a fordítási hibákhoz hasonlóan a fordítás során kaphatunk. Ezek a kódban található nem biztonságos, nem konzisztens kódrészlet előfordulását jelzik. A kód lehetfélreérthető, tartalmazhat elérhetetlen kódrészletet, hivatkozhat definiálatlan előfordítóiértékdirektívára, tartalmazhat teljesítménycsökkenést okozó implicit konverziót stb.

A figyelmeztetések generálását vagy tiltását a PLSQL_WARNINGS paraméterrel tudjuk szabályozni, az alapértelmezett értéke ’DISABLE:ALL’ letiltja a figyelmeztetések generálását.

A figyelmeztetések tartalmaznak egy hibakódot PLW-XXXXX alakban. Jelenleg 11 parametrizált figyelmeztetést találunk a hivatalos Oracle-dokumentációban (*lásd* [16] 37.

fejezet).

A figyelmeztetések (a kódjuk alapján) három csoportba tartoznak. A kategóriák megkönnyítik a figyelmeztetések állapotának kezelését:

- SEVERE (SÚLYOS): a kiváltó kód nem várt viselkedést, rossz működést eredményezhet;

- PERFORMANCE (TELJESÍTMÉNY): a kiváltó kód a teljesítmény csökkenését eredményezi (például implicit konverzió hozzárendelt változónál SQL utasításban, vagy NOCOPY elhagyása);
- INFORMATIONAL (TÁJÉKOZTATÓ): a kód szemantikailag nem hibás és nem is okoz teljesítménycsökkenést, viszont kevésbé karbantartható és olvasható, például elérhetetlen kódrészlet van a programban.

A fordító szempontjából minden figyelmeztetés három lehetséges állapot egyikében lehet:

1. DISABLE: tiltott, a figyelmeztetés nem generálódik a fordítás során,
2. ENABLE: engedélyezett, a figyelmeztetés generálódik a fordítás során, a kód lefordul,
3. ERROR: hiba, a figyelmeztetés fordítási hibát generál, a kód nem fordul le.

Az állapotok beállítása kategória vagy kód alapján lehetséges:

```
PLSQL_WARNINGS = 'beállítás' [, 'beállítás']...

beállítás:

állapot:{ ALL | kategória | kódszám |

( kódszám [, kódszám ]… )}
```

A figyelmeztetések a fordítási hibákhoz hasonlóan a {DBA|ALL|USER}_ERRORS nézetből, vagy az SQL*Plus SHOW ERRORS parancsával kérdezhetők le. A figyelmeztetések rendszerszintű bekapcsolását javasoljuk használni (ALTER SYSTEM SET …).

Példa

```
-- Munkamenet beállítása, figyelmezetések tiltása

ALTER SESSION SET

PLSQL_WARNINGS='DISABLE:ALL'

PLSQL_CCFLAGS=''

-- Csúnya, de hibátlan alprogram

CREATE OR REPLACE PROCEDURE proc_warn

IS

v_Id VARCHAR2(100);

v_Name VARCHAR2(100);

to_char BOOLEAN; -- PLW-05004, megengedett, TO_CHAR nem kulcsszó

BEGIN

$IF $$kojak $THEN $END -- PLW-06003, kojak nincs a PLSQL_CCFLAGS-ben

SELECT cim

INTO v_Name

FROM konyv

WHERE id = v_Id -- PLW-07204, id NUMBER, v_id VARCHAR2

;

IF FALSE THEN

NULL; -- PLW-06002, az IF feltétele mindig hamis
```

```
END IF;
END proc_warn;
/
SHOW ERRORS;
/*
Az eljárás létrejött.
Nincsenek hibák.
*/
-- Súlyos figyelmeztetések engedélyezése
ALTER PROCEDURE proc_warn COMPILE
PLSQL_WARNINGS='DISABLE:ALL', 'ENABLE:SEVERE';
SHOW ERRORS;
/*
SP2-0805: Az eljárás fordítási figyelmeztetésekkel lett módosítva.
Hibák PROCEDURE PROC_WARN:
LINE/COL ERROR
-------- -----------------------------------------------------------------
5/3 PLW-05004: a(z) TO_CHAR azonosító a STANDARD csomagban is
definiálva van vagy beépített SQL elem.
*/
-- Teljesítmény figyelmeztetések engedélyezése
ALTER PROCEDURE proc_warn COMPILE
PLSQL_WARNINGS='DISABLE:ALL', 'ENABLE:PERFORMANCE';
SHOW ERRORS;
/*
SP2-0805: Az eljárás fordítási figyelmeztetésekkel lett módosítva.
Hibák PROCEDURE PROC_WARN:
LINE/COL ERROR
-------- -----------------------------------------------------------------
11/15 PLW-07204: az oszlop típusától eltérő konverzió optimum alatti
lekérdezési szerkezetet eredményezhet
*/
-- Tájékoztató figyelmeztetések engedélyezése
ALTER PROCEDURE proc_warn COMPILE
PLSQL_WARNINGS='DISABLE:ALL', 'ENABLE:INFORMATIONAL';
SHOW ERRORS;
```

```
/*

SP2-0805: Az eljárás fordítási figyelmeztetésekkel lett módosítva.

Hibák PROCEDURE PROC_WARN:

LINE/COL ERROR

-------- -----------------------------------------------------------------

7/7 PLW-06003: ismeretlen lekérdezési direktíva: '$$KOJAK'

13/6 PLW-06002: Nem elérhető kód

-- Minden figyelmeztetés legyen fordítási hiba

ALTER PROCEDURE proc_warn COMPILE

PLSQL_WARNINGS='ERROR:ALL';

SHOW ERRORS;

/*

Figyelmeztetés: Az eljárás módosítása fordítási hibákkal fejeződött be.

Hibák PROCEDURE PROC_WARN:

LINE/COL ERROR

-------- -----------------------------------------------------------------

7/7 PLS-06003: ismeretlen lekérdezési direktíva: '$$KOJAK'

*/
```

A figyelmeztetések beállításai módosíthatók a DBMS_WARNING csomag eljárásaival is. A csomag ezenkívül tartalmaz még függvényeket a beállítások lekérdezésére.

