

PL/SQL ALAPOZÓ TANFOLYAM

1.1 verzió

2017

Készítette:

Ender János
NAV Informatikai Intézet Pillér Kft.

Lektor:

Parditkáné Jakubovics Erzsébet
NAV Informatikai Intézet

Lezárva:
Budapest, 2015. március 15.

Tartalomjegyzék

I. Bevezetés.....	5
A könyv jelölései	5
II. PL/SQL blokkok.....	6
II. 1. Névtelen blokk.....	6
II. 2. Tárolt eljárás	7
II. 3. Függvény	7
II. 4. Paraméterek átadásának módjai.....	9
III. Konstansok, változók	11
III. 1. PL/SQL változók deklarálása	11
III. 2. Alaptípusok.....	11
III. 3. Összetett típusok.....	12
Rekord.....	12
Kollekció.....	13
IV. Értékadás változóknak.....	16
IV. 1. Értékadás.....	16
IV. 2. A változók érvényességi köre	17
3. Operátorok	18
V. Vezérlési szerkezetek	18
V. 1. Feltételes elágazás.....	18
V. 2. Ciklus	19
V. 3. Feltételes ciklus.....	19
V. 4. Számláló ciklus	20
V. 5. Beágyazott ciklusok	20
VI. Kurzorok	21
VI. 1. Implicit kurzor.....	21
VI. 2. Explicit kurzor.....	21
FOR használata explicit kurzornál (gyorsabb)	22
FOR használata allekérdéssel	22
FOR UPDATE utasításrész.....	23
WHERE CURRENT OF utasításrész.....	23
VI. 3. Kurzorváltozók.....	23
VI. 4. Tömb feltöltése kurzor segítségével.....	23
VII. Kivételek.....	26
VII. 1. Kivételek elfogása.....	26
VII. 2. Gyakori kivételek.....	26
VII. 3. Saját kivételek.....	28
VII. 4. RAISE_APPLICATION_ERROR eljárás	28
VIII. Csomagok, alprogramok	29
VIII. 1. Csomag létrehozása	29
VIII. 2. Csomagváltozók	29

VIII. 3. Törzs nélküli csomag.....	30
VIII. 4. Előre deklarálás	31
VIII. 5. Egyszer végrehajtódó kódrészlet.....	32
VIII. 6. Többjelentésű programok.....	32
IX. Programok és jogosultságok	33
IX. 1. Programok készítéséhez szükséges jogosultságok.....	33
IX. 2. A programok hozzáférése adatokhoz	33
X. Tranzakció kezelés	34
X. 1. Normál tranzakciók	34
X. 2. Autonóm tranzakciók.....	35
XI. Triggerek	35
XI. 1. Alkalmazás triggerek.....	35
Minősített nevek	36
XI. 2. Adatbázis triggerek.....	37
XII. PL/SQL programegységek újrafordítása	39
XIII. Dinamikus SQL	39
XIV. Gyári csomagok.....	40
XIV. 1. Kiírás képernyőre.....	40
XIV. 2. Fájl kezelés.....	40
XIV. 3. JOB kezelés.....	40
A job ütemező használata	41
Operációs rendszerbeli programok futtatása	42
Job paraméterek változtatása	42
Job kézi futtatása	43
Futó job leállítása.....	43
Job letiltása, engedélyezése	43
Job eldobása.....	44
XIV. 4. Nagy méretű objektumok.....	44
XIV. 5. Hiba kiírás a DBMS_UTILITY csomaggal.....	47
XV. ORACLE adatszótárbeli nézetek	49
XV. 1. USER_OBJECTS.....	49
XV. 2. USER_SOURCE.....	49
XV. 3. USER_ERRORS	50
XV. 4. USER_DEPENDENCIES.....	50
XV. 5. USER_TRIGGERS	51
XV. 6. USER_TABLES.....	51
XV. 7. USER_VIEWS.....	51
XV. 8. USER_TAB_COLUMNS	51
XV. 9. USER_TAB_PRIVS.....	52
XV. 10. USER_SYS_PRIVS.....	52
XV. 11. USER_ROLE_PRIVS.....	52
IRODALOMJEGYZÉK / FELHASZNÁLT IRODALOM	53

I. Bevezetés

Az SQL nem algoritmikus nyelv, hiszen nem tartalmaz algoritmikus szerkezeteket, illetve nem tartalmaz normál fájlkezelésre vagy felhasználói képernyőkezelésre vonatkozó utasításokat, kimondottan az adatbázis-kezelés, adatkezelés megvalósítására szolgál.

Procedural Language/SQL. Ahogy a neve is mutatja, a szabvány SQL elemeit egy hagyományos programozási nyelv keretei közé emeli be.

A PL/SQL az SQL kiegészítése algoritmikus elemekkel. A PL/SQL nyelv segítségével olyan programkódok készíthetők, melyekbe közvetlenül beépíthetők SQL utasítások, valamint a PL/SQL programok, függvények felhasználhatók SQL utasításokban. Az SQL*PLUS is kezeli a PL/SQL utasításokat.

Az Oracle szerver oldali motorokkal valósítja meg a szétválasztást. A PL/SQL motor dolgozza fel a PL/SQL utasításokat, az SQL motor kapja meg a végrehajtandó SQL utasításokat. Ha a kliens oldalon történik az algoritmikus nyelvi feldolgozás, (forms, reports, C, Java nyelvű programok) akkor ezek csak az SQL motort használják.

A könyv jelölései

A szövegek „Times New Roman” betűtípussal íródtak, a példákban és a magyarázatokban szereplő utasítások, eredmény táblázatok „Courier New” betűtípussal készültek. Ez egyrészt segíti az egyes részek megkülönböztetését, másrészt a „Courier” betűtípus azonos betűszélessége miatt biztosítja a tagolás jó láthatóságát.

Az utasításokban a kulcsszavak vastag nagybetűkkel szerepelnek, a konkrét objektum nevek normál nagy- vagy kisbetűvel. Azok a nevek, ahol nem a konkrét nevet szerepeltetjük a leírásban, hanem csak a névre utalunk, ott kis dőlt betűkkel írtuk a névre utaló leírást.

Példák:

SELECT utasítás leírás:

```
SELECT * FROM táblanév;
```

Konkrét táblára kiadott SELECT utasítás:

```
SELECT * FROM ALKALMAZOTT;
```

II. PL/SQL blokkok

A PL/SQL nem egy önálló Oracle termék, hanem egy technológia. A PL/SQL motor végrehajtja a procedurális utasításokat, az SQL utasításokat átadja az Oracle szervernek. Az Oracle szerver is tárol PL/SQL kódot. PL/SQL motor az Oracle szervernek is része, de egyes alkalmazások (Forms, Reports) saját beépített motort használnak.

II. 1. Névtelen blokk

Névtelen blokk segítségével a fejlesztőkörnyezetből a program adatbázisban való letárolása nélkül lehet kódot futtatni. (SQL és PL/SQL utasítások egymásutánját.)

Változókat lehet létrehozni (DECLARE). Le lehet kezelni a blokkon belül keletkező hibákat az EXCEPTION ágban elhelyezett utasításokkal. A hibakezelés esemény vezérelt, a blokkon belül bárhol fellépő hiba esetén a program végrehajtása az EXCEPTION után folytatódik.

```
[DECLARE
    deklarációk]
BEGIN
    utasítások
[EXCEPTION
    kivételkezelés]
END ;
```

Lehetőség van névtelen blokkon (vagy tárolt eljáráson) belül újabb névtelen blokkok megadására. Főként a hibakezelés finomabb hangolása miatt indokolt blokkon belül újabb blokk létrehozása, de változó kezelés is indokolhatja, mivel a blokkban deklarált változók csak a blokkon belül ismertek.

Változók esetén egy adott változó abban a névtelen blokkban ismert, ahol létrehozták, és az összes olyan névtelen blokkban, amelyet ez a blokk tartalmaz. A hibakezelés esetén az adott blokk EXCEPTION ága kezeli le a hibát. Ha ebben nem kezeljük az adott hibát, akkor az a hívó blokk hibakezelőjében kezelhető, és így tovább, mindig a külső blokk felé továbbítódik a le nem kezelt hiba. Ha sehol sem kezeljük le a hibát, akkor az egész program megakad, a hibaüzenet megjelenik a képernyőn.

II. 2. Tárolt eljárás

Tárolt eljárás használatával dinamikusabbá tehetjük programjainkat, amely adatbázisban van tárolva, így újrafelhasználható kódot írhatunk.

```
CREATE [OR REPLACE] PROCEDURE eljárás neve  
    [(paraméterek)] IS  
    deklarációs rész  
BEGIN  
    utasítás rész  
    [EXCEPTION  
        kivételkezelés]  
END;
```

Hasonló a névtelen blokkhoz a szerkezete. Tartalmazhat névtelen blokkokat, hívhat másik eljárást, függvényt, és hívhatja önmagát (rekurzívhívás). A paraméterek ugyanolyan lokálisak, mint a deklarált változók. A paraméterek lehetnek input, output vagy mindkettő felhasználásúak. Az input (IN) típusú paraméter nem változtatható meg, kívülről kap értéket, ami konstans érték is lehet, vagy deklarált változó. Az output (OUT) paraméter nem kaphat értéket kívülről, csak a procedúrán belülről, amit a hívó eljárás visszakap. A vegyes (IN OUT) paraméter kívülről kap kezdőértéket, ami meg is változhat, a hívó program a megváltozott értéket visszakapja paraméter változójában. A hívó programban változónak kell állnia a formális paraméter helyén.

