Tartalom

[PL/SQL – Áttekintés (1.1) 2](#_Toc195011618)

[🔹 PL/SQL blokk szerkezete (1.2) 2](#_Toc195011619)

[PL/SQL – Változók és nevesített konstansok (1.3) 3](#_Toc195011620)

[DBMS\_OUTPUT csomag – Kiíratás PL/SQL-ben 3](#_Toc195011621)

[PL/SQL – Utasítások (1.4) 3](#_Toc195011622)

[PL/SQL – Összetett adattípusok (1.5) 6](#_Toc195011623)

[🔹 PL/SQL – %TYPE és %ROWTYPE (1.6) 6](#_Toc195011624)

[🔹 SQL utasítások PL/SQL-ben (1.7) 7](#_Toc195011625)

[Kivételkezelés PL/SQL-ben (2.1) 8](#_Toc195011626)

[RAISE Utasítás(2.2): 9](#_Toc195011627)

[Kollekciók (2.3): 10](#_Toc195011628)

[Együttes hozzárendelés (2.4) 14](#_Toc195011629)

[Kurzorok (2.5) 15](#_Toc195011630)

[PL/SQL alprogramok (2.6) 18](#_Toc195011631)

[Sémaszintű alprogramok kezelése adatbázis-objektumokként (2.7) 21](#_Toc195011632)

[Triggerek (2.8) 22](#_Toc195011633)

# PL/SQL – Áttekintés (1.1)

* A **PL/SQL (Procedural Language for SQL)** az SQL és egy procedurális nyelv kombinációja.
* Lehetővé teszi **strukturált, blokkszerű programozást** SQL utasításokkal együtt.
* Szintaxisa hasonlít az **Ada** nyelvre.

**➕ PL/SQL bővítményei az SQL-hez képest:**

* **Változók, típusok**
* **Vezérlési szerkezetek** (pl. IF, LOOP)
* **Alprogramok** (eljárások, függvények), **csomagok**
* **Kurzorok**, kurzorváltozók
* **Kivételkezelés**
* **Objektumorientált elemek**

# 🔹 PL/SQL blokk szerkezete (1.2)

A PL/SQL legkisebb önálló programegysége a **blokk**, ami lehet:

* **Önálló** vagy más blokkba **beágyazott**
* Bárhol használható, ahol végrehajtható utasítás lehet (pl ciklusok, elágazások)

**A blokk 3 fő részből áll:**

1. **Deklarációs rész** *(opcionális)* – változók, típusok, kurzorok stb.
2. **Végrehajtható rész** – itt történik a tényleges programlogika
3. **Kivételkezelő rész** *(opcionális)* – hibák, kivételek kezelése

**Címke**: Egy végrehajtható utasításhoz rendelhető azonosító <<azonosító>> formában.  
**Hatáskör (scope)**: Egy változó **a deklarációjától a blokk végéig** látszik és használható

**PL/SQL – Deklarációs rész**

A deklarációs rész a PL/SQL blokk **első, opcionális** szakasza, ahol a program működéséhez szükséges erőforrásokat (pl. változókat, típusokat) előkészítjük.

**➕ Mit tartalmazhat?**

* **Típusdefiníciók**
* **Változók deklarációja**
* **Nevesített konstansok**
* **Kivételek deklarációja**
* **Kurzorok**
* **Alprogramok (eljárások, függvények)**
* **Pragmák** *(speciális direktívák, pl. hibakezeléshez)*

**A deklaráció szerepe**

* **Tárhelyet foglal** az adott típusú értékeknek
* **Neveket rendel** hozzájuk, hogy később hivatkozhassunk rájuk
* A deklarált objektum **csak a blokk végéig látható**

⚠️ **Fontos szabály**: Minden objektumot előbb **deklarálni kell**, mielőtt használnád!

**Deklaráció helye**

Deklarációk előfordulhatnak:

* **Blokkok** deklarációs részében
* **Alprogramokban** (pl. eljárás, függvény)
* **Csomagokban**

# PL/SQL – Változók és nevesített konstansok (1.3)

**🆔 Azonosítók szabályai**

* Egy változó vagy konstans **neve egy azonosító**, amely:
  + **Betűvel kezdődik**
  + Tartalmazhat **betűt, számjegyet**, illetve **$, \_, #** karaktereket

**Maximum 30 karakter** hosszú lehet, és minden karakter számít

* **Nem érzékeny a kis- és nagybetűkre**

**Változók és konstansok tulajdonságai**

* **Változó**:
  + Ha nincs **kezdeti érték**, akkor automatikusan **NULL**
  + Lehet **NOT NULL**, de akkor **kötelező kezdőértéket** adni
* **Nevesített konstans**:
  + Kötelező **kezdeti értéket megadni**
  + Nem módosítható a futás során
  + Újraértékelődik **minden aktiváláskor**, amikor a blokk vagy alprogram lefut

# **DBMS\_OUTPUT csomag – Kiíratás PL/SQL-ben**

Mivel a PL/SQL **nem tartalmaz beépített I/O (kiíró) utasításokat**, a kimenethez a **DBMS\_OUTPUT** csomagot használjuk a SET SERVEROUTPUT ON-nal.

# PL/SQL – Utasítások (1.4)

A PL/SQL különböző utasítástípusokat támogat, amelyek a program futását irányítják.

**➕ Főbb utasítástípusok:**

* **Üres utasítás:** NULL;
  + Nem hajt végre műveletet, csak átadja a vezérlést a következő utasításnak
  + Hasznos lehet fejlesztés vagy kivételkezelés során
* **Értékadó utasítás:** x := 10;
  + Változó, mező, kollekcióelem vagy objektum attribútum értékét állítja be
  + Az értékadás mindig egy kifejezés alapján történik
* **Ugró utasítás:** GOTO címke;
  + A vezérlést egy címkézett utasításhoz irányítja
* **Elágaztató utasítások:**
  + **Feltételes (IF, ELSIF, ELSE)**
  + **CASE utasítás** (lásd részletesen lejjebb)
* **Ciklusok:**
  + LOOP, WHILE, FOR
* **SQL utasítások:**
  + SQL-parancsok beépítve a PL/SQL-be (pl. SELECT, UPDATE stb.)

**🔹 CASE elágaztató utasítás**

A CASE egy többirányú elágaztatás, mely kétféle módon használható:

**1️. Szelektoros CASE (kifejezés alapján case 2)**

* Egy kifejezés értéke alapján választ a WHEN ágak közül
* Az első egyező WHEN ág fut le
* Ha nincs egyezés és van ELSE ág, az fut le
* ELSE nélkül: **CASE\_NOT\_FOUND** kivétel váltódik ki

**2️. Szelektor nélküli CASE (feltételek alapján case when…)**

* A WHEN ágakban logikai feltételek szerepelnek
* Az első igaz feltételhez tartozó ág hajtódik végre
* Működése egyébként azonos az előző változattal

**PL/SQL – Ciklusok**

A ciklusok ismétlődő végrehajtást tesznek lehetővé. A ciklus **magját** akkor hajtjuk végre, ha a ciklus feltétele ezt engedi. Minden ciklus a mag első utasításának végrehajtásával indul.

