# Język PL/SQL

**język proceduralny serwera DB ORACLE**

# 1. Wiadomości ogólne

Język PL/SQL pełni rolę języka proceduralnego w środowisku bazy danych ORACLE DB – analogiczną do języka T-SQL w MS SQL Server. Po raz pierwszy pojawił się w ORACLE v7 i jest wbudowywany w kolejnych wersjach ORACLE DB i wraz z nimi nadal rozwijany. Składnia PL/SQL jest wzorowana na składni języka ADA (powstały w latach 70 ubiegłego wieku, do dzisiaj istnieją korzystające z niego aplikacje).

Składnia PL/SQL jest nieco bardziej skomplikowana niż składnia T-SQL, ale przede wszystkim jest znacznie bardziej uporządkowana i restrykcyjnie przestrzegana. Kod wykonywany jest bezpośrednio na serwerze.

Często spotykanym błędem są próby przenoszenia doświadczeń z użycia T-SQL do środowiska PL/SQL, co oczywiście kończy się niepowodzeniem i przed czym przestrzegam czytelnika.

Po stronie serwera bazy danych powinny być wykonywane wszystkie operacje związane ze składowaniem danych, centralną kontrolą spójności i bezpieczeństwa danych. Znaczna część tych zadań może zostać zrealizowana wprost przez polecenia języka SQL. PL/SQL pozwala rozszerzyć te zadania, a pojedyncze instrukcje łączyć w większe struktury, które mogą być przechowywane jako obiekty bazy danych:

- procedury składowane
- wyzwalacze
- funkcje
- pakiety

# 2. Programy narzędziowe

## a) SQL*Plus

Najstarszym klientem bazy ORACLE jest prosty program SQL*Plus. Uruchamiany z linii poleceń, działa z każdą instalacją ORACLE na każdej platformie systemowej. Jest też domyślnie instalowany z każdą instalacją ORACLE. Umożliwia on nawiązanie połączenia z bazą danych ORACLE, wydawanie poleceń w językach SQL i Pl/SQL oraz wykonywanie skryptów z poleceniami w tych językach. SQL*Plus jest interakcyjnym systemem umożliwiającym wprowadzanie i wykonywanie poleceń SQL, skryptów złożonych z poleceń języka SQL i Pl/SQL, uruchamianie bloków i procedur Pl/SQL. Stanowi niezależne od serwera bazy danych środowisko, operuje stosunkowo niewielką liczbą własnych instrukcji, umożliwia także operowanie zmiennymi dwóch rodzajów: wiązania i podstawienia, których można używać w instrukcjach SQL i Pl/SQL do wzajemnej komunikacji wartości. Jakkolwiek niezbyt wygodny w użyciu i dość archaiczny z dzisiejszego punktu widzenia, z uwagi na prostotę, a przede wszystkim możliwość uruchomienia w każdym środowisku, nadal jest używany przez programistów i administratorów baz ORACLE

### b) Sqldeveloper

Aplikacją narzędziową o funkcjonalności zbliżonej do Management Studio jest wspomniany wcześniej „okienkowy" Sqldeveloper. Jest to wygodne narzędzie, dające użytkownikowi wiele możliwości w operowaniu na bazie danych – począwszy od wykonywania zapytań SQL, po działania administratorskie. Sqldeveloper implementuje część instrukcji pochodzących z SQL*Plus. Znaczna część instrukcji SQL*Plus stała się zbędna w środowisku „okienkowym" i nie jest w nim realizowana. W dalszej części wykładu, jak też w trakcie ćwiczeń, Sqldeveloper będzie traktowany jako podstawowe środowisko pracy. Instalacja dostępna jest na stronach ORACLE, W PJATK można go pobrać z zasobu \\pjwstk.edu.pl\apps\ folder Program Files\Sqldeveloper4.