II. 3. Függvény

Az eljárásoktól annyiban különbözik, hogy megadhatunk visszatérési értéket.

```
CREATE [OR REPLACE] FUNCTION név [(paraméterek)]  
RETURN visszatérési típus IS  
    deklarációs rész  
BEGIN  
    utasítás rész  
    [EXCEPTION  
        kivételkezelés]  
END;
```

Függvények használhatók:

- SELECT utasítás oszloplistájában.
- WHERE és HAVING utasításrészben a logikai kifejezésrészeként.
- CONNECT BY, START WITH, ORDER BY, GROUP BY utasításrészben.
- INSERT utasítás VALUES részében.
- UPDATE utasításban a SET utasításrészben.
- Változó értékadásánál.

SQL kifejezésben csak akkor használhatók, ha:

- Tárolt függvény.
- Csak IN módú paramétereket tartalmaz.
- Csak érvényes SQL adattípusokat fogad paraméterként.
- Csak érvényes SQL adattípusokat ad vissza.
- Nem tartalmaz DML utasításokat.
- UPDATE/DELETE utasításban szereplő függvény nem olvassa a módosuló táblát.
- Nem tartalmaz tranzakció kezelő utasításokat.
- Nem hív meg olyan programokat, amelyek megsértik a fenti szabályokat.

II. 4. Paraméterek átadásának módjai

A paramétereket az eljárás, függvény létrehozásakor deklarálni kell:

paraméter_név [**IN** | **OUT** | **IN OUT**] [**NOCOPY**] *típus*
[**DEFAULT** *érték* | = *érték*]

IN paraméter:

- Értéket kap.
- Konstansként viselkedik.
- Alapértelmezett érték megadható.

OUT paraméter:

- Értéket ad vissza.
- Csak változó lehet.
- Kezdeti értékkel nem rendelkezik.

IN OUT paraméter:

- Értéket kap, és ad vissza.
- Csak változó lehet.
- Kezdeti értékkel rendelkezik.

OUT és IN OUT paraméter esetén lehet alkalmazni a NOCOPY opciót. Ez utasítja a PL/SQL fordítót, hogy a paramétert ne érték átadással, hanem cím átadással kezelje. Ha nem adjuk meg, akkor érték szerinti átadás történik. A cím szerinti átadás csökkenti a memória használatot, és a feldolgozás is gyorsabb. IN típusú paraméter esetén mindig cím szerinti átadás történik, vagyis a hívó program változójának a címét kapja meg a hívott program.

DEFAULT értéket megadva a hívó eljárásnak nem kell értéket átadni az adott paraméter helyén. Nagyon hasznos, ha utólag bővítünk egy eljárást, amit sok helyen használunk, de most egy olyan újabb paraméter kell, ami a többi helyen például NULL értékkel lenne hívva. Ekkor a DEFAULT NULL kezdőérték megadással elkerüljük, hogy a többi hívásnál meg kelljen változtatni a hívás formáját. Ha a hívás helyén nem szerepel a paraméterlistában ez a paraméter, akkor is fog kezdőértéket kapni, jelen esetben NULL-t.

Példa:

```
CREATE OR REPLACE PROCEDURE alkalmazott_leker(  
    p_nev OUT alkalmazott.nev%TYPE,  
    p_fizetes OUT alkalmazott.fizetes%TYPE,  
    p_azon IN alkalmazott.alkalmazott_azon%TYPE  
                                DEFAULT NULL)  
  
IS BEGIN  
    IF p_azon IS NULL THEN  
  
        SELECT nev, fizetes INTO p_nev, p_fizetes FROM  
            (SELECT nev, fizetes  
                FROM alkalmazott a  
                WHERE a.fizetes = (SELECT  
                                MAX(fizetes)  
                                FROM  
                                    alkalmazott)  
                ORDER BY a.nev)  
                WHERE ROWNUM = 1;  
  
    ELSE  
        SELECT nev, fizetes INTO p_nev, p_fizetes  
            FROM alkalmazott WHERE alkalmazott_azon = p_azon;  
    END IF;  
END alkalmazott_leker;
```

Három féle módon is hívhatjuk ezt az eljárást:

```
alkalmazott_leker(v_nev, v_fizetes, 10);  
alkalmazott_leker(v_nev, v_fizetes, NULL);  
alkalmazott_leker(v_nev, v_fizetes);
```

Az első esetben egy konkrét alkalmazott nevét és fizetését kapjuk vissza egy-egy változóban, a második és harmadik esetben a legmagasabb keresetű alkalmazottét. A DEFAULT értékadással válik lehetővé, hogy ahol a 3. formát használtuk eddig, ott nem kell módosítani a hívás formáját a 2. hívási módra.

III. Konstansok, változók

III. 1. PL/SQL változók deklarálása

Azonosító [**CONSTANT**] *adattípus* [**NOT NULL**] [**:=** | **DE FAULT** *kifejezés*];

NOT NULL használata esetén a kezdeti értékadás kötelező.

Kezdeti értéket a **DEFAULT** érték vagy a **:=** érték kifejezéssel lehet adni.

Azonosító:

- Legfeljebb 30 karakter hosszú lehet
- Betűvel kell kezdődnie
- Számokat, \$, # és _ jelet tartalmazhat
- -, / és szóköz jelet nem tartalmazhat
- Neve nem egyezhet meg adatbázistábla vagy oszlop nevével
- Nem lehet fenntartott szó

III. 2. Alaptípusok

Konstansok

SZÖVEG	'szöveg'	CHR(123) CHR(13)
SZÁM	123	321.123
DÁTUM	TO_DATE('2012-02-23 12:13:14', 'YYYY-MM-DD HH24:MI:SS')	
LOGIKAI	TRUE	FALSE

Változók

SZÖVEG	CHAR	Rögzített hosszúságú
	VARCHAR2	Változó hosszúságú
SZÁM	NUMBER	Pontosság megadható
	LONG	Változó hosszúságú
DÁTUM	DATE	Másodperc pontosságú
	TIMESTAMP	Pontosság megadható
LOGIKAI	BOOLEAN	True, False, NULL (SQL-ben nem használható)

Példa: `v_kezdes_datum DATE;`

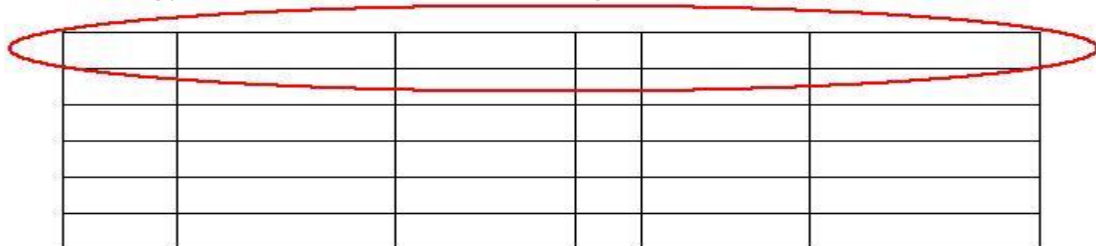
%TYPE

Ha egy adott mezőre vagy egy másik változó attribútumára szeretnénk hivatkozni, azt a %TYPE-pal tehetjük meg. Segítségével dinamikusabb kódot készíthetünk. Például ha egy tábla oszlopának a típusa megváltozik, nem kell programot változtatni hozzá. Az alábbi példában a BIZONYLAT tábla BIZT_BARKOD mezőjének típusával egyező típusú és hosszúságú változót hozunk létre.

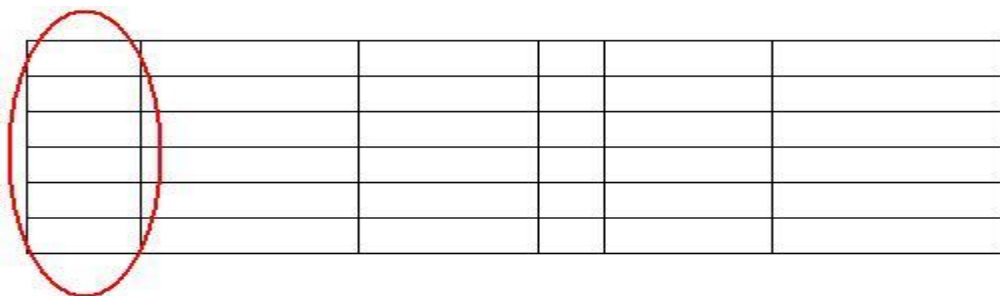
```
v_barkod BIZONYLAT.BIZT_BARKOD%TYPE;
```

III. 3. Összetett típusok

Rekordok: egy tábla sorainak felelnek meg.



Kollekciók: egy tábla oszlopainak felelnek meg.



Rekord

Egy rekord tartalmazhat:

- alaptípust
- rekordot
- kollekciót

Létrehozása: TYPE rekordnév IS RECORD (meződeklarációk);

Használata:

Változó deklarálása:

Változó rekordnév;

Rekord mezőjére történő hivatkozás:

Változó.mező

Példa:

```

TYPE YR_BIZONYLATRESZ IS RECORD (
    BRSZ_SORSZAM
        BIZONYLATRESZ.BRSZ_SORSZAM%TYPE,
    BRSZ_NYERS_AZONOSITO
        BIZONYLATRESZ.BRSZ_NYERS_AZONOSITO%TYPE,
    BRTS_AZON
        BIZONYLATRESZTIPUS.BRTS_AZON%TYPE) ;

```

```

v_brsz YR_BIZONYLATRESZ;
...
v_brsz.BRSZ_SORSZAM:=1;

```

%ROW_TYPE

A %TYPE-hoz hasonlóan egy tábla szerkezete alapján létrehozható rekordtípus.
A %TYPE-nál leírt előnyök itt is igazak!