**🌀 Ciklusfajták:**

* **Alapciklus (LOOP)**
* **WHILE ciklus**
* **FOR ciklus**
* **Kurzor FOR ciklus** *(külön tárgyalva később)*

**Alapciklus**

* Nem tartalmaz feltételt a fejrésznél
* A ciklus **végtelen** lesz, ha a magban nem szerepel kilépési lehetőség
* Befejezhető: EXIT, GOTO, kivétel

**WHILE ciklus**

* Előfeltételes ciklus: a feltétel kiértékelése megelőzi a mag futását
* Ha a feltétel az első kiértékeléskor hamis vagy NULL, a ciklusmag **nem hajtódik végre**
* Ha mindig igaz, és nem módosítjuk a magban, **végtelen ciklus** alakulhat ki

**FOR ciklus**

* Meghatározott értékkészleten fut végig
* A ciklusváltozó **PLS\_INTEGER** típusú és csak a cikluson belül érvényes
* Az **alsó és felső határ** egyszer kerül kiértékelésre, a ciklus indulása előtt
* **REVERSE** kulcsszóval a ciklus **csökkenő sorrendben** is végigfuthat
* A ciklusváltozót **nem lehet módosítani**

**EXIT utasítás**

* **Csak ciklusmagon belül használható**
* A ciklus futását leállítja, a vezérlés a ciklus utáni részre kerül
* Alakjai:
  + EXIT;
  + EXIT WHEN feltétel;
  + EXIT <<címke>> [WHEN feltétel];
* Címkével egymásba ágyazott ciklusok közül is kijelölhető, melyikből lépjünk ki

**CONTINUE utasítás**

* A ciklus aktuális iterációját megszakítja, és a **következő iterációval folytatja**
* Alakjai:
  + CONTINUE;
  + CONTINUE WHEN feltétel;
  + CONTINUE <<címke>> [WHEN feltétel];
* Csak akkor lép tovább, ha a **feltétel igaz**

# PL/SQL – Összetett adattípusok (1.5)

Az összetett adattípusok lehetővé teszik többféle adat csoportosítását egyetlen logikai egységbe. PL/SQL-ben a következő összetett adattípusok léteznek:

**🗂️ Összetett adattípusok típusai:**

* **Rekordtípus**
* **Kollekciótípusok** *(pl. asszociatív tömb, beágyazott tábla, dinamikus tömb)*

**🔸 Rekordtípus**

* A rekord egy logikailag összetartozó adatok heterogén csoportja, ahol a mezők különböző típusú adatok tárolására alkalmasak.
* **Minden mezőnek** saját neve és típusa van, és az adatok különböző típusokat képviselhetnek (pl. számok, szövegek, dátumok stb.).
* Lehetővé teszi, hogy különböző típusú adatokat egyetlen **egységként** kezeljünk.
* A rekordot gyakran használjuk **adatbázis-táblák sorainak kezelésére**.

# 🔹 PL/SQL – %TYPE és %ROWTYPE (1.6)

**🧩 %TYPE**

* A **%TYPE** egy speciális PL/SQL szintaxis, amely lehetővé teszi, hogy a változó vagy konstans típusát egy már létező adatbázis-elem, rekord, mező, kollekció, kurzorváltozó stb. típusával egyeztesd.
* **Előnyei:**
  + Nem szükséges pontosan ismerni az átvett típus részleteit, elegendő a hivatkozott elem.
  + Ha az elem típusa változik, a változó típusa automatikusan követi ezt a változást futás közben.
* **Öröklődik:**
  + Típus, méret és megszorítások (ha nem oszlopról van szó).
* **Nem öröklődik**:
  + A kezdőérték, tehát azt manuálisan kell megadni.

**🧩 %ROWTYPE**

* A **%ROWTYPE** lehetővé teszi, hogy egy rekordtípust hozz létre, amely egy **adatbázis-tábla vagy nézet** egy teljes vagy részleges sorát reprezentálja.
* **Jellemzők:**
  + Minden egyes oszlopnak egy megfelelő mezője lesz a rekordban, a mezők neve és típusa megegyezik az oszlopokéval.
  + Ha a tábla struktúrája változik, a rekord mezői is automatikusan módosulnak.
  + A rekord mezői **nem öröklik** az oszlopok megszorításait és kezdőértékeit.

# 🔹 SQL utasítások PL/SQL-ben (1.7)

* A PL/SQL programokba közvetlenül beépíthetők az SQL utasítások:
  + **DQL** (adatlekérdezés)
  + **DML** (adatkezelés)
  + **TCL** (transzakciókezelés)
  + **bárhol használhatóak, ahol végrehajtó utasítás áll**
* **Statisztikus SQL:** Az SQL utasítások szövege fordítási időben ismert, a PL/SQL-fordító a megszokott módon kezeli őket.
* **Dinamikus SQL (natív):** Ha dinamikus SQL-t kell alkalmazni, a PL/SQL egy külön eszközt biztosít ennek kezelésére, lehetővé téve a rugalmas SQL utasítások futtatását.

**🔹 PL/SQL – SELECT INTO és DELETE**

**🧩 SELECT INTO**

* A **SELECT INTO** utasítás lehetővé teszi, hogy adatokat kérdezzünk le az adatbázisból és azokat változókba vagy rekordokba helyezzük.
* **Használat:**
  + Az **INTO** részben minden egyes kifejezéshez **kompatibilis típusú** változót kell rendelni, vagy a rekordnak megfelelő mezőkkel kell rendelkeznie.
  + **Fontos:** Az utasításnak pontosan **egy sor** eredményt kell visszaadnia.
* **Lehetséges kivételek:**
  + **TOO\_MANY\_ROWS:** Ha az eredményhalmaz több mint egy sort tartalmaz.
  + **NO\_DATA\_FOUND:** Ha az eredményhalmaz üres.

**🧩 DELETE**

* A **DELETE** utasítás a rekordok törlésére szolgál.
* **RETURNING** utasításrész:
  + Lehetővé teszi, hogy a törölt sorok alapján számított értékeket visszakapjunk.
  + Nem szükséges előzőleg egy **SELECT** segítségével kiszámítani ezeket az értékeket.
  + Az értékeket **változókban** vagy **rekordban** tárolhatjuk.

# Kivételkezelés PL/SQL-ben (2.1)

A PL/SQL-ben a kivételkezelés célja, hogy a program futása közben előforduló hibákat megfelelően kezeljük. Két fő típusú kivétel létezik: **beépített** és **felhasználói**.