### c) Inne aplikacje klienckie

Zamiast SQL*Plus oraz Sqldeveloper w roli aplikacji klienckiej serwera ORACLE można użyć narzędzi Dbeaver lub DataGrip. Oba te rozwiązania są uniwersalne, czyli mogą łączyć się z innymi niż ORACLE serwerami baz danych (również z bazami NoSQL). Zwłaszcza Dbeaver jest intuicyjny, łatwy w użyciu. Oba te narzędzia są dostępne w zasobach PJATK.

### d) Instrukcje SQL*Plus dostępne w Sqldeveloper

Część instrukcji dostępnych w SQL*Plus może zostać użyta w Sqldeveloper. Poniżej kilka ważniejszych:

SET AUTO[COMMIT ON | OFF – włącza/wyłącza automatyczne zatwierdzanie każdej poprawnie wykonanej instrukcji SQL,

SET SERVEROUTPUT ON | OFF – włącza/wyłącza wyświetlanie wyników działania bloków i [1]procedur,

SET ECHO {OFF | ON} – wyłącza/włącza wypisanie instrukcji przed jej wykonaniem ¬ SET FEEDBACK – wyłącza/włącza komunikat o liczbie zwróconych rekordów

SET VERIFY OFF | ON – wyłącza / włącza wyświetlanie instrukcji przed i po podstawieniu zmiennej z klawiatury

EXEC[UTE] *nazwa_procedury* (*parametry*…) – uruchamia procedurę,

VARIABLE X *typ_danych* – deklaracja zmiennej wiązania X,

EXECUTE :X := *wyrażenie* – podstawienie wartości na zmienną wiązania

ACCEPT '*nazwa_zmiennej*' PROMPT '*tekst_komunikatu'* – deklaracja i pobranie od użytkownika wartości zmiennej podstawienia

### e) Polecenie wypisania komunikatu

Polecenie zapisania komunikatu do bufora pamięci (dostępnego dla innych procesów) jest[2]:

---

[1] Użycie tego polecenia jest konieczne w SQL*Plus oraz Sqldeveloper. W DataGrip jest zastępowane przez check box, w Dbeaver jest zbędne (nie może zostać użyte). W zamieszczonych dalej przykładach będzie jednak umieszczane.

[2] Dokładnie jest to procedura (metoda) pakietu dbms_output oferującego znacznie szersze spektrum operacji, które nie będą tu omawiane. Zainteresowanych odsyłam do dokumentacji serwera.

```
Dbms_output.put_line(tekst lub zmienna lokalna);
```

Treść komunikat pojawi się na konsoli, jeżeli wcześniej zostanie uruchomione polecenie `SET SERVEROUTPUT ON`.

## 3. Nazwy w ORACLE DB

Reguły nadawania nazw obiektów bazy danych, dotyczące także nazw także zmiennych

- Muszą zaczynać się od litery alfabetu łacińskiego,
- mogą zawierać litery, cyfry oraz znaki $, #, _
- nie mogą być słowami kluczowymi SQL ani Pl/SQL,
- powinny być unikalne w obrębie bazy danych,
- maksymalna liczba znaków w nazwie nie może przekraczać 30 bajtów.

Nazwami w ORACLE mogą też być ciągi dowolnych znaków ujęte w podwójne cudzysłowy, np. ” x+y” , ”last minute” , ”on/off switch”. Użycie tego typu nazw powoduje, że nazwy zmiennych stają się „case sensitive” i muszą być przywoływane dokładnie tak, jak zostały zadeklarowane.

Nieco więcej informacji na temat nazw obiektów i zmiennych w db ORACLE znajduje się w części VI.

## 4. Komentarze

Część kodu nie przeznaczona do wykonywania (wyłączona), nazywana jest komentarzem. W trakcie wykonywania programu będzie ona ignorowana przez interpreter. W obu środowiskach (ORACLE i MS SQL) komentarz oznaczany jest jednakowo.