Használata:

Változó deklarálása:	Változónév táblanév%ROWTYPE;
Rekord mezőjére történő hivatkozás:	Változónév.oszlopnév

Példa:

```

v_bizt BIZONYLAT%ROWTYPE;
...
v_bizt.BIZT_BARKOD:='1234567890';

```

Kollekció

- TÖMBÖK (array)
- INDEX BY táblák

Tartalmaz:

- indexet (BINARY_INTEGER, VARCHAR2,...)
- adatrészt (tetszőleges alap vagy rekordtípus)

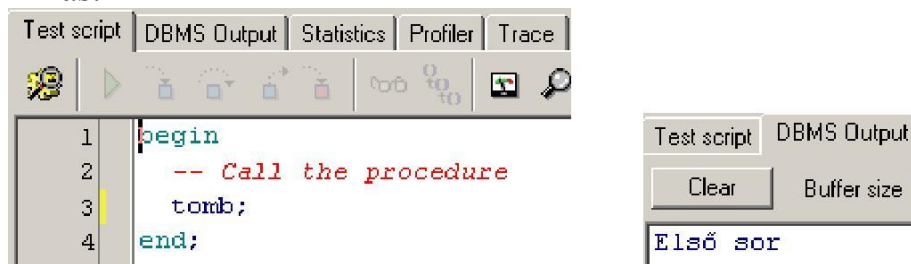
Tömb:

Létrehozása:	TYPE típusnév IS TABLE OF típus;
Használata:	Változó típusnév;

Példa:

```
--Egyszerű sztring tömb létrehozása:
CREATE OR REPLACE PROCEDURE tomb IS
TYPE params_array IS TABLE OF VARCHAR2(2048);
v_arr params_array;
BEGIN
v_arr := params_array();
v_arr.extend(1);
v_arr(1) := 'Első sor';
v_arr.extend(1);
v_arr(2) := 'Második sor';
    dbms_output.put_line(v_arr(1));
END tomb;
/
```

Kipróbálás PLSQL Developer Test ablakban, DBMS Output ablakban látszik a kiírás.



INDEX BY tábla:

Létrehozása: **TYPE típusnév IS TABLE OF típus INDEX BY index típus;**

Használata: *Változó típusnév;*

Példa:

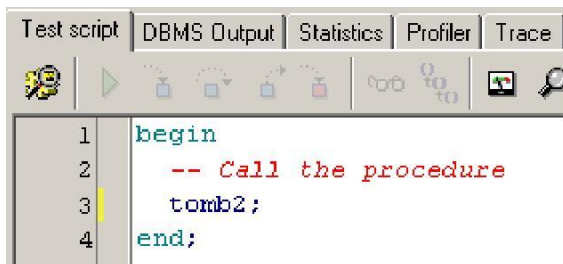
```
TYPE gyt_cache IS
    TABLE OF bizonylatmezo.bimo_azon%TYPE
    INDEX BY VARCHAR2(50);
v_bimo_azon    gyt_cache;
...
v_bimo_azon('teszt'):=1;
```

Index by táblák használatát segítő eljárások:

- EXISTS(x) létezik-e a x elem
- COUNT a táblában tárolt rekordok száma
- FIRST első elem
- LAST utolsó elem
- PRIOR(x) x előtti rekord indexe
- NEXT(x) x utáni rekord indexe
- TRIM utolsó elem törlése, (darab) esetén az utolsó darab elem törlése
- DELETE összes elem törlése, (x) x index ű elem törlése, (x,y) x és y közé eső elemek törlése

Példa:

```
--Egyszerű sztring tömb létrehozása:
CREATE OR REPLACE PROCEDURE tomb2 IS
    TYPE p_array IS TABLE OF VARCHAR2(2048)
        INDEX BY VARCHAR2(4);
    v_arr p_array;
BEGIN
    v_arr(1) := 'Első sor';
    v_arr(2) := 'Második sor';
    dbms_output.put_line(v_arr(2));
END tomb2;
```



IV. Értékadás változóknak

IV. 1. Értékadás

- Közvetlen értékadás := (pl. v_debug := 'teszt';)
- SQL utasítás (DQL, DML) eredményével
 - **INTO**
 - **BULK COLLECT INTO**

SELECT utasítás PL/SQL-ben is használható, de az eredményét egy változóba (INTO) vagy tömbbe (BULK COLLECT INTO) kell irányítani.

```
SELECT oszlopok
      INTO változók
FROM ...
```

INTO-t csak akkor lehet használni, ha a SELECT utasítás eredménye pontosan egy rekord. Ellenkező esetben NO_DATA_FOUND (nincs a feltételnek megfelelő rekord) vagy TOO_MANY_ROWS (több rekord is megfelel a feltételnek) kivétel keletkezik.

A változók típusának meg kell egyeznie a felsorolt oszlopok típusával.

Ha több rekord is előfordulhat a SELECT eredményeként, akkor használható a BULK COLLECT INTO értékadás.

```
SELECT oszlopok
      BULK COLLECT INTO tömb
FROM ...
```

BULK COLLECT INTO estén nem kell tartani a sima INTO-nál előforduló NO_DATA_FOUND kivételtől. Ilyen esetben üres tömböt kapunk vissza. TOO_MANY_ROWS pedig fel se merülhet, mert az összes rekord bekerül a tömbbe.

A tömb rekordszerkezetének meg kell egyeznie a felsorolt oszlopok típusával.

Lehetőség van a SELECT-tel egy kurzor nyitása (a tanfolyam későbbi részének témája) után soronként feltölteni a tömböt, de sokkal lassabb, mint a most taglalt változat! A későbbiekben látunk egy példát tömb feltöltésre.

IV. 2. A változók érvényességi köre

A változók abban a blokkban, ahol deklarálásra kerültek, illetve azon belül létrehozott újabb blokkokban érvényesek. Azonos nevű változó esetén a használat helyéhez képest legszűkebb blokkban deklarált lesz érvényes. (Minősítéssel a többi is elérhető) Megfordítva, blokkon belüli blokkban felvett változó a külső blokkban nem használható.

Névtelen blokkoknál a minősítés a blokk előtt elhelyezett címkével oldható meg. <<címke>>

Példa:

```
1      <<kulso>>
2      DECLARE
3          x NUMBER;
4          y NUMBER;
5      BEGIN
6          y := 13;
7      DECLARE
8          x NUMBER;
9      BEGIN
10         x := 1;
11         y := 77;
12         kulso.x :=
13         2;
14     END;
15 END;
```

A fenti programrészletben két x és egy y változó van deklarálva. A 3., 4. sorban egy x és egy y változó jön létre NULL értékkel. A 6. sorban az y 13-at kap értékül. A 8. sorban létrejön egy újabb x NULL értékkel, de ez csak a belső blokkban él. A 10. sorban a belső x 1 értéket kap, a külső marad NULL. A 11. sorban y 77 értéket kap, ez mindkét blokkban lekérdezhető, és 77-et fog visszaadni. A 12. sorban a külső blokkban szereplő x kap 2 értéket, a belső 1 marad. A 14. sorban lekérdezhetjük x értékét, ekkor 2-t kapnánk, mivel itt már csak egy x létezik, a külső blokké.

3. Operátorok

Precedencia:

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

PL/SQL-ben nem használhatók végrehajtható utasításokban a csoportfüggvények és a DECODE.

V. Vezérlési szerkezetek

V. 1. Feltételes elágazás

```
IF (feltétel) THEN
    utasítások
[ELSE
    utasítások]
END IF;

CASE [szelektor]
WHEN kifejezés1 THEN utasítások
WHEN kifejezés2 THEN utasítások
...
[ELSE utasítások]
END CASE;
```

Az első teljesülő kifejezés után álló utasítást hajtja végre.

Példa:

```
CASE x
  WHEN 1 THEN a := 'A';
  WHEN 2 THEN a := 'B'; b := 'A';
  ELSE a := 'X'
END CASE;

CASE
  WHEN x = 1 THEN a := 'A';
  WHEN x = 2 THEN a := 'B';
  WHEN z = 0 OR x = 3 THEN a := 'C';
  ELSE a := 'X'
END CASE;
```

V. 2. Ciklus

```
LOOP
    utasítások;
END LOOP;
```

A programnak kell gondoskodni a kilépésről, ezért a kilépési pont megadásával határozhatjuk meg, mennyi fusson le a ciklusmagból.

Kilépési lehetőségek:

- **EXIT** [**WHEN** feltétel];
- **IF THEN ELSE** szerkezetben feltétel nélküli **EXIT**
- **RETURN** segítségével is ki lehet lépni a ciklusból (és az aktuális tárolt eljárásból is).

A kilépési feltételre fokozottan figyelni kell, mert hibás megadása vagy elmaradása végtelen ciklust okoz!

V. 3. Feltételes ciklus

```
WHILE feltétel LOOP
    utasítások
END LOOP;
```

A ciklusmag addig hajtódik végre amíg a feltétel igaz. Amennyiben az első kiértékelés is igaznak bizonyul, a mag egyszer sem fut le.

V. 4. Számláló ciklus

FOR számláló IN [REVERSE] alsó határ .. felsőhatár LOOP
utasítások;
END LOOP;

Alsó és felső határ között egyesével megy végig. Ha van REVERSE, akkor csökken ő, egyébként növekvő sorrendben.

V. 5. Beágyazott ciklusok

A ciklusok több mélységben egymásba ágyazhatók.

Címkéket használhatunk az egyes szintek, illetve blokkok és ciklusok megkülönböztetésére. Külső ciklusból a saját címkéjére hivatkozó EXIT utasítással léphetünk ki.

Az END LOOP után írva a címke nevét jelezhetjük, melyik beágyazott ciklus végét jelzi az END LOOP.