**🛠 Kivételfajták:**

1. **Beépített kivételek** (azokat a rendszer váltja ki):
   * **Előre definiált kivételek**: Ezek a hibák előre meghatározottak a PL/SQL rendszerében (pl. NO\_DATA\_FOUND, TOO\_MANY\_ROWS).
   * **Nem előre definiált kivételek**: Olyan hibák, amelyek a futtató rendszer által generáltak, de nem előre definiáltak.
2. **Felhasználói kivételek**:
   * A felhasználó által explicit módon meghatározott kivételek, amelyeket a programban egyedi helyzetekben indítanak el (pl. RAISE utasítással).

**🧑‍💻 SQLCODE és SQLERRM**

A kivételkezelés során két beépített függvényt használhatunk, hogy információkat szerezzünk a legutóbbi kivételről:

* **SQLCODE**:
  + A legutoljára bekövetkezett kivétel kódját adja vissza.
  + Lehetséges értékek:
    - **0**: Ha nem történt kivétel (minden rendben van).
    - **1**: Ha felhasználói kivétel történt.
    - **+100**: Ha a **NO\_DATA\_FOUND** beépített kivétel történt.
    - **Negatív szám**: Bármely más beépített kivétel esetén (pl. -1 vagy -20001).
* **SQLERRM**:
  + A legutoljára bekövetkezett kivételhez tartozó hibaüzenetet adja vissza.
  + Az opcionális (k) paraméter segítségével a kódú kivételhez tartozó hibaüzenet is lekérdezhető.

**💡 Fontos tudnivalók:**

* Minden kivételnek van neve, kódja és szövege.
* A kivételkezelésben a BEGIN...EXCEPTION...END blokkot használjuk a hibák kezelésére

**Kivételkezelés PL/SQL-ben**

**Kivétel előfordulása:**

1. **Kivétel bekövetkezése**:
   * Ha egy kivétel történik a végrehajtható részben, a végrehajtás félbeszakad.
   * A futtató rendszer megvizsgálja, hogy van-e a blokk végén kivételkezelő, és annak egyik **WHEN** ága nevesíti-e a kivételt.
     + Ha van megfelelő kivétel, akkor a kivételkezelő **THEN** ágában lévő utasítások futnak le.
     + Ha nincs megfelelő kivétel, és van **WHEN OTHERS** ág, akkor annak utasításai futnak le.
     + Ha nincs kivételkezelő, akkor a kivétel továbbadódik az aktiváló környezetnek.
     + Ha nem kezelik a kivételt, akkor az **UNHANDLED\_EXCEPTION** kivétel lép életbe.
2. **Kivételkezelés a deklarációs részben**:
   * Ha a kivétel a deklarációs részben vagy kivételkezelőben váltódik ki, akkor az azonnal továbbadódik a környezetnek.

**Kivételkezelési technika:**

1. **Előre definiált kivételek**:
   * Az előre definiált kivételek esetében a kivételt a EXCEPTION WHEN szintaxissal kezelhetjük. Ezek olyan hibák, amelyeket a rendszer automatikusan generál, például adatbázis-hibák vagy logikai hibák.
2. **Nem előre definiált kivételek**:
   * Ezek esetében először deklaráljuk a kivételt, majd a rendszer által generált hiba kódját hozzárendeljük a kivételhez. Így a hiba kódjától függően egy adott kivételt kezelhetünk.
3. **Felhasználó által definiált kivételek**:
   * A felhasználó által definiált kivételek lehetővé teszik, hogy saját hibákat generáljunk és kezeljünk a kód futása közben. A kivételt a kódban szükség szerint hívhatjuk elő, ha valamilyen nem kívánt állapotot észlelünk.

# RAISE Utasítás(2.2):

A **RAISE** utasítást arra használjuk, hogy egy meglévő kivételt újra aktiváljunk, vagy egy másik kivételt generáljunk. Ez különösen hasznos akkor, ha egy kivételt már kezeltek, de azt tovább szeretnénk adni egy magasabb szintű kivételkezelőnek.

* **Kivétel újbóli kiváltása**:
  + Ha egy kivételt lokálisan kezelünk, de tovább szeretnénk adni, akkor a RAISE utasítással az aktuálisan kezelt kivételt továbbadhatjuk.

**A kivételkezelő működése:**

1. **Kivétel kiváltása**:
   * A kivétel akkor váltódik ki, ha valamilyen nem várt állapot, hiba lép fel a program futása közben. Ezek lehetnek rendszer által előre definiált hibák (pl. oszlop nem található) vagy felhasználó által előidézett problémák.
2. **Kivételkezelő ága**:
   * Amikor egy kivétel bekövetkezik, a rendszer megnézi, hogy van-e kivételkezelő ága a programban. A kivételkezelő olyan utasításokat tartalmaz, amelyek a hibák kezelésére szolgálnak. Ha a kivétel kezelésére nincs megfelelő ág, akkor a kivétel továbbadódik a külső környezetnek.
3. **Továbbadás**:
   * Ha a kivételt nem kezeljük megfelelően a helyi környezetben, akkor a kivétel visszakerül a hívó környezethez. Ez a kivétel továbbadásának egyik formája, és lehetőséget ad arra, hogy a hívó környezetben történjen meg a hiba megfelelő kezelése.

# Kollekciók (2.3):

* **Definíció**: Azonos típusú adatelemek rendezett együttese, minden elemnek egyedi indexe van.
* **PL/SQL kollekciótípusok**:
  + Asszociatív tömb
  + Beágyazott tábla
  + Dinamikus tömb
* **Típusok**: Az elemek bármilyen PL/SQL típusúak lehetnek (kivéve REF CURSOR).
* **Felhasználás**: Egydimenziós és többdimenziós tömböket is kezelhetünk. Kollekciók adatbázisokkal történő adatmozgathatásra is alkalmasak. A 3GL nyelvek tömb fogalmának felelnek meg.

Kollekciók létrehozása:

1. Kollekciótípus létrehozása
2. Kollekcióváltozó deklarálása

**Metódusok**:

* **EXISTS:** Igaz értéket ad vissza, ha az adott indexű elem létezik a kollekcióban.
* **COUNT:** A kollekció elemeinek számát adja vissza.
* **LIMIT:** A kollekció maximális méretét adja vissza.
* **FIRST, LAST:** Az első és utolsó elem indexét adja vissza. Ha nincs ilyen, NULL-t ad.
* **NEXT, PRIOR:** Egy adott indexű elem előző vagy következő elemének indexét adja vissza. Ha nincs ilyen, NULL-t ad.
* **EXTEND:** Bővíti a kollekciót.
* **TRIM:** Eltávolítja a kollekció utolsó elemeit.
* **DELETE:** Törli a megadott elemeket a kollekcióból.