5. Natív fordítás

Natív fordítás során a p-kódból (*lásd* 9. fejezet) a fordító egy olyan C nyelvű forrásszöveget hoz létre, amely közvetlenül tartalmazza az utasításoknak megfelelő API hívásokat. Ezt a Cforrást aztán a rendszerben rendelkezésre álló C fordítóval fordítja be egy osztott könyvtárba(*.so* vagy *.dll* állományba). Az osztott könyvtárat az adatbázisban is tárolja, hogy azadatbázis mentésében benne legyen. Amikor az így lefordított programegységet végre kellhajtani, már nincs szükség az interpreterre, elég az osztott könyvtárat használni. Így megnyerjüka kód értelmezésének idejét és a C fordító által végzett optimalizálásokból is profitálunk,a fordítási idő azonban növekszik.

Ebből következik, hogy a natív fordítás elsősorban a nagy számításigényű kódoknál eredményezhetjelentősebb teljesítménynövekedést. Az SQL utasítások végrehajtását nem gyorsítja, a típus- és csomagspecifikációkat sem nagyon, mert azokban vagy nincs futtathatókód vagy csak nagyon kevés. A fordítás módja miatt garantált, hogy a natívra fordított kódfutási ideje nem lesz rosszabb, mint az interpretált kódé, és működésük is azonos lesz,mivel ugyanazok az alacsony szintű API hívások történnek meg mindkettő esetén. A PL/SQL fordító optimalizáló tevékenysége is ugyanaz mindkét esetben.

A natív fordítást a PLSQL_CODE_TYPE inicializációs paraméterrel szabályozzuk, értéke INTERPRETED vagy NATIVE lehet. Az INTERPRETED az alapértelmezett. A PL/SQL programegységeknél egyenként is dönthetünk a fordítás módjáról. Lehetőség van arra is, hogy akár az összes kódot natívra fordítsuk, beleértve a rendszerhez tartozó PL/SQL csomagokat is. Mindezt megtehetjük az adatbázis létrehozásakor vagy már létező adatbázisnál is.

Natív fordításhoz a DBA segítségével be kell még állítani a PLSQL_NATIVE_LIBRARY_DIR inicializációs paraméterben azt, hogy az állományrendszerben hol helyezkedjenek el majd a kódokhoz tartozó osztott könyvtárak. Több ezer lefordított állomány esetén javasolt azok alkönyvtárakba szervezése és a

PLSQL_NATIVE_LIBRARY_SUBDIR_COUNT paraméter használata. A könyvtárakat az operációs rendszerben kézzel kell létrehozni, megfelelő védelmet biztosító beállításokkal. Ezekről a beállításokról és az adatbázis összes PL/SQL kódjának natívra fordításáról bővebben *lásd* [19] 11. fejezet és [17].

A C kódból az osztott könyvtár létrehozásához meghívandó parancsokat tartalmazó sablon a $ORACLE_HOME/plsql/spnc_commands állomány. Ebben kell ellenőrizni és szükség esetén módosítani a parancsok elérési útjait.

Fordítsuk le a 16.2. alfejezet 1. példájában megismert PROC_SZAMITAS eljárást és hasonlítsuk össze a futási időket.

Példa

```
CREATE OR REPLACE PROCEDURE proc_szamitas(

p_Iter PLS_INTEGER

) IS

a PLS_INTEGER;

b PLS_INTEGER;

c PLS_INTEGER;

BEGIN

FOR i IN 1..p_Iter

LOOP

a := i+1;

b := i-2;

c := b-a+1;

d := b-a-1; -- ismétlődő kifejezés előfordulása

END LOOP;

END proc_szamitas;

/
```

Az eljárást négy különböző módon újrafordítottuk majd futtattuk:

```
ALTER PROCEDURE proc_szamitas COMPILE

PLSQL_DEBUG=FALSE

PLSQL_OPTIMIZE_LEVEL={2|0}

PLSQL_CODE_TYPE={INTERPRETED|NATIVE}

;

SET TIMING ON

EXEC proc_szamitas(10000000);

SET TIMING OFF;
```

A futási idők a következőképpen alakultak:

```
2I - PLSQL_OPTIMIZE_LEVEL=2 PLSQL_CODE_TYPE=INTERPRETED

Eltelt: 00:00:03.05
```

```
2N - PLSQL_OPTIMIZE_LEVEL=2 PLSQL_CODE_TYPE=NATIVE

Eltelt: 00:00:01.82

0I - PLSQL_OPTIMIZE_LEVEL=0 PLSQL_CODE_TYPE=INTERPRETED

Eltelt: 00:00:05.78

0N - PLSQL_OPTIMIZE_LEVEL=0 PLSQL_CODE_TYPE=NATIVE

Eltelt: 00:00:03.33
```

A natívra fordított kód gyorsabb volt az interpretált párjánál, a legjobb eredményt az optimalizálás és a natív fordítás együtt adta.