Példa beágyazott ciklusra:

```
<<kulso_ciklus>>
LOOP
    ...
    EXIT WHEN feltétel1;
    ...
    <<belso_ciklus>>
    LOOP
        ...
        EXIT belso_ciklus WHEN feltétel2;
        ...
        EXIT kulso_ciklus WHEN feltétel3;
        ...
        EXIT WHEN feltétel4;
        ...
        IF feltétel5 THEN
            RETURN;
        END IF;
        ...
    END LOOP belso_ciklus;
    ...
END LOOP kulso_ciklus;
```

A fenti példában a feltétel1 és feltétel3 esetén a külső ciklust hagyjuk el, és a futás az „END LOOP kulso_ciklus;” utasítás után folytatódik. A feltétel2 és a feltétel4 a belső ciklusból léptet ki, a futás az „END LOOP belso_ciklus;” utasítás után folytatódik. A feltétel5 teljesülése esetén az egész programegységből kilépünk, vagyis egyik LOOP után sem fut le egy utasítás sem, hanem a program hívásának helyét követő sorban folytatódik a futás a hívó eljárásban.

EXIT-tel a FOR és a WHILE ciklusból ugyanígy ki lehet lépni, de az ritkábban használt, mivel a ciklusfej is tartalmaz feltétel kiértékelés.

VI. Kurzorok

Kurzorok segítségével egy SELECT utasítás eredményén lehet soronként végigmenni. Kétféle kurzor létezik:

- IMPLICIT (Minden DML és PL/SQL SELECT utasításhoz a szerver generálja.)
- EXPLICIT (A programozó definiálja, névvel látja el.)

VI. 1. Implicit kurzor

A számlálós ciklushoz hasonló kurzor. Előnye az egyszerűség:

```
FOR kurzornév IN (SELECT ...) LOOP
...
END LOOP;
```

VI. 2. Explicit kurzor

Deklarálás: **CURSOR** *kurzor név* **IS**
SELECT ... ;

Megnyitás: **OPEN** *kurzor név*;

Használat: **FETCH** *kurzor név* **INTO** *változók | rekord* | **BULK COLLECT**
INTO tömb;

Bezárás: **CLOSE** *kurzor név*;

Lekérdezhető tulajdonságok:

- | | |
|--------------------|--|
| - %ISOPEN | Nyitva van-e a kurzor |
| - %NOTFOUND | Az utolsó FETCH adott-e vissza értéket |
| - %FOUND | %NOTFOUND ellentétje |
| - %ROWCOUNT | eddig kiolvasott rekordok száma |

A kurzorokat dinamikusabbá tehetjük, ha nem fixen van megadva a lekérdezés

részük, hanem paramétereket viszünk be:

```
CURSOR kurzornév  
[(paraméterek)] IS  
    SELECT ...;  
....  
OPEN kurzornév(paraméterek);
```

Példa:

DECLARE

```
CURSOR v_sor (p_btip_kod bizonylattipus.btip_kod%TYPE)  
    IS SELECT bimo_sorszam FROM mv_btip_parameter  
    WHERE btip_kod= p_btip_kod;
```

```
vn_sorszam mv_btip_parameter.bimo_sorszam%TYPE;
```

BEGIN

```
    OPEN v_sor('1053');  
    IF v_sor%ISOPEN  
        THEN LOOP  
            FETCH v_sor INTO vn_sorszam;  
            EXIT WHEN v_sor%NOTFOUND;  
            ...  
        END LOOP;  
        CLOSE v_sor;  
    END IF;  
END;
```

FOR használata explicit kurzornál (gyorsabb)

```
FOR rekordnév IN kurzornév LOOP  
...  
END LOOP;
```

FOR használata allekérdezéssel

```
FOR rekordnév IN (SELECT ....) LOOP  
...  
END LOOP;
```

FOR UPDATE utasításrész

Explicit zárolással megoldjuk, hogy más tranzakció ne módosíthassa a kurzor által érintett sorokat.

```
SELECT ... FROM ...  
FOR UPDATE [OF oszlophivatkozás] [NOWAIT];
```

WHERE CURRENT OF utasításrész

```
UPDATE táblanév  
SET ...  
WHERE CURRENT OF kurzornév;
```

VI. 3. Kurzorváltozók

A kurzorváltozó is az aktuális sorra mutat, mint a kurzor. Úgy viszonyul a kurzorhoz, mint a konstans a változóhoz. Egy kurzorváltozó több különböző lekérdezés eredményére is mutathat a futás során.

Először a típust kell definiálni:

```
TYPE típus_név IS REF CURSOR [RETURN eredmény_típus];
```

Az *eredmény_típus* egy felhasználó által definiált rekordtípus vagy tábla sortípus.

A kurzorváltozó deklarálása:

```
Kurzor_változó_név típus_név;
```

A kurzorváltozó megnyitása:

```
OPEN kurzor_változó_név FOR select_utasítás;
```

Adatsorok behozatala:

```
FETCH kurzor_változó_név INTO változó_név1 [,változó_név2 ...] |  
rekord_név;
```

Kurzor lezárása:

```
CLOSE kurzor_változó_név;
```

A kurzorváltozó átadható másik programnak paraméterként, vagy felhasználható közvetlenül. Lezárt kurzorra való hivatkozás kiváltja az INVALID_CURSOR előre definiált kivételes helyzetet.

VI. 4. Tömb feltöltése kurzor segítségével

Típust séma objektumként is létre lehet hozni, ha több helyen használt általános szerkezet.

Rekordszerkezettel rendelkező tömb típus létrehozása:

```
CREATE OR REPLACE TYPE UBV_TORTENET_REC AS
  OBJECT (nev varchar2(50), sor varchar2(500)
  );
CREATE OR REPLACE TYPE UBV_TORTENET_ARRAY IS
  TABLE OF UBV_TORTENET_REC;
```

Egyszerű sztring tömb létrehozása:

```
CREATE OR REPLACE TYPE UBEV.JABEV_PARAMS_ARRAY IS
  TABLE OF VARCHAR2(2048);
```

A tömb használata a programban:

```
PROCEDURE bizonylat_reszek (
  p_ret      OUT VARCHAR2,
  p_bizt_azon IN  NUMBER,
  p_reszek_array OUT jabev_params_array
) IS
/*****
* Leírás : A bizonylathoz tartozó részleteket
*         adja vissza
*
* Paraméterek:
* -----
* - p_barkod IN bizonylat_barkód
* - p_ret OUT visszatérési érték
* - p_reszek_array OUT bizonylat adatai tömb-ben
*****/
v_count NUMBER := 0;
v_nev VARCHAR2(200);
v_cim VARCHAR2(200);
BEGIN
  p_ret := '0';
  p_reszek_array := jabev_params_array();

  SELECT COUNT(*)
    INTO v_count
  FROM bizonylatresz resz, bizonylatresztipus tip
  WHERE resz.bizt_azon = p_bizt_azon
    AND tip.brts_azon = resz.brts_azon;

  IF v_count = 0 THEN
    RETURN;
  END IF;

  p_reszek_array.EXTEND(1);
  p_reszek_array(1) := 'Ssz. Kód Adóazon Név';
  v_count := 2;
```



```

FOR reszlista IN
  (SELECT  resz.brsz sorszam,
           tip.brts rovid nev,
           resz.brsz nyers azonosito
    FROM    bizonylatresz resz,
           bizonylatresztipus tip
   WHERE    resz.bizt azon = p bizt azon
           AND tip.brts_azon = resz.brts_azon
   ORDER BY 1)
LOOP
  bef_partner.get_prn_nev_cim(
    -bef_partner.prn_azon by apeh kulcs(
    -reszlista.brsz_nyers_azonosito),
    v_nev,
    v_cim);

  p_reszek_array.EXTEND(1);
  p_reszek_array(v_count) :=
    LPAD(reszlista.brsz_sorszam, 6, ' ')
    || RPAD(reszlista.brts_rovid_nev, 20, ' ')
    || RPAD(reszlista.brsz_nyers_azonosito,
            11, ' ')
    || RPAD(v_nev, 50, ' ');

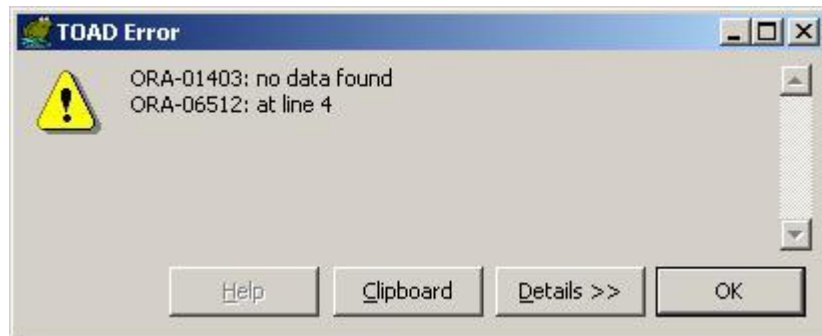
  v_count := v_count + 1;
END LOOP;
EXCEPTION
  WHEN OTHERS THEN
    p_ret := SUBSTR (SQLERRM, 1, 255);
END;

```

VII. Kivételek

A program futása közben előálló, a normális végrehajtástól eltérő állapotot kivételnek nevezzük. Kétféle kivétel létezik:

- Rendszer rendellenes futása következtében keletkező
- A programozó által definiált



VII. 1. Kivételek elfogása

Általában szükséges a kivételek programból való kezelése. Erre a blokk EXCEPTION részében van lehetőség:

DECLARE

...

BEGIN

...

EXCEPTION

```

    WHEN kivétel1 [OR kivétel2] THEN utasítások
    [WHEN kivétel3 THEN utasítások]
    [WHEN OTHERS THEN utasítások]

```

END ;

A blokkban le nem kezelt kivételek a hívóblokkban kezelhetők. Az egyáltalán nem kezelt kivételek megakasztják a program futását.