**Dinamikus tömb:**

* **Deklarálás**: A dinamikus tömb típusát VARRAY vagy VARYING ARRAY kulcsszóval lehet létrehozni, megadva a maximális méretet és az elemtípust. Az indexelés 1-től kezdődik.
* **Méret**: A tömb maximális mérete rögzített, de a tényleges elem szám változó lehet, 0 és a maximális méret között.
* **Műveletek**: Új elemek hozzáadhatók, de a maximális méretet nem léphetik túl. Az elemek törlése csak a végéről lehetséges.
* **Inicializálás**: A dinamikus tömb egy referencia, amelynek kezdőértéke NULL. A tömb példányosítása a típus konstruktorának meghívásával történik. A konstruktor paramétereinek száma tetszőleges lehet, de összhangban kell lenniük az elemek típusával.
* **Egyéb jellemzők**: Elemei lehetnek objektumtípusok, és a dinamikus tömb használható objektumtípus attribútumaként vagy adatbázistábla oszlopaként. A tömb NULL értéke tesztelhető (IS NULL).

**Dinamikus tömb - Kollekciómetódusok:**

* **EXISTS(i)**: Ellenőrzi, hogy létezik-e az i indexű elem a kollekcióban.
* **COUNT**: Visszaadja a kollekcióban lévő elemek számát.
* **LIMIT**: Visszaadja a kollekció maximális méretét.
* **FIRST, LAST**: Visszaadják az első és utolsó elem indexét.
* **NEXT(i), PRIOR(i)**: Visszaadják az i indexű elem következő és előző elemének indexét.
* **EXTEND, EXTEND(n), EXTEND(n,m)**:
  + Paraméter nélküli **EXTEND** egyetlen NULL elemet ad hozzá a kollekció végéhez.
  + **EXTEND(n)** n darab NULL elemet ad hozzá.
  + **EXTEND(n,m)** az m indexű elemet n-szer helyezi el a kollekció végén.
* **TRIM, TRIM(n)**:
  + A **TRIM** egy elemet távolít el a kollekció végéről.
  + **TRIM(n)** az utolsó n elemet távolítja el. Ha n > COUNT, akkor a SUBSCRIPT\_BEYOND\_COUNT kivétel keletkezik.
* **DELETE**: A paraméter nélküli **DELETE** törli a kollekció összes elemét.

**Dinamikus tömb - Kivételek:**

* **COLLECTION\_IS\_NULL**: NULL értékű kollekcióra történő metódus meghívásakor (kivéve **EXISTS**).
* **SUBSCRIPT\_BEYOND\_COUNT**: Ha olyan indexre hivatkozunk, amely nagyobb a kollekció elemszámánál.
* **SUBSCRIPT\_OUTSIDE\_LIMIT**: Ha az index érvénytelen, pl. -1.
* **VALUE\_ERROR**: Ha az index NULL vagy nem konvertálható a kulcs típusára.

**Asszociatív tömb**

**Alapfogalom**

Az asszociatív tömb (más néven indexelt tábla) egy kulcs-érték párokat tartalmazó kollekció (hash tábla), amely csak PL/SQL-ben használható. Dinamikusan kezelhető, az indexeknek nincs rögzített alsó vagy felső határa.

**Lehetséges index típusok**

* PLS\_INTEGER
* BINARY\_INTEGER
* VARCHAR2(n)
* STRING(n)

**Működés**

Egy i indexű elem értékadásával az elem automatikusan létrejön (ha korábban nem létezett), vagy felülíródik (ha már létezett). Az indexnek nem szabad NULL értékűnek lennie, és konvertálhatónak kell lennie a kulcs típusára – ellenkező esetben VALUE\_ERROR kivétel keletkezik.

**Fontos megkötések**

* Az asszociatív tömb **nem használható adatbázistábla oszlopának típusaként**.
* Nem lehet **adatbázis objektum** típusa.
* Kizárólag PL/SQL kódban alkalmazható.

**Kollekció metódusok**

* EXISTS(i) – Megvizsgálja, hogy az i indexű elem létezik-e.
* COUNT – Megadja az elemek számát.
* LIMIT – A kollekció maximális mérete (asszociatív tömbnél NULL).
* FIRST, LAST – Első és utolsó index lekérdezése.
* NEXT(i), PRIOR(i) – Következő, illetve előző index i-hez képest.
* DELETE – Elemtörlés (részletezve alább).

**Törlés (DELETE) használata**

* DELETE – Az összes elem törlése, azaz a kollekció kiürítése.
* DELETE(i) – Csak az i indexű elem törlése.
* DELETE(i, j) – Az i és j index közötti összes elem törlése.  
  Megjegyzés: Ha i > j vagy bármelyik paraméter NULL, akkor nem történik semmi.

**Kivételkezelés**

* NO\_DATA\_FOUND – Nem létező elemre történő hivatkozás esetén lép fel.
* VALUE\_ERROR – Ha az index NULL, vagy nem konvertálható a kulcs típusára.

**Beágyazott tábla**

**Alapfogalom**

A beágyazott tábla egy speciális PL/SQL kollekció, amely objektumtípus. Elemei egész típusú indexekkel érhetők el, az alsó határ mindig 1, a felső határ nincs rögzítve. Az egyes elemek törölhetők, de a helyükön „lyuk” marad – a kollekció nem tömörödik automatikusan.

**Deklaráció és inicializálás**

A beágyazott tábla deklarálásakor egy referencia jön létre, amely alapértelmezés szerint NULL. Az inicializálásához példányosítani kell, mégpedig a kollekció típusának megfelelő konstruktor hívásával. A konstruktor hívható paraméterekkel is, ha a típus kompatibilis a kollekció elemeinek típusával.

**Használhatóság**

* A beágyazott tábla **lehet objektumtípus attribútuma**.
* Lehet **adatbázistábla oszlopának típusa**.
* Lehet **objektumtípus példányainak kollekciója**.
* A IS NULL segítségével tesztelhető, hogy a kollekció inicializálva lett-e.
* Két beágyazott tábla **egyenlősége vizsgálható**, ha azonos típusúak és az elemeik összehasonlíthatók. **Rendezettségi viszony** azonban nem vizsgálható köztük.

**Kollekciómetódusok**

**📌 Általános metódusok:**

* EXISTS(i) – Megvizsgálja, létezik-e az i indexű elem.
* COUNT – Az aktuálisan meglévő elemek száma.
* LIMIT – A maximális elemszám (NULL érték).
* FIRST, LAST – Az első és utolsó létező index.
* NEXT(i), PRIOR(i) – A következő és előző létező index i-hez képest.