6. Tanácsok a hangoláshoz és a hatékonyság növeléséhez

Ebben a részben a hatékony kód írásához adunk néhány rövid tanácsot.

Főbb hangolási alapelvek

1. *Mérjünk*. Mérjünk hangolás előtt, hogy legyen referenciánk.

2. *Tűzzük ki a célt, amivel elégedettek leszünk.* Próbáljuk felmérni, hogy ez a teljesítménynövekedés mennyi munkát igényel és döntsünk, hogy megéri-e.

3. *Keressük meg a legszűkebb keresztmetszetet.* A *Pareto-elv*, másik nevén a *80/20 szabály* szerint általánosságban egy rendszerben a tényezők egy kisebbik része (körülbelül 20%) felelős a jelenségek nagyobbik részéért (körülbelül 80%). Ezt hatékonyságra úgy fordíthatjuk le, hogy a futási idő 80%-ában a szerver a kód ugyanazon 20%-át futtatja. Az elv iteratív alkalmazásával már 64/4 arányt kapunk. A hangsúly tulajdonképpen nem a pontos számokon van, jelentése az, hogy a megfelelő helyen történő kis módosítással is nagy eredményt érhetünk el.

4. *Egyszerre egy módosítást végezzünk.* Az összetett hatásokat megjósolni és mérni is nehezebb. A mérési eredmény rossz értelmezése viszont félrevezető lehet.

5. *A módosítás hatását jósoljuk meg, állítsunk fel hipotézist.* Ez könnyű, hiszen azért módosítunk, mert várunk tőle valamit. Ezt az elvárást tudatosítsuk, pontosítsuk.

6. *Mérjünk.* Mérjünk minden egyes módosítás után, hogy a hipotézisünket bizonyítsuk, vagy cáfoljuk. Ezzel egyben a módosítás hatására bekövetkező teljesítményváltozást is dokumentáljuk.

7. A fejlesztés során minél később gondolunk a teljesítményre, annál költségesebb a hangolás.

8. Ha a fejlesztés során túl korán kezdünk el a teljesítménnyel foglalkozni az a tervezést félreviheti, az architektúra túlzott megkötését eredményezheti.

A méréshez és a szűk keresztmetszet kereséséhez használhatjuk a legegyszerűbb eszközöket, mint a SET TIMING parancs, vagy a DBMS_UTILITY.GET_TIME függvény, de komolyabb eszközök is rendelkezésre állnak, mint a DBMS_PROFILER, DBMS_TRACE csomagok, a dinamikus teljesítménynézetek (V$SESSTAT, V$SYSSTAT stb.), vagy a tágabb értelemben vett adatbázis szintű optimalizálást támogató STATSPACK programcsomag, a beépített Automatic Workload Repository (AWR) vagy az SQL trace eszközrendszer.

Ezek az eszközök minden Oracle-adatbázisnál megtalálhatók.

Tanácsok hangolási tanácsok megfogadásához

1. *Egészséges kételkedés* – Minden hangolási tanácsot fenntartással kezeljünk. Az Oracle adatbázis-kezelő folyamatos fejlesztés alatt áll. Ami egy korábbi verzióban igaz lehetett, az már nem biztos, hogy megállja a helyét. Az internetes fórumok is bizonytalan forrásnak tekintendők.

2. *Bizonyosság* – A tanácsokat igyekezzünk egyszerű mérésekkel ellenőrizni mielőtt teljesen megbíznánk bennük.

Hatékonyságnövelő tanácsok

1. *Hangoljuk a kódba ágyazott SQL utasításokat.* A PL/SQL kódban kiadott SQL utasításokat (akár statikusak, akár dinamikusak) az SQL motor hajtja végre ugyanúgy, mintha azt más nyelvből adtuk volna ki. Az SQL utasítások hangolása nem triviális feladat, a hivatalos dokumentáció itt is jó kiindulási pont *lásd* [17].

2. *Használjuk a RETURNING utasításrészt DML utasításoknál a sor adatainak visszaolvasására.* Beszúrásnál például a beszúrt sor egészét visszaolvashatjuk, nem kell az oszlopok adatait megadó kifejezéseket a beszúrás előtt változókban tárolni, a nem explicit módon inicializált oszlopértékeket utólag lekérdezni.

```
SELECT t_seq.NEXTVAL

INTO v_Id

FROM dual;

INSERT INTO t(id, oszlop_explicit_ertekkel)

VALUES(v_Id, 'ERTEK');

SELECT oszlop_default_ertekkel

FROM t

INTO v_Mi_lett_az_erteke

WHERE id = v_Id;

-- helyette

INSERT INTO t(id, oszlop_explicit_ertekkel)

VALUES(t_seq.NEXTVAL, 'ERTEK')

RETURNING id, oszlop_default_ertekkel

INTO v_Id, v_Mi_lett_az_erteke;
```

3. *Használjunk együttes hozzárendelést.* Ciklusban elhelyezett DML utasításoknál a FORALL különböző változatait, míg SELECT, FETCH és RETURNING esetén a BULK COLLECT utasításrészt használhatjuk statikus és dinamikus SQL esetén egyaránt. Láthattuk, hogy az optimalizáló már megtesz bizonyos lépéseket ennek automatizálásában, ugyanakkor még mindig az explicit együttes hozzárendelés teljesítménye a legjobb, ezenfelül a kód így jobban is érthető. Hátránya, hogy több kódolással jár, deklarálnunk kell a változókat és esetleg a típusokat is, a végrehajtható kód is hosszabb lesz. A nagyobb méretű, bonyolultabb kód nehezebben olvasható és tartható karban.