VII. 2. Gyakori kivételek

- | | | |
|--------------------------|-----------|--|
| - NO_DATA_FOUND | ORA-01403 | a SELECT nem hozott eredményt |
| - TOO_MANY_ROWS | ORA-01422 | a SELECT 1-nél több rekordot talált |
| - DUP_VAL_ON_INDEX | ORA-00001 | Elsődleges vagy egyedi kulcs kényszer megsértése |
| - SUBSCRIPT_BEYOND_COUNT | ORA-06533 | tömb méreténél nagyobb index használata |
| - ZERO_DIVIDE | ORA-01476 | nullával való osztás |

.....

Két rendszerfüggvény segítséget nyújt a kivételkezelésében:

- **SQLCODE**: kivétel kódja
- **SQLERRM**: kivétel hibaszövege

```
DECLARE
    vv_bizt_barkod bizonylat.bizt_barkod%TYPE;
BEGIN
    SELECT bizt_barkod INTO vv_bizt_barkod
    FROM bizonylat
    WHERE bizt_azon = 0;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.put_line('SQLCODE: ' || SQLCODE) ;
        DBMS_OUTPUT.put_line('SQLERRM: ' || SQLERRM) ;
END;
```

SQLCODE:100

SQLERRM:ORA-01403: no data found

Szükség esetén a RAISE utasítással lehet kivételes helyzetet előidézni:

RAISE kivétel;

Ha hibakezelő ágban tovább szeretnénk adni a hibát a hívó programnak is, akkor a hibakezelés végén is ki kell adni:

RAISE;

```
DECLARE
    vv_bizt_barkod bizonylat.bizt_barkod%TYPE;
BEGIN
    RAISE TOO_MANY_ROWS;
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.put_line('SQLCODE: ' || SQLCODE) ;
        DBMS_OUTPUT.put_line('SQLERRM: ' || SQLERRM) ;
END;
```

SQLCODE:-1422

SQLERRM:ORA-01422: exact fetch returns more than requested number of rows

Az SQLERRM helyett újabban a **DBMS_UTILITY.FORMAT_ERROR_STACK** függvényt használjuk. Az SQLERRM 512 byte hibaüzenetet ad vissza, ez utóbbi függvény 2000 karaktert. Ennek használatáról a „XIV. 5. Hiba kiírás a DBMS_UTILITY csomaggal” fejezetben lesz szó.

VII. 3. Saját kivételek

Saját kivételeket is definiálhatunk:

```
DECLARE kivétel_név EXCEPTION;  
BEGIN  
...  
    RAISE kivétel_név;  
...  
EXCEPTION  
    WHEN kivétel_név THEN  
...  
END;
```

Példa:

```
DECLARE e_teszt EXCEPTION;  
BEGIN  
    RAISE e_teszt;  
EXCEPTION  
    WHEN e_teszt THEN  
        DBMS_OUTPUT.put_line('SQLCODE: ' || SQLCODE) ;  
        DBMS_OUTPUT.put_line('SQLERRM: ' || SQLERRM) ;  
END;
```

SQLCODE:1
SQLERRM:User-Defined Exception

VII. 4. RAISE_APPLICATION_ERROR eljárás

Programozó által definiált hibaüzeneteket lehet kiadni programokból.

RAISE_APPLICATION_ERROR(*hibakód*, *üzenet* [, {TRUE|FALSE}]);

Hibakód: -20000 és -20999 között

lehet. Üzenet: Max. 2048 byte

hosszú hibaüzenet.

Opcionális paraméter: Ha TRUE, akkor a korábbi hibákat tartalmazó hibaverembe helyezi, ha FALSE, akkor lecseréli a korábbi hibákat erre (ez alapértelmezés).

Kiadható a PL/SQL eljárás végrehajtható részében és a kivételkezelő részben is.

VIII. Csomagok, alprogramok

A logikailag összetartozó tárolt eljárások, függvények, típusok, változók, konstansok csomagokba szervezhetők. Így könnyebben átlátható a rendszer.

A csomag két részből áll:

- **Specifikáció:** ebben találhatóak azok az objektumok, melyek kívülről is meghívhatóak.
- **Törzs:** a specifikációban deklarált tárolt eljárások definíciója (body) .

VIII. 1. Csomag létrehozása

```
CREATE [OR REPLACE] PACKAGE csomag_név IS
```

```
...
```

```
END [csomag_név];
```

```
CREATE [OR REPLACE] PACKAGE BODY csomag_név IS
```

```
...
```

```
END [csomag_név];
```

A csomag spec részében található objektumok a csomag és objektum nevével (ponttal elválasztva) elérhetők. Amik csak a body-ban találhatóak, azok a csomagon belül elérhetők, kívülről nem.

VIII. 2. Csomagváltozók

A csomag deklarációs részében (akár spec, akár body) megadott változók (csomagváltozók) a csomag összes tárolt eljárásában láthatóak. Egy session (kapcsolat) alatt a változó megtartja az értékét.

Ha a csomagváltozó a spec-ben szerepel, akkor kívülről is elérhető, ha a body-ban, akkor csak a csomagon belülről.

Példa:

```
CREATE OR REPLACE PACKAGE BEVFELD.BEF_COMMON_CHECK IS
/* PL/SQL csomag verziószáma */
FUNCTION GET_VERZIO RETURN VARCHAR2;
```

```
...
```

```
END BEF_COMMON_CHECK;
```

```
CREATE OR REPLACE PACKAGE BODY
```

```
BEVFELD.BEF_COMMON_CHECK IS
GCV_VERZIO CONSTANT VARCHAR2(30) := 'v1.10.0';
```

```
/* PL/SQL csomag verziószáma */
```

```
FUNCTION GET_VERZIO RETURN VARCHAR2 IS
```

```
BEGIN
    RETURN GCV_VERZIO;
END;
...
END BEF_COMMON_CHECK;
```

```
BEGIN
    DBMS_OUTPUT.put_line(bef_common_check.get_verzio);
END;
```

v1.10.0

VIII. 3. Törzs nélküli csomag

Amennyiben több csomag által is használt változókat, típusokat, rekordokat szeretnénk használni, azt érdemes kiemelni egy új csomagba. Ebben az esetben (nincs tárolt eljárás) szükségtelen az üres csomagtest létrehozása.

Ezt leginkább akkor érdemes használni, ha egyébként a csomagok egymásra hivatkoznának. A fordítás nehézségekre ütközhet a függőségek miatt.

Példa:

```
CREATE OR REPLACE PACKAGE csak_spec IS
    TYPE yr_bizonylatresz IS RECORD (
        brsz_sorszam bizonylatresz.brsz_sorszam%TYPE,
        brsz_nyers_azonosito
            bizonylatresz.brsz_nyers_azonosito%TYPE,
        brts_azon bizonylatresztipus.brts_azon%TYPE
    );
    TYPE yt_bizonylatresz IS TABLE OF yr_bizonylatresz;
END csak_spec;
```

Csomagon belül az YT_BIZONYLATRESZ típus felhasználható memória tömb deklarálásához. Más program CSAK_SPEC.YT_BIZONYLATRESZ –ként tud hivatkozni rá. Hasonlóan az eljárások esetén is, kívülről történő hivatkozás esetén a csomag nevét is meg kell adni egy ponttal elválasztva a hivatkozott csomag objektumtól.

VIII. 4. Előre deklaráció

Csomagon belül egymást hívhatják az eljárások, de hívása helyén már ismertnek kell lennie a hívott eljárásnak. Ez előre deklarációval biztosítható a lokális eljárások esetén a csomag törzsben.

Példa:

```
CREATE OR REPLACE PACKAGE BODY pelda IS
  PROCEDURE a (...);          -- Ez az előre deklaráció

  PROCEDURE b (...) IS
  BEGIN
    ...
    a (...);                  -- Itt már ismert az „a” eljárás
    ...
  END b;

  PROCEDURE a (...) IS        -- Ez az „a” eljárás definíciója
  BEGIN
    ...
  END a;
END pelda;
```

VIII. 5. Egyszer végrehajtódó kódrészlet

A csomag betöltésekor hívódik meg egyszer. Csomagváltozók értékeit állíthatjuk be vele. (például a felhasználó neve, bizonyos induló vagy a futás során többször lekérdezhető adatok)

Példa:

```
CREATE OR REPLACE PACKAGE pelda IS
    gv_user VARCHAR2(10);
    gv_user_nev VARCHAR2(50);
...
END pelda;
```

```
CREATE OR REPLACE PACKAGE BODY pelda IS
```

...

```
BEGIN
```

```
    gv_user := get_user;
    SELECT nev
        INTO gv_user_nev
        FROM XPUSER
        WHERE azon = gv_user;
```

```
END pelda;
```

Az egyszer végrehajtandó blokk végén nincs külön END, mivel ez a blokk mindig a csomag legvégén áll.

VIII. 6. Többjelentésű programok

Csomagban ugyanazon név alatt hozhatók létre programváltozatok, amelyek különböző paramétereket kezelnek. A formális paramétereknek különbözniük kell számukban, sorrendjükben vagy típusukban.

Példa Oracle többjelentésű függvényekre:

```
FUNCTION TO_CHAR(p1 DATE) RETURN VARCHAR2;
FUNCTION TO_CHAR(p1 DATE, p2 VARCHAR2) RETURN
VARCHAR2;
FUNCTION TO_CHAR(p1 NUMBER) RETURN VARCHAR2;
FUNCTION TO_CHAR(p1 NUMBER, p2 VARCHAR2) RETURN
VARCHAR2;
```


IX. Programok és jogosultságok

IX. 1. Programok készítéséhez szükséges jogosultságok

A programok létrehozásához a rendszergazda adhat jogosultságokat, futtatásukhoz pedig a rendszergazda vagy a program tulajdonosa.

Kb. 80 rendszer privilégium van. SYSTEM és SYS nevű felhasználó rendelkezik velük (DBA). Saját objektumokhoz a tulajdonos is adhat mások számára jogosultságokat.