**📌 Bővítés – EXTEND:**

* EXTEND – Egyetlen NULL elemet ad a kollekció végéhez.
* EXTEND(n) – n darab NULL elem hozzáadása.
* EXTEND(n, m) – Az m indexű elem n példányban történő hozzáadása a végéhez.

A törölt elemek helyét megtartja a PL/SQL, így azokra új érték adható. A bővítés mindig az utolsó aktív elem után történik.

**📌 Törlés – DELETE:**

* DELETE – Minden elem törlése (kollekció kiürítése).
* DELETE(i) – Az i indexű elem törlése.
* DELETE(i, j) – Az i és j közötti (beleértve i-t és j-t is) elemek törlése.

A törlés „lyukat” hagy a kollekcióban – az indexek felszabadulnak, de a memóriafoglalás megmarad.

**📌 Vágás – TRIM:**

* TRIM – Az utolsó elem eltávolítása.
* TRIM(n) – Az utolsó n elem eltávolítása.

Ha n > COUNT, akkor SUBSCRIPT\_BEYOND\_COUNT kivétel keletkezik. A TRIM által eltávolított elemek nem visszaállíthatók.

**Törölt elemek hatása**

* A törölt elemekre történő hivatkozás NO\_DATA\_FOUND kivételt eredményez.
* A törölt indexeket a FIRST, LAST, NEXT, PRIOR metódusok **figyelmen kívül hagyják**.
* A COUNT az aktuálisan elérhető elemek számát adja vissza (törölteket nem számolja).
* Az elemek újra felhasználhatók értékadás után.

**Kivételkezelés:**

 **COLLECTION\_IS\_NULL**  
Akkor keletkezik, ha egy **nem inicializált (NULL)** kollekcióra próbálunk metódust hívni.

Kivétel: az EXISTS(index) metódus hívható NULL kollekció esetén is.

 **SUBSCRIPT\_BEYOND\_COUNT**  
Akkor fordul elő, ha az elemszámot meghaladó indexű elemre hivatkozunk.

Például: TRIM(n) esetén, ha n > COUNT.

 **SUBSCRIPT\_OUTSIDE\_LIMIT**  
Olyan index hivatkozásnál keletkezik, amely **kívül esik az érvényes tartományon** (pl. negatív index: -1).

 **NO\_DATA\_FOUND**  
Akkor jelentkezik, ha egy **törölt vagy sosem létezett elemre** történik hivatkozás.

 **VALUE\_ERROR**  
Akkor dobódik, ha az index értéke NULL, vagy **nem konvertálható** az elvárt kulcstípusra.

# Együttes hozzárendelés (2.4)

**Alapfogalom**

A hozzárendelés az a művelet, amely során egy PL/SQL változó értéket kap egy SQL utasításból.  
Az együttes hozzárendelés (bulk binding) lehetővé teszi, hogy egy kollekció összes elemét egyszerre töltsük fel vagy frissítsük – ez jelentősen javítja a teljesítményt, mivel csökkenti a PL/SQL és SQL motor közötti kontextusváltások számát.

**Típusai**

* **FORALL** – A PL/SQL oldalon történő együttes műveletekhez (DML műveletek: INSERT, UPDATE, DELETE).
* **BULK COLLECT** – Az SQL oldalról történő együttes lekérdezéshez (SELECT, FETCH, stb.).

**BULK COLLECT – SQL-oldali együttes hozzárendelés**

**Használat:**

* A SELECT, INSERT, UPDATE, DELETE, és FETCH utasítások INTO részében alkalmazható.
* Az SQL motor egyszerre tölti fel a megadott kollekciókat.

**Jellemzők:**

* A kollekciókat nem szükséges előre inicializálni.
* Az adatokat 1-es indextől kezdve tölti fel sorban.
* A korábban már betöltött adatokat felülírja.
* Csak szerveroldali PL/SQL programokban használható (pl. eljárások, függvények, triggerek).

**FORALL**

**Alapfogalom**

A FORALL a PL/SQL oldali együttes hozzárendelés eszköze, amely lehetővé teszi több DML (INSERT, UPDATE, DELETE) utasítás hatékony végrehajtását kollekciók alapján.  
Az SQL utasítás minden megadott indexre egyszer hajtódik végre, jelentősen javítva a teljesítményt.

**részletes magyarázat**

* A FORALL ciklus automatikusan deklarálja az index változót, amely típusban megegyezik a kollekció indexével.

**Háromféle indexelési lehetőség:**

1. alsó\_határ .. felső\_határ  
   – Az index egy fix numerikus tartományon halad végig.
2. INDICES OF kollekció  
   – Az index a megadott kollekció létező indexeit veszi fel.  
   – Opcionálisan BETWEEN alsó AND felső megadásával szűkíthető.
3. VALUES OF indexkollekció  
   – Az indexértékeket egy külön kollekció (indexkollekció) tartalmazza.

**A sql\_utasítás típusa:**

* Csak INSERT, UPDATE vagy DELETE lehet.
* Az utasítás a kollekció elemeit hivatkozza a WHERE, SET, vagy VALUES részben.

**Hibatűrés – Kivételkezelés**

* Ha a FORALL ciklusban lévő SQL utasítás nem kezelt kivételt vált ki, akkor a teljes FORALL ciklus visszagörgetődik.
* Ha a kivételt kezeljük, akkor a kivételt kiváltó végrehajtás külön visszagörgetődik, de a korábban sikeresen lefutott utasítások érvényben maradnak.

A SAVE EXCEPTIONS opció lehetővé teszi, hogy a kivételeket később, összegyűjtve kezeljük (SQL%BULK\_EXCEPTIONS kollekció segítségével).

# Kurzorok (2.5)

**Mi az a kurzor?**

A **kurzor** egy olyan PL/SQL eszköz, amely lehetővé teszi, hogy egy **SQL lekérdezés eredményét soronként dolgozzuk fel**.  
Kapcsolódik a **környezeti területhez** (memory area), amely tartalmaz:

* A lekérdezés vagy utasítás **belső reprezentációját** (utasításstruktúra)
* A **feldolgozott sorokat** (eredményhalmazt), ha SELECT utasításról van szó  
  – Ezek a sorok alkotják az **aktív halmazt** (active set)

**Kurzor szerepe**

A kurzor segítségével:

* Nevet adhatunk a környezeti területnek
* Hozzáférhetünk az ott lévő adatokhoz
* **Több soros eredményhalmaz** esetén a sorokat egyenként feldolgozhatjuk

**Kurzor típusok**

**1. Explicit kurzor**

* Kifejezetten a felhasználó által létrehozott és vezérelt kurzor
* Akkor használjuk, ha a **lekérdezés több sort** ad vissza
* Lépései: deklarálás → megnyitás → sorok beolvasása → lezárás

**2. Implicit kurzor**

* A PL/SQL automatikusan létrehozza minden **egyszerű SQL utasítás** (SELECT INTO, INSERT, UPDATE, DELETE) végrehajtásakor
* Nem kell külön deklarálni vagy kezelni

**3. Rejtett kurzor (Cursor FOR Loop)**

* Olyan **kurzor FOR ciklus**, amely automatikusan kezeli a kurzor deklarálását, megnyitását, olvasását és lezárását
* Kifejezetten kényelmes és tömör szintaxist biztosít a több soros eredmények feldolgozásához

**Explicit kurzor – Életciklus**

Az explicit kurzorok használata során a lekérdezés eredményeinek feldolgozása **négy fő lépésből** áll: deklarálás, megnyitás, betöltés, lezárás.