4. *Ha lehet, használjunk SQL-t PL/SQL helyett (!).* Ez igen furcsa tanács egy PL/SQL-ről szóló könyvben, mégis helytálló. A PL/SQL egy imperatív nyelv, ezért majdnem mindent meg tudunk benne oldani egytáblás SQL utasításokkal és ciklusokkal is. A kezdő PL/SQL programozó hajlamos ezeket az imperatív eszközöket túlzott mértékben használni, előnyben részesíteni az SQL megfelelőjükkel szemben. Az Oracle minden új verziójában az SQL nyelv nagyon sok olyan új eszközzel gazdagodott, melyek a nyelv kifejezőerejét és hatékonyságát is növelik. Ezért PL/SQL programozóként törekedjünk az SQL minél több lehetőségének megismerésére, és mindig tegyük fel a kérdést, hogy melyik világ eszközeit tudnánk az adott helyzetben hatékonyabban, szebben, a célnak leginkább megfelelő módon használni.

Néhány fontosabb eset, ahol a PL/SQL kód sokszor helyettesíthető SQL használatával:

– Szűrjünk SQL-ben. A WHERE korábban szűr, mint egy IF. A felesleges sorok nem kerülnek betöltésre és az SQL utasítás végrehajtási terve is javulhat.

– Kurzor ciklusába ágyazott lekérdezéseknél a kurzor SELECT-je és a magban szereplő lekérdezés esetlegesen egyetlen SELECT-be is összevonható allekérdezések és összekapcsolások használatával.

```
FOR v_Konyv IN (
SELECT id, cim FROM konyv
) LOOP
FOR v_Kocsonzo IN (
SELECT kolcsonzo FROM kolcsonzes WHERE konyv = v_Konyv.id
) LOOP
.
.
.
END LOOP;
END LOOP;
-- helyette
FOR v_Konyv_kolcsonzo IN (
SELECT kv.id, kv.cim, ko.kolcsonzo
FROM konyv kv, kolcsonzes ko
WHERE ko.konyv = kv.id
) LOOP
.
.
.
END LOOP;
```

Itt még tovább tudnánk növelni a teljesítményt együttes hozzárendelés alkalmazásával.

Hátránya akkor van, ha az új SELECT végrehajtása lassabb lesz a nagyobb bonyolultság miatt, vagy ha az így létrejövő hosszabb sorok jelentősen nagyobb adatmozgást eredményeznek.

– Kurzor ciklusába ágyazott DML utasításoknál a ciklus és a DML utasítás összevonható egyetlen MERGE utasításba. A két művelet közötti korreláció a MERGE feltételeivel megfogalmazható: az esetleges ciklusbeli IF feltételek a USING ON feltételévé, vagy a MERGE ágainak WHEN feltételeivé alakíthatók.

Hátránya, hogy a tömören fogalmazó MERGE nem mindenki számára olvasható könnyen, és ha a ciklusban több független DML művelet is van, akkor a ciklus kiemelő hatása általában erősebb, a ciklus használata ilyenkor jellemzően elegánsabb és hatékonyabb kódot eredményez.

– Néha a több ciklussal összekombinált lekérdezések, esetleg a ciklusmagba ágyazott elágaztató utasítások lényegében halmazműveleteket valósítanak meg. Ilyenkor gondoljunk az SQL INTERSECT, UNION, UNION ALL, MINUS, NOT EXISTS, IN stb. eszközeire.

– Használjuk SQL-ben az analitikus függvényeket, aggregálásnál a KEEP utasításrészt. Nagyon gyakran lehet sok-sok lekérdezést megspórolni ezekkel.

– Használjuk ki a SELECT lehetőségeit, az új csoportképző eszközöket, a CONNECT BY, MODEL utasításrészeket stb.

– Használjuk a feltételes INSERT-et olyan kurzort használó ciklus helyett, ahol a magban bizonyos feltételektől függően végzünk beszúrást egy vagy több táblába.

– Néha ciklusok helyett használhatunk sorszámokat és egyéb adatokat generáló lekérdezéseket, amiket aztán más utasításokban alkérdésként használhatunk.

```
FOR i IN 1..100

LOOP

INSERT INTO t(sorszam, paros_paratlan)

VALUES(i, DECODE(MOD(ROWNUM,2),0, 'páros', 'páratlan'))

;

END LOOP;

-- helyette

INSERT INTO t(sorszam, paros_paratlan)

SELECT ROWNUM sorszam

, DECODE(MOD(ROWNUM,2),0, 'páros', 'páratlan') paros_paratlan

FROM dual

CONNECT BY LEVEL <= 100

;
```

– Használjuk DML utasításoknál a hibanaplózás funkciót ahelyett, hogy minden utasítás előtt explicit ellenőriznénk a DML utasítás előfeltételeit, vagyis azt, hogy a művelet nem fog-e sérteni megszorítást a konkrét sor esetében. Így összevonhatunk több műveletet esetleg együttes hozzárendelés használatával. A hibanaplót aztán elemezzük szükség esetén és csak a hibás sorokra hajtsuk végre újra a műveleteket. Használjuk ezt a lehetőséget a FORALL utasítás SAVE EXCEPTIONS utasításrészéhez hasonlóan.