Komentarz blokowy, dowolna liczba linii, pomiędzy nawiasami:

```
/* te wiersze są zakomentowane
i nie będą brane pod uwagę w trakcie
wykonywania programu*/
```

Komentarz jednoliniowy – do końca linii:

```
-- a to są zakomentowane dwa  pojedyncze wiersze,
-- one też zostaną pominięte
SELECT * FROM emp; -- polecenie odczytania danych
```

Skrótem klawiszowym komentowania / odkomentowania jest CTRl + „/”

## 5. Blok anonimowy

Podstawową konstrukcją programistyczną w języku Pl/SQL jest blok - struktura w języku ORACLE Pl/SQL posiadająca sformalizowaną składnię. Poza blokiem mogą być uruchamiane tylko polecenia SQL. Deklarowanie zmiennych i uruchamianie instrukcji sterujących musi być wykonywane w obrębie bloku.

Pl/SQL restrykcyjnie przestrzega konieczności zakończenia każdej instrukcji średnikiem, co w porównaniu z T-SQL wprowadza porządek w kodzie.

Bloki Pl/SQL mogą być zagnieżdżane, z czym wiąże się zasięg dostępności zmiennych deklarowanych w poszczególnych blokach. Blok zagnieżdżony jest traktowany przez blok zewnętrzny jako pojedyncza instrukcja.

Składnia bloku anonimowego jest następująca:

```
DECLARE
   Deklaracje obiektów Pl/SQL (zmienne, stałe, wyjątki, procedury,
   funkcje)
BEGIN
   Ciąg instrukcji SQL i Pl/SQL
EXCEPTION
   Obsługa wyjątków
END;
```

Sekcje **DECLARE** i **EXCEPTION** są opcjonalne, bloki mogą być zagnieżdżane, w bloku mogą się pojawić instrukcje **SELECT**, **INSERT**, **UPDATE**, **DELETE**, **COMMIT**, **ROLLBACK**.

## 6. Zmienne

### a) Typy zmiennych

W Pl/SQL są dostępne typy danych z dialeku ORACLE języka SQL a ponadto m.in.

- **BOOLEAN** – wartości logiczne
- **BINARY_INTEGER** (albo **PLS_Integer**) typ liczb całkowitych ze znakiem, zapisywany na 32 bitach – niezależny od podtypów typu **NUMBER** i przez to wymagający przy zapisie mniej miejsca w pamięci RAM, korzystający z arytmetyki realizowanej przez hardware.

W języku Pl/SQL użycie każdej zmiennej musi zostać poprzedzone jej deklaracją w sekcji DECLARE bloku. Nazwy zmiennych i stałych podlegają ogólnym regułom nazewnictwa w ORACLE

### b) Deklaracja zmiennych i stałych

Składnia deklaracji zmiennej w bloku DECLARE:

```
nazwa_zmiennej [CONSTANT] typ_danych [NOT NULL] [:= |
DEFAULT wartość_początkowa]
```

Nie można łączyć deklaracji kilku zmiennych jednego typu, a każda deklaracja musi być zakończona średnikiem. Zmiennej w trakcie deklaracji można nadać wartość początkową pod postacią stałej, wyrażenia lub nazwy funkcji.

### c) Podstawienie wartości na zmienną

Aby na zmienną można był podstawić wartość, musi ona zostać wcześniej zadeklarowana. Wartość początkowa może zostać nadana zmiennej łącznie z jej deklaracją.

Operatorem podstawienia w Pl/SQL jest :=

```
nazwa_zmiennej := wyrażenie;
```

Drugim sposobem podstawienia jest skorzystanie z instrukcji SELECT

```
SELECT wyrażenie, wyrażenie (,…)
INTO   zmienna, zmienna (,…)
FROM   źródła_danych….)
[WHERE …] [GROUP BY…] [HAVING…] (…);
```

Wyrażenia mogą zawierać nazwy kolumn.