A kiadás módja: **GRANT** *privilégium* **TO** *felhasználó|szerepkör*;

DBA által adható privilégiumok:

CREATE (ANY) PROCEDURE

ALTER ANY PROCEDURE

DROP ANY PROCEDURE

EXECUTE ANY PROCEDURE

A PROCEDURE kulcsszó itt nemcsak eljárást, hanem függvényt és csomagot is jelent. A CREATE PROCEDURE privilégiummal létrehozhatunk eljárást, függvényt, csomagot. Minden további privilégium nélkül tudunk módosítani, törölni és futtatni is a CREATE privilégiummal.

Tulajdonos által adható privilégium: **EXECUTE**

GRANT EXECUTE ON *programnév* **TO** *felhasználónév|szerepkör*
[WITH GRANT OPTION];

A **WITH GRANT OPTION** záradék a jogosultság továbbadására is lehetőséget ad.

IX. 2. A programok hozzáférése adatokhoz

A programok a tulajdonos jogaival futnak. Ha a tulajdonos hozzáfér egy táblához, amit a program használ, akkor amennyiben a tulajdonos vagy a rendszergazda jogosultságot adott valakinek a program futtatásához, akkor a program is el fogja érni az adatokat akkor is, ha a futtató felhasználónak nincs joga az adott tábla eléréséhez.

Ha azt szeretnénk, hogy a program a futtató jogosultságaival rendelkezzen, akkor azt közölni kell a program végrehajtható része előtt az **AUTHID CURRENT_USER** kulcsszavakkal:

```
CREATE PROCEDURE pr( x IN NUMBER) AUTHID CURRENT_USER  
IS  
BEGIN  
....  
END;
```

X. Tranzakció kezelés

X. 1. Normál tranzakciók

Az **UPDATE**, **INSERT** utasítások mindaddig egy tranzakciót alkotnak, amíg nem érkezik egy explicit vagy implicit **COMMIT**. Ez idő alatt a változások csak a saját session-on belül kérdezhetők le, minden más session-ben a korábbi adatok láthatók. A módosításkor a táblába bekerülnek az új adatok, de egy másolat készül a korábbi adatokról. A más session-ok **SELECT**-jei a másolat rekordot kapják meg az adatbázis kezelőtől. Ha **COMMIT** érkezik, akkor a másolat törlődik, így minden **SELECT** már a módosult adatokat látja. Ha **ROLLBACK** utasítás érkezik, akkor visszamásolódnak az adatbázisba a korábbi adatok. A **ROLLBACK** visszaállítja a tranzakció kezdeténél lévő állapotot. Köztes állapotig is vissza lehet lépni **SAVEPOINT**-ok elhelyezésével.

Implicit **COMMIT** következik be, ha DML utasítás hajtódik végre.

Szintaktika:

```
COMMIT;  
SAVEPOINT név;  
ROLLBACK [TO SAVEPOINT név];
```

Példa:

```
INSERT ...;  
SAVEPOINT a;  
INSERT ...;  
SAVEPOINT b;  
INSERT ...;  
...  
ROLLBACK TO SAVEPOINT b;
```

X. 2. Autonóm tranzakciók

Eljárások, függvények deklarációs részében elhelyezhető a

PRAGMA AUTONOMOUS TRANSACTION;

utasítás. A programrészben kiadott COMMIT vagy ROLLBACK csak a programrészben kiadott DML utasításokra van hatással, a hívóprogram tranzakcióját nem érinti. Tipikus használata, amikor hibakezelésben meghívunk egy hiba naplózó eljárást. A naplózó eljárás COMMIT-tal letárolja a naplózandó adatokat, ugyanakkor a hibakezelő részbe visszatérve a fő tranzakció minden bizonnyal vissza lesz görgetve a ROLLBACK utasítással a hiba előállását megelőző konzisztens állapotba.

XI. Triggerek

Egy esemény hatására, automatikusan lefutó, előre definiált programokat nevezzük triggereknek. Ezek az események lehetnek DML műveletek, rendszeresemények, felhasználói események (login, logoff, select,...).

XI. 1. Alkalmazás triggerek

Futása szerint:

- **BEFORE**: DML utasítás végrehajtása előtt
- **AFTER**: DML utasítás végrehajtása után
- **INSTEAD OF**: DML utasítás végrehajtása helyett (nézetnél)

Kiváltó ok szerint:

- **INSERT**
- **UPDATE**
- **DELETE**

Futások száma szerint:

- Sor szintű (**FOR EACH ROW**): minden módosuló sorra lefut
- Utasítás szintű: felhasználói auditálás

Példák:

BEFORE: kényszerekkel túl bonyolultan védhető konzisztencia.

Beszúrás, módosítás dátumának feltöltése.

AFTER: naplózás

Szintaktika

```
CREATE [OR REPLACE] TRIGGER trigger_név  
{BEFORE|AFTER|INSTEAD OF}  
{esemény1 [OR esemény2 [OR esemény3]]} ON tábla  
[FOR EACH ROW [WHEN feltétel]]  
BEGIN  
...  
END;
```

Esemény lehet: **INSERT, DELETE, UPDATE** [**OF oszlop**]

Több eseményhez kapcsolódó triggerben az események elágaztathatók IF-ekkel a trigger törzsében a **DELETING, INSERTING** és **UPDATING** kulcsszavak segítségével.

Ki-be kapcsolás:

```
ALTER TRIGGER trigger DISABLE|ENABLE;
```

Törlés:

```
DROP TRIGGER trigger;
```

Tábla összes triggerének ki-be kapcsolása

```
ALTER TABLE tábla DISABLE|ENABLE ALL TRIGGERS;
```

Minősített nevek

A soronkénti trigger utasítás részében lehet hivatkozni a módosítás előtti és a módosítás folyamán változott értékre:

:OLD a módosítás előtti
érték **:NEW** a módosítás
utáni érték

Értelemszerűen a **DELETE**-nél csak az **:OLD**, **INSERT**-nél csak a **:NEW** használható

Jogosultság:

CREATE/ALTER/DROP TRIGGER jogosultság kell a saját sémában lévő trigger kezeléséhez, 'ANY' ha tetszőleges sémában. A trigger a tulajdonos jogosultságaival fut.

Példa trigger:

```
CREATE OR REPLACE TRIGGER ellenorzo
BEFORE UPDATE OR DELETE
ON en_tablam
BEGIN
    IF DELETING THEN
        RAISE_APPLICATION_ERROR(-20500, 'Az
        EN_TABLAM táblából tilos törölni !');
    ELSIF UPDATING ('AZON') THEN
        RAISE_APPLICATION_ERROR(-20501, 'Az
        azonosító mezőt tilos módosítani !');
    END IF;
END;
```

```
CREATE OR REPLACE TRIGGER kitolto
BEFORE INSERT OR UPDATE
ON en_tablam
FOR EACH ROW
BEGIN
    :NEW.MODIF_DATE := SYSDATE;
    IF INSERTING THEN
        :NEW.CREATE_DATE := SYSDATE;
    END IF;
END;
```

XI. 2. Adatbázis triggererek

Lefut, ha adott esemény (pl. DDL utasítás) vagy rendszeresemény (login, logoff, shutdown) bekövetkezik.

Trigger készítése DDL utasításra

```
CREATE [OR REPLACE] TRIGGER trigger_név
BEFORE | AFTER
    [ddl_esemény1 [OR ddl_esemény2 OR ...]]
ON DATABASE | SCHEMA
BEGIN
    trigger_törzs
END;
```

A lehetséges ddl_esemény értékek:

```
CREATE
ALTER
DROP
```

A trigger lefut bármely objektumra, amelyen bekövetkezik a fenti események valamelyike.

Példa DDL szintű triggerre:

```
CREATE OR REPLACE TRIGGER ddl_trigger
AFTER CREATE OR ALTER OR DROP ON DATABASE
WHEN (RTRIM(user) <> 'SYS'
AND RTRIM(user) <> 'SYSTEM')
BEGIN
    INSERT INTO naplo
        (user_name, event_date, sys_event,
         object_type, object_name)
    VALUES
        (ORA_LOGIN_USER, CURRENT_TIMESTAMP,
         ORA_SYSEVENT, ORA_DICT_OBJ_TYPE,
         ORA_DICT_OBJ_NAME);
END;
```

A trigger naplózza a SYS és SYSTEM userek kivételév az összes CREATE, ALTER és DROP utasítást.

Trigger készítése rendszereseményre:

```
CREATE [OR REPLACE] TRIGGER trigger_név
BEFORE | AFTER
    [adatbázis_esemény1 [OR adatbázis_esemény2 OR ...]]
ON DATABASE | SCHEMA
BEGIN
    trigger_törzs
END;
```

Az adatbázis_esemény a következő lehet:

```
AFTER SERVERERROR
AFTER LOGON
BEFORE LOGOFF
AFTER STARTUP
BEFORE SHUTDOWN
```

XII. PL/SQL programegységek újrafordítása

Az invalid objektumok bizonyos esetekben automatikusan, implicit, futás időben újrafordulnak. Explicit újrafordítást végezhetünk **ALTER** paranccsal.

```
ALTER PROCEDURE [séma.]név COMPILE;  
ALTER FUNCTION [séma.]név COMPILE;  
ALTER PACKAGE [séma.]név COMPILE [PACKAGE];  
ALTER PACKAGE [séma.]név COMPILE BODY;  
ALTER TRIGGER [séma.]név [COMPILE [DEBUG]];
```

Újrafordításkor először minden olyan objektumot lefordít, amelytől függ. Fordítási hiba esetén **INVALID** lesz az objektum.

Az **ALTER PACKAGE** [séma.]név **COMPILE** lefordítja a package specifikációt is és a body-t is.