5. SQL-ben használt függvényeknél, ha lehet, adjuk meg a DETERMINISTIC kulcsszót. Ilyenkor az SQL motor egy gyorsítótáblában tárolja a már kiszámolt értékeket, és ezzel függvényhívást takaríthat meg. WHERE feltételben szereplő oszlopra hivatkozó determinisztikus függvénykifejezések esetén használhatunk függvényalapú indexet, ilyenkor a függvényhívás a beszúrás vagy módosítás során történik meg, nem a lekérdezés végrehajtásakor.

6. SQL-ben szereplő függvényhívások vagy bonyolult kifejezések esetén törekedjünk a hívások/kiértékelések számának minimalizálására. Ha a függvény oszlopra hivatkozik, használjunk alkérdést, mert az csökkentheti a hívás alapjául szolgáló sorok számát, így a kiértékelések számát.

```
SELECT DISTINCT SQRT(x) FROM ...

-- helyette

SELECT SQRT(distinct_x)

FROM (SELECT DISTINCT x AS distinct_x FROM ... )
```

Vegyük észre, hogy a fenti példában alkalmazott átalakítás nem tehető meg automatikusan,

nem lenne azonos átalakítás például MOD(x, k) hívás mellett.

7. Nagyméretű paraméterek átadásakor fontoljuk meg a NOCOPY használatát. Nagyméretű típus lehet rekord, kollekció, objektum, LOB, karakteres típus. A NOCOPY használható objektumtípusok metódusaiban is.

A későbbiekben viszont vegyük figyelembe a megváltozott paraméterátadás módjából adódó viselkedést.

8. Logikai operátorokkal képzett bonyolult kifejezésekben használjuk ki a rövidzár kiértékelést. Vegyük figyelembe a részkifejezések költségét és a rövidzárt kiváltó eredményük valószínűségét.

9. Kerüljük az adatkonverziót, főleg az implicit adatkonverziót. Használjunk megfelelő típusú literálokat és változókat.

10. A számítások eredményeit gyorsítás céljából őrizzük meg. Használhatunk lokális változókat, kollekciókat, ideiglenes vagy normál táblákat. A csomagban deklarált változók a munkameneten belül hívások közt is megtartják értéküket. Gyorsítótár gyanánt deklarálhatunk egy csomagban karakteres indexű asszociatív tömböket, amik használhatók kulcs-érték párok memóriában való tárolására. Ilyen kollekcióknál figyeljünk oda a memóriahasználatra.

```
.

.

.

TYPE t_num_tab IS TABLE OF NUMBER INDEX BY VARCHAR2(1000);

v_Cache_tab t_num_tab;

.

.

.

FUNCTION fn(p1 NUMBER, p2 VARCHAR2) RETURN NUMBER

IS

.

.

.

FUNCTION cached_fn(p1 NUMBER, p2 VARCHAR2) RETURN NUMBER

IS

v_Key VARCHAR2(1000);

rv NUMBER;

BEGIN

v_Key := p1 || '#' || p2;

IF v_Cache_tab.EXISTS(v_Key) THEN

rv := v_Cache_tab(v_Key);

ELSE

rv := fn(p1, p2);

v_Cache_tab(v_Key) := rv;

END IF;

RETURN rv;

END cached_fn;

.

.

.
```

A példa csak az idiómát mutatja be, a memóriahasználatra nem ügyel, és nem is elég általános.

- Számításoknál használjunk megfelelő típust:

– Használjunk PLS_INTEGER típust egész számításokhoz.

– Korlátozott altípusok használata felesleges ellenőrzésekkel járhat a számítások során, ilyen típusok például INTEGER, NATURAL, NATURALN, POSITIVE, POSITIVEN.

– Használjunk BINARY_FLOAT vagy BINARY_DOUBLE típust valós számításokhoz. Üzleti számításoknál ez a megkövetelt pontosság miatt sokszor nem jó megoldás, mert kettes számrendszerben már egy tized (1/10) is végtelen bináris tört, így a konverziók során pontosságot veszíthetünk.

```
DECLARE

a BINARY_DOUBLE := 1;

b BINARY_DOUBLE := 10;

BEGIN

DBMS_OUTPUT.PUT_LINE('Egy tized: ' || TO_NUMBER(a/b));

END;

/

-- Egy tized: ,10000000000000001
```

- Lokális VARCHAR2 típusú változók hosszát válasszuk elég nagyra a futási idejű hibák megelőzése céljából. Egy VARCHAR2 típusú változó memóriában lefoglalt mérete mindig az aktuális értéktől függ. A nagyon megszorító korlátozásnak általában akkorvan értelme, ha a változó adatbázisbeli tábla oszlopának beszúrás előtti értékét tárolja. Ilyenkor használjunk %TYPE attribútumot.

- A hatékonyabb memóriahasználat és a függőségek egyszerűsítése érdekében a logikailag összetartozó alprogramokat tegyük egy csomagba. Ha egy csomag egy alprogramját meghívjuk, az egész csomag betöltődik a memóriába. Ha ezután egy ugyanebben a csomagban definiált alprogramot hívunk meg, már annak a kódja is a memóriában lesz.