**1. Megnyitás (OPEN)**

* A kurzor megnyitásakor lefut a kurzorhoz rendelt **SELECT lekérdezés**, létrejön az **aktív halmaz**, és a kurzormutató az **első sorra** mutat.
* Egy már megnyitott kurzort **nem lehet újra megnyitni**.
  + Újramegnyitás esetén: **CURSOR\_ALREADY\_OPEN** kivétel.
* A megnyitott kurzor **nem használható kurzor FOR ciklusban**.

**2. Betöltés (FETCH)**

* A FETCH utasítás segítségével az aktív halmaz **egy-egy sorát töltjük be**:
  + Vagy **skalárváltozókba** (pl. v\_name, v\_salary)
  + Vagy egy **rekord változóba**
* A FETCH automatikusan a **következő sorra** lépteti a kurzormutatót.
* A változók típusának kompatibilisnek kell lennie a lekérdezés által visszaadott oszlopok típusaival.
* A változók (vagy rekord mezők) számának **meg kell egyeznie** az oszlopok számával.

**Speciális esetek:**

* Nem megnyitott kurzor esetén: **INVALID\_CURSOR** kivétel
* Ha az utolsó sor után történik FETCH, a változók **nem változnak meg**
* Nem létező sor FETCH-elése **nem vált ki kivételt**
  + Ilyen esetekben a **%FOUND, %NOTFOUND** kurzorattribútumokat használjuk az ellenőrzéshez

**3. Lezárás (CLOSE)**

* A CLOSE utasítás:
  + **Megszünteti** a kapcsolatot az aktív halmaz és a kurzor között
  + **Érvényteleníti** a kurzormutatót
* Csak **megnyitott kurzort** lehet lezárni.
  + Ha a kurzor nincs megnyitva: **INVALID\_CURSOR** kivétel

**Kurzor attribútumok**

A kurzor attribútumok segítségével információkat nyerhetünk ki a **DQL (SELECT)** és **DML (INSERT, UPDATE, DELETE)** utasítások végrehajtásáról. Ezek az attribútumok **csak PL/SQL procedurális utasításokban** használhatók, **SQL utasításokban nem**.

**Elérhető attribútumok:**

* **%FOUND**
  + Értéke:
    - **NULL** → a kurzor megnyitása után, de még az első FETCH előtt
    - **TRUE** → ha a FETCH utasítás legalább egy sort sikeresen betöltött
    - **FALSE** → ha nem történt sikeres betöltés
* **%NOTFOUND**
  + A **%FOUND logikai negáltja**
  + TRUE, ha **nincs több sor** vagy nem történt sikeres betöltés
* **%ISOPEN**
  + TRUE, ha a kurzor **nyitva van**
  + FALSE, ha **nem nyitották meg**, vagy már **lezárták**
* **%ROWCOUNT**
  + A sikeresen betöltött sorok **számát adja meg**
  + Értéke:
    - **0** a megnyitás után, de első FETCH előtt
    - Minden sikeres FETCH után **1-gyel nő**

**Kurzor FOR ciklus**

A **kurzor FOR ciklus** egy egyszerűsített módja az **explicit kurzorok** kezelésének, amely automatikusan végrehajtja a kurzor életciklusának minden lépését: **megnyitás**, **betöltés**, **lezárás**.

**Jellemzők:**

* A ciklus **implicit módon megnyitja** a lekérdezéshez tartozó kurzort.
* **Automatikusan betölti** az aktív halmaz összes sorát.
* A ciklus végén **lezárja a kurzort**.
* A **ciklusváltozó** típusa automatikusan: kurzornév%ROWTYPE

# PL/SQL alprogramok (2.6)

* **Alprogramok:** A PL/SQL alprogramok nevesített PL/SQL blokkok, amelyek paraméterek halmazával hívhatók meg, ha azok rendelkeznek paraméterekkel. Az alprogramok lehetnek **eljárások** vagy **függvények**.

**Alprogram típusok:**

* **Eljárás:** Általában egy művelet végrehajtására szolgál.
* **Függvény:** Általában egy érték kiszámítására és visszaadására használják.

**Alprogramok elhelyezése:**

* **Blokkba ágyazva:** Az alprogramok más alprogramokban is szerepelhetnek.
* **Séma szinten:** Alprogramok a sémák szintjén is létrehozhatók.
* **Csomagban:** Az alprogramok csomagok részeként is deklarálhatók.

**Alprogram deklaráció:**

* Az alprogramok deklarációját csak az összes többi objektum deklarációja után helyezhetjük el.
* A **pragma** deklarációt a specifikáció után követheti.

**Rekurzivitás:**

* Az alprogramok **rekurzívan hívhatók**.

**Az alprogramok felépítése:**

* Az alprogramoknak két fő része van:
  1. **Specifikáció:** A paraméterek és a visszatérési értékek meghatározása.
  2. **Törzs:** A végrehajtható kód, amely megegyezik egy névtelen blokkal a DECLARE kulcsszó elhagyásával.

**Eljárások:**

* Az eljárásokat utasításszerűen tudjuk meghívni.
* Az eljáráshívás bárhol szerepelhet a programban, ahol végrehajtható utasítások állnak.
* **Hívás formája:** név és aktuális paraméter lista.
* Az eljárás befejeződik, ha elfogynak a végrehajtható utasításai, vagy a **RETURN** utasítást hajtják végre.
* Eljárás nem fejeztethető be **GOTO** utasítással.

**Függvények:**

* A **RETURN** alapszó után a függvény **visszatérési értékének típusát** határozza meg a specifikációban.
* Függvényt **CSAK** **kifejezésben** lehet meghívni.
* **Hívás formája:** név és aktuális paraméter lista.
* A függvény törzsében **legalább egy RETURN utasításnak** szerepelnie kell. Ha nincs, **PROGRAM\_ERROR** kivételt vált ki.
* A **RETURN** utasítás meghatározza a visszatérési értéket

**Formális paraméterlista**

**Paraméterátadás módok:**

* IN: Érték szerinti paraméterátadás (de nincs értékmásolás).
* OUT: Eredmény szerinti paraméterátadás.
* IN OUT: Érték-eredmény szerinti paraméterátadás.
* NOCOPY: Ajánlás a fordítónak, hogy OUT és IN OUT esetén se másoljon értéket.