XIII. Dinamikus SQL

Előre nem, vagy nagyon bonyolultan elkészíthető utasítás esetén van lehetőség string műveletekkel való operálásra, majd az elkészült utasítás futtatására. Az utasítást program állítja elő egy változóban, majd lefuttatja.

```
EXECUTE IMMEDIATE stringes_utasítás  
[INTO változók] --ha egy sor az eredmény  
[USING paraméterek]
```

Egyes utasítások (DDL) csak így futtathatók PL/SQL programból:

```
EXECUTE IMMEDIATE 'TRUNCATE TABLE my_tab';
```

XIV. Gyári csomagok

XIV. 1. Kiírás képernyőre

A standard kimenetre való írásban segít a **DBMS_OUTPUT** gyári csomag. Képernyős alkalmazások esetén tesztelésben segít:

Kimenet bekapcsolása:

PROCEDURE ENABLE(buffer_size in integer default 20000);

Írás a kimenetre:

PROCEDURE PUT(a varchar2);

PROCEDURE PUT_LINE(a varchar2);

Kimenet kikapcsolása: **PROCEDURE DISABLE**;

Példa:

```
BEGIN
  DBMS_OUTPUT.put_line(SYSDATE);
END;
```

XIV. 2. Fájl kezelés

Oracle rendszerben a fájlkezelést az **UTL_FILE** csomag segíti.

Könyvtár elérést **DIRECTORY** létrehozásával lehet adni.

CREATE DIRECTORY *név* **AS** *könyvtár*;

GRANT READ ON DIRECTORY *név* **TO** *felhasználó* | *szerepkör* | **PUBLIC**;

GRANT WRITE ON DIRECTORY *név* **TO** *felhasználó* | *szerepkör* | **PUBLIC**;

Lehetőségek:

Fájlnyitás: **FOPEN**

Olvasás: **GET_LINE**

Sorírás: **NEW_LINE**

Fájlzárás: **FCLOSE**

XIV. 3. JOB kezelés

Háttérben való időzített futtatásra szolgál a **DBMS_JOB** csomag:

Ütemezés: **SUBMIT**

Törlés: **REMOVE**

Indítás: **RUN**

A futó job két helyen is megtalálható (job queue, és a session). Törlés esetén mindkettőre figyelni kell, mert vagy nem áll le, vagy újra fog indulni.

Az ORACLE 10g verzió óta a **DMS_SCHEDULER** csomagot ajánlják, több hangolási lehetősége van.

A job ütemező használata

A **DBMS_SCHEDULER.CREATE_JOB** eljárással lehet létrehozni ütemezett job-okat. Egy példa tárolt eljárás hívására:

```
BEGIN
dbms_scheduler.create_job(
  job_name          => 'run_load_sales',
  job_type          => 'STORED_PROCEDURE',
  job_action        => 'system.load_sales',
  start_date        => '01-MAR-2010 03:00:00 AM',
  repeat_interval   => 'FREQ=DAILY',
  enabled           => TRUE);
END;
/
```

A fenti példa létrehoz egy run_load_sales nevű munkafolyamatot, amely egy system.load_sales nevű tárolt eljárást futtat. Az indítás időpontja 2010 március 1, hajnali 3 óra és naponta elindul. Általában a job-ok a létrehozásukkor nincsenek engedélyezve, de ebben a példában rögtön engedélyeztük is a futását.

Az ütemezett feladatok neve, úgy viselkedik, mint bármely más adatbázis objektum, a sémán belül egyedinek kell lennie.

Minden alkalommal, amikor fut a job, az ütemező kiértékeli az indítási feltételeket, és beütemezi a következő futást.

Sok kifejezést lehet használni a gyakoriság beállítására, néhány példa:

Ismétlési intervallum

freq=hourly

freq=daily; byhour=3

freq=daily; byhour=8,20

freq=monthly; bymonthday=1

freq=monthly; bymonthday=-1

freq=yearly; bymonth=sep; bymonthday=20;

Leírás

Minden órában fut

Minden nap 3 órakor fut

Minden nap reggel és este 8-kor fut

Minden hónap elsején fut

Minden hónap utolsó napján fut

Minden év szeptember 20-án

Operációs rendszerbeli programok futtatása

A **job type** paraméter **executable** értékének a megadásával tudunk olyan ütemezett feladatot létrehozni, amely egy operáció rendszer alatti programot vagy parancsfájlt futtat le. A szintaxisa megegyezik a többi esettel a különbség a **job_type** megadásában van, valamint a **job_action** paraméterben a futtatandó program teljes könyvtári elérését is meg kell adni:

```
BEGIN
dbms_scheduler.create_job (
    job_name           => 'migrate_files',
    job_type           => 'executable',
    job_action         => '/home/bin/files.sh',
    start_date         => '01-mar-2010 07:00:00 am',
    repeat_interval    => 'freq=daily',
    enabled            => true);
END;
/
```

A kijelölt időben az Oracle lefuttatja a files.sh Unix szkriptet annak a felhasználónak a nevében, aki alatt az adatbázisban elindult (tipikusan oracle). Ennek a felhasználónak futtatási jogának kell lennie a futtatandó parancsfájlon.

Job paraméterek változtatása

A **DBMS_SCHEDULER.SET_ATTRIBUTE** eljárással a job nevéen kívül bármelyik paraméterét meg lehet változtatni.. A job neve után meg kell adni a megváltoztatni kívánt tulajdonság nevét, majd az új értéket:

```
BEGIN
    dbms_scheduler.set_attribute (
        name           => 'run_load_sales',
        attribute       => 'repeat_interval',
        value          => 'freq=daily;
        byhour=3');
END;
/
```

A job futása közben is meg lehet változtatni a paramétereket, de csak a következő futáskor lépnek érvénybe.

Job kézi futtatása

Ha azonnali futtatás szükséges, akkor a **DBMS_SCHEDULER.RUN_JOB** nevű eljárást kell elindítani.

```
BEGIN
dbms_scheduler.run_job(job_name => 'run_load_sales');
END;
/
```

Futó job leállítása

Futó ütemezett feladatot a **DBMS_SCHEDULER.STOP_JOB** eljárással lehet leállítani.

```
BEGIN
dbms_scheduler.stop_job(job_name => 'run_load_sales');
END;
/
```

Csak az éppen futó job-ot állítja le, a jövőbeni futásokra nincs hatása.

Job letiltása, engedélyezése

A **DBMS_SCHEDULER** csomag tartalmaz egy **DISABLE** és egy **ENABLE** eljárást a feladatok tiltására vagy engedélyezésére. Ha tiltott egy job, akkor nem fog elindulni.

```
BEGIN
dbms_scheduler.disable(job_name => 'run_load_sales');
END;
/
```

Ha a job fut, akkor a **DISABLE** eljárás hibát jelez. A futó job-ot az előző eljárással lehet leállítani, vagy meg kell adni egy **force => true** paramétert is a **DISABLE** eljárásnak.

Az újra engedélyezés az **ENABLE** eljárással történik.

```
BEGIN
dbms_scheduler.enable(job_name => 'run_load_sales');
END;
/
```

Több feladat is leállítható vagy elindítható egyidejűleg, ekkor vesszővel elválasztva kell felsorolni a nevüket.

Job eldobása

Egy feladat végleges eldobásához a **DBMS_SCHEDULER.DROP_JOB** eljárást kell meghívni. Itt is vesszővel elválasztott listát is át lehet adni a csoportos eldobáshoz.

```
BEGIN
dbms_scheduler.drop_job('run_load_sales');
END;
/
```

Ha éppen fut a job, amit el akarunk dobni, hibaüzenetet kapunk. Ha mégis el akarjuk dobni, akkor először le kell állítani, vagy a **force** paramétert TRUE értékre állítani. Ekkor leállítja a futást, majd eldobja a job bejegyzést az ütemezőből.

XIV. 4. Nagy méretű objektumok

Nagy méretű (2 GByte-nál nagyobb) adatok kezelése.

- BLOB: bináris
- CLOB: karakteres
- CNLOB: fix méretű karakteres
- BFILE: bináris file

A BFILE az adatbázison kívül tárolódik, míg a többi belül, de nem a táblában. Ott csak egy mutató található.

Mielőtt használunk egy LOB mezőt, inicializálni kell. Ez létrehoz egy lokátort, ami egy üres LOB-ra mutat. Ez nem azonos a NULL értékkel. Az inicializálatlan LOB mező értéke NULL, de ez nem használható. A LOB inicializálása úgy történik, hogy INSERT vagy UPDATE során az EMPTY_LOB() rendszer függvényt adjuk meg a mező értékének. Ezzel a NULL helyett egy üres LOB mutatója kerül a mezőbe. Ebbe az üres LOB-ba már lehet adatokat tárolni.