- A munkamenetek által visszatérően és elég gyakran használt nagyméretű csomagokat rögzítsük a memóriába a DBMS_SHARED_POOL csomaggal. Az ilyen csomag kódja mindig a memóriában, az osztott tartományban marad, függetlenül az osztott tartomány telítődésétől és a csomag használatának sűrűségétől.

- Ne használjuk a COMMIT parancsot túl gyakran és fölöslegesen. Ha tehetjük, tegyük a COMMIT-ot cikluson kívülre, vagy cikluson belül valamilyen gyakorisággal hajtsuk csak végre. Esetleg használjuk a COMMIT WRITE BATCH NOWAIT utasítást, ha elfogadható a működése.

- Memóriában rendezhetünk asszociatív tömbbel. Ha egy asszociatív tömbbe elemeket szúrunk be és az elemeken a FIRST, NEXT vagy a LAST, PRIOR párokkal iterálunk, a kulcsokon rendezetten haladunk végig. Karakteres kulcsok esetén vegyük figyelembe, hogy a rendezést az NLS_COMP és az NLS_SORT inicializációs paraméterek befolyásolják, a rendezés a kis- és nagybetűkre is érzékeny.

```
ALTER SESSION SET NLS_COMP=ANSI;

DECLARE

s VARCHAR2(100);

TYPE t_karakteres_kulcsok IS

TABLE OF BOOLEAN INDEX BY s%TYPE;

v_Rendezo_tabla t_karakteres_kulcsok;

BEGIN
```

```
v_Rendezo_tabla('öszvér') := TRUE;

v_Rendezo_tabla('ló') := TRUE;

v_Rendezo_tabla('szamár') := TRUE;

s := v_Rendezo_tabla.FIRST;

WHILE s IS NOT NULL

LOOP

DBMS_OUTPUT.PUT_LINE(s);

s := v_Rendezo_tabla.NEXT(s);

END LOOP;

END;

/
```

- Natív dinamikus SQL használatával meghívhatunk egy a nevével paraméterként átvett alprogramot (callback function). Ehelyett objektumtípust és késői kötést is alkalmazhatunk. A hívás így sokkal gyorsabb és ellenőrzöttebb lesz.

```
CREATE OR REPLACE PROCEDURE proc(p_Number_fuggveny VARCHAR2)

IS

v_Num NUMBER;

BEGIN

EXECUTE IMMEDIATE 'BEGIN :x := ' || p_Number_fuggveny || '; END;'

USING OUT v_Num;

DBMS_OUTPUT.PUT_LINE('Num: ' || v_Num);

END proc;

/

CREATE OR REPLACE FUNCTION a_number_fuggveny_1 RETURN NUMBER

IS

BEGIN

RETURN 1;

END a_number_fuggveny_1;

/

SET SERVEROUTPUT ON FORMAT WRAPPED

EXEC proc('a_number_fuggveny_1');

-- Num: 1

--------------

-- helyette --

--------------

CREATE OR REPLACE TYPE T_Obj1 IS OBJECT (
```

```
attr CHAR(1),
MEMBER FUNCTION a_number_fuggveny RETURN NUMBER
)
NOT FINAL NOT INSTANTIABLE
/
CREATE OR REPLACE TYPE T_Obj2 UNDER T_Obj1(
CONSTRUCTOR FUNCTION T_Obj2 RETURN SELF AS RESULT,
OVERRIDING MEMBER FUNCTION a_number_fuggveny RETURN NUMBER
)
/
CREATE OR REPLACE TYPE BODY T_Obj2 IS
CONSTRUCTOR FUNCTION T_Obj2 RETURN SELF AS RESULT
IS BEGIN RETURN; END;
OVERRIDING MEMBER FUNCTION a_number_fuggveny RETURN NUMBER
IS
BEGIN
RETURN 1;
END a_number_fuggveny;
END;
/
CREATE OR REPLACE PROCEDURE proc(p_Obj T_Obj1)
IS
v_Num NUMBER;
BEGIN
v_Num := p_Obj.a_number_fuggveny;
DBMS_OUTPUT.PUT_LINE('Num: ' || v_Num);
END proc;
/
SET SERVEROUTPUT ON FORMAT WRAPPED
EXEC proc(T_Obj2());
-- Num: 1
```

- Explicit OPEN utasítással megnyitott kurzor esetén a használó blokk kivételkezelőjének valamennyi ágában biztosítsuk, hogy a kurzor véletlenül se maradhasson nyitva. A nyitott kurzorok száma adatbázis példányonként limitált, telítettsége kényszerleálláshoz is vezethet. Kurzor FOR ciklus esetén erre nincs szükség, a kurzor automatikusan kerül megnyitásra és lezárásra, függetlenül attól, hogy a ciklus szabályosan vagy kivétellel fejeződik-e be.

```
BEGIN
```

```
.

.

.

OPEN cur;

.

.

.

CLOSE cur;

.

.

.

EXCEPTION

WHEN kivétel THEN

IF cur%ISOPEN THEN CLOSE cur; END IF;

… -- Kivétel kezelése

WHEN OTHERS THEN

IF cur%ISOPEN THEN CLOSE cur; END IF;

RAISE; -- Kivétel továbbadása a kurzor lezárása után.

END;
```

- Készítsük fel a programjainkat arra, hogy támogassák az előre nem látott hibák könnyű lokalizálását és javítását. Naplózzuk, vagy jelentsük a bekövetkezett kivételeket. Ha máshol nem, legalább a legkülső hívó programegység szintjén tegyük ezt meg. A hívási láncról és a bekövetkezett hibákról, azok helyéről alapvető információkkal szolgálnak a DBMS_UTILITY csomag FORMAT_CALL_STACK, FORMAT_ERROR_BACKTRACE és FORMAT_ERROR_STACK függvényei. A hibanapló bejegyzéseinek méretén ne spóroljunk, több információ mellett kevesebb idő alatt megtalálhatjuk a hiba valódi okát.