**Paraméter-összerendelés:**

* A paramétereket pozíció és/vagy név szerint is hozzárendelhetjük.

A lokális és csomagbeli alprogramnevek túlterhelhetők.

**Paraméterek típusai és alapértelmezett értékei**

**IN:** A paraméter értékét nem módosíthatjuk. Az **IN** módú paraméterek nem kaphatnak értéket az alprogram törzsében.

**OUT:** Az **OUT** módú paraméterek automatikusan NULL értékkel inicializálódnak.

**IN OUT:** Az **IN OUT** módú paraméterek értéke módosítható a függvényhívás során.

**Kezdőértékek:**

* Az **IN (alapértelemzett)** módú paraméterekhez kezdőérték adható egy kifejezés segítségével (**:=** vagy **DEFAULT**).
* Az **OUT** módú formális paraméter automatikus kezdőértéke **NULL**.

**Formális és aktuális paraméterek hozzárendelése**

* **Egymáshoz rendelés módjai:**
  + **Pozíció szerinti kötés:** A formális és aktuális paraméterek sorrendje a döntő.
  + **Név szerinti kötés:** Az aktuális paraméterek listájában tetszőleges sorrendben felsorolhatjuk a formális paraméterek nevét, majd a **=>** jelkombináció után megadjuk a megfelelő aktuális paramétert.
  + **Keverve:** Először pozíció szerinti, majd név szerinti összerendelés.
* **Aktuális paraméterek száma:**
  + Az aktuális paraméterek száma lehet kevesebb, mint a formális paraméterek száma.
  + Ez az **IN** módú paraméterek kezdőértékétől függ.
  + Ha egy formális paraméternek van kezdőértéke, akkor nem szükséges aktuális paramétert megadni.
  + Ha megadjuk az aktuális paramétert, akkor az inicializálás az aktuális paraméter értékével történik. Ha nem adunk meg értéket, akkor a formális paraméter listán szereplő kezdőérték kerül felhasználásra.

A **formális paraméterek** a függvények és eljárások deklarációjában szereplő változók, amelyeken keresztül az értékek beérkeznek (paraméterátadás). A paraméterek három fő módja létezik:

**1. IN paraméterek:**

* **Alapértelmezett típus**, ha nem adunk meg egyértelműen paraméterátadási módot. Tehát ha nem adjuk meg az IN, OUT vagy IN OUT módot, akkor automatikusan IN típusú paramétert jelent.
* **Használat**: Az IN paraméterek kizárólag olvashatók a programon belül, és nem módosíthatók. A paraméter értékét a hívás során átadjuk, de nem tudjuk megváltoztatni az alprogram törzsében.
* **Nevesített konstansként kezelhetők**: Az alprogram törzsében az IN paraméterek csak értékekként jelennek meg, nem változók.
* **Kezdőérték adható**: Az IN paramétereknek **kezdőérték** adható, ha nem kapnak értéket a híváskor.

**OUT paraméterek:**

* **Használat**: Az OUT paraméterek kimeneti paraméterek, vagyis ezekben a paraméterekben értéket adunk vissza a hívó környezetnek. Az alprogram belsejében az OUT paraméterek nem használhatók bemeneti értékként.
* **Automatikus kezdőérték**: Az OUT paraméterek alapértelmezett kezdőértéke **NULL**, tehát ha nem állítjuk be őket egy értékre az alprogramon belül, akkor NULL értékkel térnek vissza.
* **Változóként kezelhetők**: Az OUT paraméterek változókként kezelhetők az alprogram törzsében, és ezek módosíthatók.
* **Nem lehet IN kifejezés**: Az OUT paraméterek nem használhatók kifejezésként, csak **változóként**.

**3. IN OUT paraméterek:**

* **Használat**: Az IN OUT paraméterek mind bemeneti, mind kimeneti paraméterek, tehát az alprogram mind olvashatja, mind módosíthatja őket. Az értéküket az alprogram módosíthatja, és a hívó környezet számára visszaadja az új értéket.
* **Kezdőérték adható**: Az IN OUT paraméterekhez is adhatunk kezdőértéket, ahogy az IN paraméterekhez.
* **Inicializált változóként kezelhetők**: Az IN OUT paraméterek változókként kezelhetők, amelyek kezdeti értékkel rendelkeznek a híváskor.
* **Kifejezésként és változóként is kezelhetők**: Az IN OUT paraméterek egyszerre lehetnek kifejezésként és változóként is használhatók.

**Alprogramok túlterhelése**

* **Túlterhelt alprogramok:** Az alprogramok neve azonos, de a formális paraméterek száma, típusa, vagy sorrendje eltérő kell legyen.
* A fordító az aktuális paraméter lista alapján választja ki a megfelelő törzset a hívás során.

A **sémaszintű alprogramok** PLSQL-ben olyan eljárások vagy függvények, amelyeket az adatbázis szintjén tárolunk, és azokat SQL-es DDL (Data Definition Language) utasításokkal kezelhetjük. Ez lehetővé teszi, hogy az alprogramokat hosszú távon, a rendszer teljes életciklusa alatt használjuk és karbantartsuk. A sémaszintű alprogramok tárolása **lefordított formában**, ún. **M-kódban történik**, ami az Oracle belső végrehajtási formája.

# Sémaszintű alprogramok kezelése adatbázis-objektumokként (2.7)

* Az alprogramokat (függvényeket és eljárásokat) **adatbázis-objektumként** kezelhetjük. Ez azt jelenti, hogy a PLSQL-ben írt eljárások és függvények tárolhatóak az adatbázisban, és az adatbázis DDL parancsain keresztül kezelhetjük őket.
  + **Létrehozás**: Az alprogramokat **CREATE** kulcsszóval hozhatjuk létre.
  + **Módosítás**: Az alprogramokat **ALTER** kulcsszóval módosíthatjuk.
  + **Törlés**: Az alprogramokat **DROP** kulcsszóval törölhetjük.

**OR REPLACE**

* **OR REPLACE** egy fontos kulcsszó, amely lehetővé teszi, hogy **újrageneráljuk** az alprogramot, ha az már létezik.
  + Ha egy eljárás vagy függvény már létezik, és szeretnénk módosítani, akkor nem kell manuálisan törölni a régi objektumot. Az **OR REPLACE** lehetőséggel az alprogram automatikusan újra létrejön a módosított definícióval, így nem szükséges újrahozni az objektum jogosultságait.