Példa LOB inicializálásra, feltöltésre:

```
INSERT INTO lobtab (id, szoveg, kep)
      VALUES (123, EMPTY_CLOB(), NULL);
UPDATE lobtab
  SET szoveg = 'Jó hosszú szöveget lehet ide beírni.',
      kep = EMPTY_BLOB()
WHERE id = 123;
```

A LOB-ok használatát a DBMS_LOB csomag segíti. Lehetőségek:

```
APPEND
COPY
ERASE
TRIM
WRITE
COMPARE
INSTR
READ
SUBSTR
GETLENGTH
LOADFROMFILE
...
```

Példa:

```
/* BLOB tartalmának fájlba írása */
PROCEDURE SAVE_BLOB_TO_FILE(PV_DIR IN VARCHAR2
                           ,PV_FILE IN VARCHAR2
                           ,PBLOB IN BLOB )

IS
BEGIN
  /* A megadott BLOB tartalmát a megadott
könyvtárba (Oracle Directory), a megadott
névre menti. */
  DECLARE
    vfile utl_file.file_type;
  BEGIN vfile
    DECLARE
      vraw      RAW(32767);
      vn_amount BINARY_INTEGER DEFAULT
        DBMS_LOB.GETCHUNKSIZE(pblob);
      vn_offset INTEGER DEFAULT 1;
```

```
BEGIN
LOOP
    DBMS_LOB.READ(pblob, vn_amount, vn_offset,
                  vraw);
    UTL_FILE.PUT_RAW(vfile, vraw);
    vn_offset := vn_offset + vn_amount;
END LOOP;
EXCEPTION
    WHEN no_data_found THEN
        NULL; -- Vége a fájlnek, nincs mit tenni
    END;
    UTL_FILE.FCLOSE(vfile);
EXCEPTION
    WHEN OTHERS THEN
        IF UTL_FILE.IS_OPEN(vfile) THEN
            BEGIN
                UTL_FILE.FCLOSE(vfile);
            EXCEPTION
                WHEN OTHERS THEN NULL;
            END;
        END IF;
        RAISE;
    END;
EXCEPTION
    WHEN OTHERS THEN
        QMS$ERRORS.UNHANDLED_EXCEPTION('SAVE_BLOB_TO
        _FILE');
END;
```

XIV. 5. Hiba kiírás a DBMS_UTILITY csomaggal

A hibakezelés tárgyalásakor kipróbáltuk az SQLCODE és SQLERRM használatát (VII. Kivételek fejezet). Az SQLERRM 512 karakteren adja vissza a hibaüzenetet, emiatt csönkulhat a kiírt hibaüzenet.

A DBMS_UTILITY.FORMAT_ERROR_STACK használatával megtudhatjuk ugyanezt az 512-es korlát nélkül (ez 2000 karaktert tud visszaadni). A DBMS_UTILITY.FORMAT_ERROR_BACKTRACE mutatja meg, hol keletkezett hiba. A hibaüzenet ORA-06512, ami nem a valós hiba, viszont pontosan megmutatja a hívási láncot, és a hiba keletkezésének helyét. Ezek főként akkor hasznosak, ha több eljárás hívogatja egymást. Az alábbi mintaprogramban egy nullával osztást helyeztünk el, ennek a hibának a megjelenését íratjuk ki az EXCEPTION részben a különböző eszközökkel. Hogy a hívási láncot is szemléltessem, 3 eljárást hoztam létre, amelyek egymást hívják. A hiba a legbelsőben van elhelyezve, a hibakezelés viszont a hívó névtelen blokkban.

```
CREATE OR REPLACE PROCEDURE a IS
BEGIN
    b;
END;
/
```

```
CREATE OR REPLACE PROCEDURE b IS
BEGIN
    c;
END;
/
```

```
CREATE OR REPLACE PROCEDURE c IS
    x NUMBER;
BEGIN
    x := 1/0;
END;
/
```

A hívó névtelen blokk:

```
SET SERVEROUTPUT ON
BEGIN
    a;
EXCEPTION
    WHEN OTHERS
    THEN
        DBMS_OUTPUT.put_line('SQLCODE: ');
        DBMS_OUTPUT.put_line(SQLCODE);
        DBMS_OUTPUT.put_line('SQLERRM: ');
        DBMS_OUTPUT.put_line(SQLERRM);
        DBMS_OUTPUT.put_line(
            'DBMS_UTILITY.FORMAT_ERROR_STACK:');
        DBMS_OUTPUT.put_line(
            DBMS_UTILITY.FORMAT_ERROR_STACK);
        DBMS_OUTPUT.put_line(
            'DBMS_UTILITY.FORMAT_ERROR_BACKTRACE:');
        DBMS_OUTPUT.put_line(
            DBMS_UTILITY.FORMAT_ERROR_BACKTRACE);
END;
/
```

Az eredmény:

```
SQLCODE:
-1476
SQLERRM:
ORA-01476: divisor is equal to zero
```

```
DBMS_UTILITY.FORMAT_ERROR_STACK:
ORA-01476: divisor is equal to zero
```

```
DBMS_UTILITY.FORMAT_ERROR_BACKTRACE:
ORA-06512: at "TANF.C", line 4
ORA-06512: at "TANF.B", line 3
ORA-06512: at "TANF.A", line 3
ORA-06512: at line 2
```

Mint látható, a hibaüzenet kiolvasható az SQLERRM függvényből és a FORMAT_ERROR_STACK package függvényből, a hiba helye pedig a FORMAT_ERROR_BACKTRACE függvényből.

XV. ORACLE adatszótárbeli nézetek

XV. 1. USER_OBJECTS

A felhasználó saját objektumai a USER_OBJECTS nézetből kérdezhetők le. Az ALL_OBJECTS és a DBA_OBJECTS nézetekkel mások objektumai is lekérdezhetők, ha van jogunk ezekhez a nézetekhez.

A USER_OBJECTS nézet oszlopai:

OBJECT_NAME	Az objektum neve
OBJECT_ID	Az objektum belső azonosítója
OBJECT_TYPE	Az objektum típusa (pl. TABLE, PROCEDURE)
CREATED	Létrehozás ideje
LAST_DDL_TIME	Utolsó módosítás ideje
TIMESTAMP	Az utolsó újrafordítás ideje
STATUS	VALID vagy INVALID

XV. 2. USER_SOURCE

A programok forráskódja a USER_SOURCE nézetből kérdezhető le. (ALL_SOURCE, DBA_SOURCE)

Oszlopai:

NAME	Az objektum neve
TYPE	Típusa (PROCEDURE, FUNCTION, PACKAGE, ...)
LINE	A forrás sor sorszáma
TEXT	A forrás sor szövege

Eljárások és függvények paramétereirőla DESCRIBE parancs segítségével kaphatunk információt.

XV. 3. USER_ERRORS

Fordítási hibák megtekintése USER_ERRORS nézettel.

Oszlopai:

NAME	Az objektum neve
TYPE	Típusa (PROCEDURE, FUNCTION,
PACKAGE, ...)	
SEQUENCE	Szekvencia sorszáma
LINE	A forrásprogram hibás sorának sorszáma
POSITION	A hiba soron belüli pozíciója
TEXT	A hibaüzenet szövege

A USER_ERRORS minden korábbi fordítási hibát tárol.

Az utolsó fordítási hiba megtekintése:

SHOW ERRORS [FUNCTION | PROCEDURE | PACKAGE |
PACKAGE_BODY | TRIGGER | VIEW] [séma.]név]

A SHOW ERRORS parancs az utoljára keletkezett fordítási hibát írja ki.

XV. 4. USER_DEPENDENCIES

USER_DEPENDENCIES nézet tartalmazza az objektumok közötti függőségeket.

Oszlopai:

NAME	A függő objektum neve
TYPE	Típusa (PROCEDURE, FUNCTION)
REFERENCED_OWNER	A hivatkozott objektum sémája
REFERENCED_NAME	Neve
REFERENCED_TYPE	Típusa
REFERENCED_LINK_NAME	Adatbázis kapcsolat, amellyel elérhető.

XV. 5. USER_TRIGGERS

Információk a triggererekről a USER_TRIGGERS (ALL_TRIGGERS, DBA_TRIGGERS) nézetben találhatók.

Oszlopai:

TRIGGER_NAME	A trigger neve
TRIGGER_TYPE	Típusa (BEFORE, AFTER, INSTEAD OF)
TRIGGERING_EVENT	Az esemény megnevezése
TABLE_NAME	A tábla neve
REFERENCING_NAMES	Nevek az :OLD, :NEW rekordokra
WHEN_CLAUSE	Az alkalmazott WHEN feltétel
STATUS	A trigger státusza
TRIGGER_BODY	A trigger törzse

XV. 6. USER_TABLES

Körülbelül 50 oszlopot tartalmaz. Ezekben megtalálhatók a tárolásra vonatkozó információk, és statisztikai adatok is, mint például a sorok átlagos hossza, üres blokkok száma, stb.

XV. 7. USER_VIEWS

Tíz oszlopából az alábbi három fontos:

VIEW_NAME	A nézet neve
TEXT	A nézet által használt lekérdezés
TEXT_LENGTH	A lekérdezés hossza karakterben

XV. 8. USER_TAB_COLUMNS

Táblák oszlopainak adatai.

- Azonosító adatok: TABLE_NAME, COLUMN_NAME
- Definiáló adatok: DATA_TYPE, DATA_LENGTH, NULLABLE, ...
- Statisztikák: NUM_DISTINCT, NUM_NULLS, LOW_VALUE, HIGH_VALUE, ...

XV. 9. USER_TAB_PRIVS

Azokat a jogosultságokat láthatjuk, amelyeket a user adott vagy kapott, vagy az ő objektumaira vonatkozik.

Oszlopok: GRantee
GRANTOR
OWNER
TABLE_NAME
HIERARCHY
PRIVILEGE
GRANTABLE

XV. 10. USER_SYS_PRIVS

A rendszer jogosultságok kérdezhetők le.

Oszlopok: USERNAME
PRIVILEGE
ADMIN_OPTION

XV. 11. USER_ROLE_PRIVS

A felhasználóhoz tartozó szerepkörök jelennek meg. A PUBLIC felhasználóhoz rendelt szerepkörök is megjelennek.

Oszlopai:

USERNAME	A felhasználó neve, vagy PUBLIC
GRANTED_ROLE	A felhasználóhoz rendelt szerepkör
ADMIN_OPTION	Jelzi, ha a WITH ADMIN OPTION-nal lettünk hozzárendelve.
DEFAULT_ROLE	Jelzi, hogy a felhasználó alapértelmezett szerepköre-e
OS_GRANTED	Jelzi, hogy az operációs rendszert használja-e a kezelésre

IRODALOMJEGYZÉK / FELHASZNÁLT IRODALOM

1. Kevin Loney: ORACLE database 10g Teljes referencia – Panem Könyvkiadó 2000