- A kód átláthatóságát növeli és a „copy-paste” hibák előfordulását csökkenti az, ha az alprogramokat, triggereket és csomagokat záró END utasításban használjuk a programegység nevét.

- Értelmezzük a fordító figyelmeztetéseit, a kód javításával törekedjünk azok megszüntetésére.

- Ismerjük meg a szerver alapcsomagjait. Sokszor olyan műveletekre is van már alapcsomagban eszköz, amire nem is gondolnánk: mátrixszámításokhoz hatékony implementációt tartalmaz az UTL_NLA, webszolgáltatások használatát támogatja az UTL_DBWS, időzíthetünk költséges batch műveleteket a DBMS_JOB segítségével stb.

- Ismerjük meg és használjuk a beépített SQL függvényeket. Használjuk ezeket ahelyett, hogy saját magunk újraimplementálnánk a funkcionalitásaikat. A meglevő függvények alapos teszteléseken mentek át, megbízhatóan működnek, sok esetben hatékonyabbak, mivel az implementációjuk gyakran C nyelvű, ilyen hatékony kódot PL/SQL-ben nem is tudnánk írni. Különösen igaz ez a karakteres függvényekre.

További tanácsok és ajánlások a dokumentációban (*lásd* [19] 11. fejezet) és az interneten

(*lásd* [26], [27]) találhatók.

A. függelék - A PL/SQL foglalt szavai

A függelékben felsorolt szavak a PL/SQL foglalt szavai. A foglalt szavak a nyelvben speciális szintaktikai jelentéssel bírnak, s mint ilyenek nem használhatók fel azonosítóként (változók, eljárások stb. elnevezésére). A szavak egy része egyben SQL foglalt szó is, azaz nem nevezhetünk így adatbázisbeli objektumokat (táblák, nézetek, szekvenciák stb.).

Alább felsoroljuk a PL/SQL foglalt szavait az Oracle10*g* verzióval bezárólag. Az Oracle8*i* és Oracle9*i* verziókból számos foglalt szót töröltek, mivel azok feleslegessé váltak. Azonban egyes korábbi verziókban ezek foglalt szavak voltak. Természetesen azonban több új szó bekerült a foglalt szavak közé. A következő felsorolás az Oracle10*g* foglalt szavait vastagon szedve tartalmazza. Azon szavak mellett, amelyek egyben az SQL foglalt szavai is, egy csillag (*) szerepel. A normál vastagsággal szedett szavak csak a korábbi verziókban voltak foglalt szavak. Ennek ellenére tanácsos kerülni ezek használatát az esetleges problémák elkerülése végett.

Ugyanígy kerülendő olyan azonosítók használata, amelyek megegyeznek valamely beépített csomag nevével, vagy valamely STANDARD csomagbeli alprogram nevével.

ABORT	ACCEPT	ACCESS*
ADD*	ALL*	ALTER*
AND*	ANY*	ARRAY
ARRAYLEN	AS*	ASC*
ASSERT	ASSIGN	AT
AUDIT*	AUTHID	AUTHORIZATION
AVG	BASE_TABLE	BEGIN
BETWEEN*	BINARY_INTEGER	BODY
BOOLEAN	BULK	BY*
CASE	CHAR*	CHAR_BASE
CHECK*	CLOSE	CLUSTER*
CLUSTERS	COALESCE	COLAUTH
COLLECT	COLUMN*	COMMENT*
COMMIT	COMPRESS*	CONNECT*
CONSTANT	CONSTRUCTOR	CRASH
CREATE*	CURRENT*	CURRVAL
CURSOR	DATABASE	DATA_BASE
DATE*	DAY	DBA
DEBUGOFF	DEBUGON	DECIMAL*

DECLARE	DEFAULT*	DEFINITION
DELAY	DELETE*	DESC*
DIGITS	DISPOSE	DISTINCT*
DO	DROP*	ELSE*
ELSIF	END	ENTRY
EXCEPTION	EXCEPTION_INIT	EXCLUSIVE*
EXECUTE	EXISTS*	EXIT
EXTENDS	EXTRACT	FALSE
FETCH	FILE*	FLOAT*
FOR*	FORALL	FORM*
FROM*	FUNCTION	GENERIC
GOTO	GRANT*	GROUP*
HAVING*	HEAP	HOUR
IDENTIFIED	IF	IMMEDIATE*
IN*	INCREMENT*	INDEX*
INDEXES	INDICATOR	INITIAL*
INSERT*	INTEGER*	INTERFACE
INTERSECT*	INTERVAL	INTO*
IS*	ISOLATION	JAVA
LEVEL*	LIKE*	LIMITED
LOCK*	LONG*	LOOP
MAX	MAXEXTENTS	MIN
MINUS*	MINUTE	MLSLABEL*
MOD	MODE*	MODIFY
MONTH	NATURAL	NATURALN
NEW	NEXTVAL	NOAUDIT*
NOCOMPRESS*	NOCOPY	NOT*