**AUTHID**

* Az **AUTHID** kulcsszó segítségével meghatározhatjuk, hogy az alprogram mely felhasználói jogosultságokat használja.
  + **DEFINER (alapértelmezett)**: Az alprogram a **létrehozó** felhasználó jogosultságaival fut. Ez azt jelenti, hogy akkor is végrehajtható, ha a hívó felhasználónak nincs jogosultsága a használt táblákhoz vagy objektumokhoz, feltéve, hogy a létrehozó felhasználónak megvannak ezek a jogosultságai.
  + **CURRENT\_USER**: Az alprogram az aktuális **hívó felhasználó** jogosultságait használja. Tehát az alprogram csak azokkal a jogosultságokkal fut, amikkel a hívó rendelkezik.

**DETERMINISTIC**

* A **DETERMINISTIC** kulcsszó egy optimalizálási előírást jelent, amely azt jelzi, hogy egy függvény mindig ugyanazzal a bemenettel ugyanazt az eredményt adja vissza. Ez lehetővé teszi az optimalizálónak, hogy ne hajtsa végre újra a függvényt ugyanazzal a bemenettel, hanem a korábbi eredményre hivatkozzon.
* Ha egy függvény **DETERMINISTIC** és ugyanazokkal a paraméterekkel hívják meg többször, akkor az első hívás eredményét cache-eli az adatbázis, így elkerülhető a redundáns számítás.
* **Optimalizálás**: Ez hasznos lehet a teljesítmény szempontjából, különösen akkor, ha a függvény sok adatot dolgoz fel vagy hosszú ideig fut.

# Triggerek (2.8)

* A triggerek **névvel ellátott adatbázis objektumok**, amelyek **eseményorientált feldolgozást** tesznek lehetővé.
* **Trigger definíció:**
  + Olyan tevékenységet ír le, amely **automatikusan végrehajtódik**, amikor egy tábla vagy nézet módosul, vagy más **felhasználói vagy rendszer esemény** történik.

**Triggerek osztályozása**

* **Kiváltó esemény alapján:**
  + **Rendszeresemény trigger:** például bejelentkezés, táblalétrehozás.
  + **Felhasználói esemény trigger:** például INSERT, UPDATE, DELETE.
* **Időzítés alapján:**
  + **BEFORE:** az esemény előtt hajtódik végre.
  + **AFTER:** az esemény után hajtódik végre.
  + **INSTEAD OF:** a művelet **helyett** hajtódik végre (általában nézetekhez használják).

**Triggerek használatának célja**

* **Automatizálás és adatbiztonság:**  
  Triggerek segítségével bizonyos műveletek automatikusan végrehajthatók, illetve megakadályozhatók.
* **Példák használati célokra:**
  + Oszlopbeli értékek automatikus generálása (pl. dátum, azonosító)
  + Események naplózása (audit)
  + Táblák elérésének statisztikai nyilvántartása
  + Nézeteken futó DML utasítás esetén a forrástáblák módosítása
  + Elosztott adatbázisok esetén hivatkozási integritás fenntartása
  + DML műveletek tiltása üzleti órákon kívül
  + Érvénytelen tranzakciók kiszűrése
  + Komplex üzleti/hivatkozási szabályok érvényesítése, amelyeket **nem lehet megszorításokkal** kifejezni

**DML triggerek**

* A **DML (Data Manipulation Language)** triggerek egy **táblához vagy nézethez** kapcsolódnak.
* **Kiváltó események:**
  + INSERT
  + UPDATE
  + DELETE

**Trigger típusai**

* **Egyszerű DML trigger:**
  + Csak **egy időzítési ponton** (BEFORE vagy AFTER) aktiválódik.
  + Kódja kizárólag az adott eseményre vonatkozik.
* **Összetett DML trigger:**
  + Több időzítési ponton is aktiválódhat.
  + Akkor előnyös, ha különböző időzítési pontokhoz tartozó kódok **közös adatokon osztoznak**.

**INSTEAD OF DML trigger**

* **Kizárólag nézetekhez** (vagy azok beágyazott táblatípusaihoz) hozható létre.
* A **DML utasítás helyett** hajtódik végre.
* **Sorszintű (FOR EACH ROW)** trigger.
* Akkor használjuk, ha a nézet közvetlenül nem módosítható DML utasítással.

**Egyszerű DML trigger szerkezete**

* **REFERENCING OLD AS régi, NEW AS új:**
  + Lehetővé teszi a régi (OLD) és új (NEW) értékek hivatkozását.
  + INSERT esetén csak NEW, DELETE esetén csak OLD, UPDATE esetén mindkettő elérhető.
* **FOR EACH ROW:**
  + A trigger minden egyes érintett sor esetén lefut.
  + Ezzel jelezzük, hogy **sorszintű trigger**.
* **WHEN (feltétel):**
  + A trigger csak akkor hajtódik végre, ha a megadott **feltétel teljesül**.
  + Csak **sorszintű triggerek** esetén használható.

**OLD és NEW pszeudorekordok**

* A trigger aktiválásakor a PL/SQL létrehoz két **pszeudorekordot**:
  + :OLD — a sor **régi** értékeit tartalmazza
  + :NEW — a sor **új** értékeit tartalmazza
* Tartalom a DML művelettől függően:
  + INSERT esetén: :OLD **nem tartalmaz** értéket, :NEW az **új értékeket** tartalmazza
  + UPDATE esetén: :OLD a **régi**, :NEW az **új értékeket** tartalmazza
  + DELETE esetén: :OLD a **régi értékeket** tartalmazza, :NEW **üres**
* A pszeudorekordok majdnem úgy viselkednek, mint egy normál rekord (típus, mezőhivatkozás stb.)

**Trigger szintek**

* **Sorszintű trigger** (FOR EACH ROW):
  + **Minden egyes érintett sorra** lefut
  + Ha nem változik sor, **nem fut le**
* **Utasítás szintű trigger**:
  + Egyszer fut le, **függetlenül a kezelt sorok számától**
  + Akkor is lefut, ha **egy sor sem** módosul
* **Időzítés alapján**:
  + BEFORE trigger – a módosítás előtt fut
  + AFTER trigger – a módosítás után fut
  + Mindkettő lehet **sor**- vagy **utasítás szintű**
* Egy tábla/nézet DML műveletéhez **több trigger is megadható** (ugyanarra az eseményre is)

**Trigger kiváltó események és feltételes predikátumok**

* Egy trigger akár **több eseményre is reagálhat** (INSERT, UPDATE, DELETE)
* A trigger törzsében a **feltételes predikátumokkal** meghatározhatjuk, mely esemény váltotta ki:
  + IF INSERTING THEN ...
  + IF UPDATING THEN ...
  + IF UPDATING('column\_name') THEN ...
  + IF DELETING THEN ...
* Ezek a predikátumok **BOOLEAN kifejezésként** használhatók bárhol a trigger törzsén belül