NOWAIT*	NULL*	NULLIF
NUMBER*	NUMBER_BASE	OCIROWID
OF*	OFFLINE*	ON*
ONLINE*	OPAQUE	OPEN
OPERATOR	OPTION*	OR*
ORDER*	ORGANIZATION	OTHERS
OUT	PACKAGE	PARTITION
PCTFREE*	PLS_INTEGER	POSITIVE
POSITIVEN	PRAGMA	PRIOR*
PRIVATE	PRIVILIGES*	PROCEDURE
PUBLIC*	RAISE	RANGE
RAW*	REAL	RECORD
REF	RELEASE	REMR
RENAME*	RESOURCE*	RETURN
REVERSE	REVOKE*	ROLLBACK
ROW*	ROWID*	ROWLABEL*
ROWNUM*	ROWS*	ROWTYPE
RUN	SAVEPOINT	SCHEMA
SECOND	SELECT*	SEPARATE
SESSION*	SET*	SHARE*
SIZE	SMALLINT*	SPACE
SQL	SQLCODE	SQLERRM
START*	STATEMENT	STDDEV
SUBTYPE	SUCCESSFUL*	SUM
SYNONYM*	SYSDATE*	TABAUTH
TABLE*	TABLES*	TASK
TERMINATE	THEN*	TIME

TIMESTAMP	TIMEZONE_ABBR	TIMEZONE_HOUR
TIMEZONE_MINUTE	TIMEZONE_REGION	TO*
TRIGGER*	TRUE	TYPE
UI	UID*	UNION*
UNIQUE*	UPDATE*	USE
USER*	VALIDATE*	VALUES*
VARCHAR*	VARCHAR2*	VARIANCE
VIEW*	VIEWS	WHEN
WHENEVER*	WHERE*	WHILE
WITH*	WORK	WRITE
XOR	YEAR	ZONE

B. függelék - A mellékelt CD használatáról

A CD állományai a könyvben szereplő példákat tartalmazzák könyvtárakba rendezve, az egyes fejezeteknek megfelelően.

Az állományok nevei utalnak a fejezetre, amelyben előfordulnak. Számozásuk az alfejezetek számozását követi. Mivel nem minden fejezetben van ugyanolyan mélységben alfejezet, ezért az elnevezések során az elsődleges cél az azonosíthatóság mellett az volt, hogy az állományok alfabetikus rendezése minél jobban kövesse a példák előfordulási sorrendjét. Csomagok forrásai esetén ettől a számozástól eltekintettünk.

A könyvtárakban található néhány olyan segédállomány is, amelyek a könyvben nem szereplő kódot tartalmaznak (például séma és felhasználó létrehozása). A számozástól ezek nevénél is eltekintettünk.

Az egyes állományok felhasználási módját a kiterjesztésük szabja meg:

- Az SQL kiterjesztésű állományok futtathatók SQL*Plusban.
- Az SQL_TXT kiterjesztésű állományok tartalmaznak SQL kódot, de nem futtathatók a bennük szereplő szöveg miatt.
- A többi kiterjesztés szöveges állományt vagy generált kimenetet jelöl.

A mellékelt állományok használatával az olvasó megszabadulhat a gépelés fáradalmaitól, viszont ne feledje, hogy a kódok begépelése segíti a tanultak elmélyítését, a valódi mély tudás megszerzéséhez pedig elengedhetetlen saját kódok írása. Mindazonáltal reméljük, hogy a példaállományok segítenek a PL/SQL nyelv minél alaposabb megismerésében.

Irodalomjegyzék

Abbey, Michael, Corey, Michael C., és Abramson, Ian. *ORACLE8i Kézikönyv kezdőknek*. Panem. Budapest. 2001.

Gábor, András és Juhász, István. *PL/SQL-programozás. Alkalmazásfejlesztés ORACLE 9iben*. Panem. Budapest. 2002.

Gábor, András, Gunda, Lénárd, Juhász, István, Kollár, Lajos, Mohai, Gábor, és Vágner, Anikó. *Az Oracle és a WEB. Haladó Oracle9i ismeretek*. Panem. Budapest. 2003.

Horowitz, Ellis. *Magasszintű programnyelvek*. Műszaki Könyvkiadó. Budapest. 1987.

Kende, Mária, Kotsis, Domokos, és Nagy, István. *Adatbázis-kezelés az Oracle-rendszerben*. Panem. Budapest. 2002.

Kende, Mária és Nagy, István. *ORACLE példatár SQL, PL/SQL*. Panem. Budapest. 2005.

Loney, Kevin és Koch, George. *Oracle8i Teljes referencia*. Panem. Budapest. 2001.

Loney, Kevin. *Oracle Database 10g Teljes referencia*. Panem. Budapest. 2006.

Melton, Jim és Simon, Alan R.. *SQL:1999*. Morgan Kaufmann Publishers. San Francisco, Budapest. 1998.

Nyékiné, Gaizler Judit (szerk). *JAVA 2 Útikalauz programozóknak*. ELTE TTK Hallgatói Alapítvány. Budapest. 2000.

Nyékiné, Gaizler Judit (szerk.). *Programozási nyelvek*. Kiskapu. Budapest. 2003.

Pyle, Ian C.. *Az ADA programozási nyelv*. Műszaki Könyvkiadó. Budapest. 1987.

Urman, Scott, Hardman, Ron, és , McLaughlin, Michael. *ORACLE DATABASE 10g PL/SQL Programming*. McGraw-Hill/Osborne. New York. 2004.