Powyższa instrukcja **SELECT** musi zwracać dokładnie jeden rekord, w przeciwnym wypadku serwer podniesie błąd. Wynika z tego wniosek, iż strategia sprawdzenia istnienia rekordu:

- Zadeklaruj zmienną,
- Podstaw na zmienną wartość odczytaną instrukcją **SELECT**,
- Sprawdź, czy zmienna ma wartość inną niż NULL

w przypadku PL/SQL nie sprawdzi się w przypadku braku poszukiwanego rekordu.

W celu zachowania zgodności składni z opisem standardu języka SQL (obligatoryjność klauzuli **FROM**) ORACLE implementuje dwie pomocnicze tabele, zwracające pojedyncze wartości, które mogą być wykorzystywane w instrukcjach powyższego typu:

**DUMMY** zwraca 0,

**DUAL** zwraca 'X'

*Przykład IV.7.1*

```
Set Serveroutput on;

DECLARE
   v_nazwisko Varchar2(30);
   v_placa Int := 1200;
   v_date DATE;
   v_info VARCHAR2(100);
BEGIN
SELECT Sysdate INTO v_date FROM dual;
   v_nazwisko := 'Kowalski';
   v_info := 'Pracownik ' || ' zarabia ' || v_placa || ' od dnia ' || v_date;
   dbms_output.put_line(v_info);
END;
```

Wszystkie niezainicjowane zmienne (zmienne, którym nie nadano wartości początkowej) są NULL. Zmienne, których wartości nie powinny ulegać zmianom to stałe. Podlegają one tym samym regułom nazewnictwa i wykorzystania co zmienne, ale musi im zostać nadana wartość podczas deklaracji.

Stałe i zmienne mogą być używane wewnątrz poleceń SQL. Zmienna może być zadeklarowana jako NOT NULL, ale wówczas trzeba w trakcie deklaracji nadać jej wartość początkową.

*Przykład V.7.2*

```
DECLARE
   v_ile INTEGER; --zmienna nie została zainicjowana
   v_max CONSTANT INTEGER := 10; --stała musi zostać zainicjowana
BEGIN
   v_ile := v_ile + 1; --zmienna nadal ma wartość NULL
   dbms_output.put_line(v_ile); --czyli nic nie zostanie wypisane!
END;
```

**UWAGA !!!:**

W **PL/SQL NIE MOŻNA** użyć konstrukcji znanych z T-SQL

*DECLARE Nazwa_zmiennej Typ_danych* := (*SELECT ….)*

ani też

*Nazwa_zmiennej* := (*SELECT ….)*

### d) Definiowanie typu zmiennej przez odwołanie do typu istniejącego obiektu.

Deklaracja typu zmiennej może się odwoływać do typu obiektu, który został zdefiniowany wcześniej, bez konieczności sprawdzenia tego typu. Może to być typ innej, wcześniej zadeklarowanej zmiennej, albo typ kolumny w tabeli.

Jeżeli typ „wzorca" ulegnie zmianie, również ulegnie zmianie typ zmiennej lub stałej odwołującej się do tego „wzorca".

Składnia jest zaprezentowana w poniższym przykładzie.

*Przykład V.7.3*

```
DECLARE
   v_ile INTEGER; -- typ zmiennej zdefiniowany „klasycznie"
   v_max v_ile%type :=10; -- typ zgodny z typem wcześniej zadeklarowanej zmiennej
   v_sal CONSTANT emp.sal%type := 1100; -- typ stałej zgodny z typem kolumny sal
                                        -- w tabeli emp
BEGIN
   v_ile := v_max + 1;
   dbms_output.put_line(v_ile);
END;
```

### e) Definiowanie zmiennych wierszowych (rekordowych)

Wzorcem dla deklarowanej zmiennej może być cały wiersz tabeli, albo instrukcja **SELECT** definiująca kursor. Na zmienną podstawiana jest wówczas cała zawartość rekordu, a odczyt wartości z poszczególnych pól odbywa się poprzez odwołanie:

*nazwa_zmiennej.nazwa_kolumny*

Zmiennej wierszowej nie można użyć bezpośrednio po słowie kluczowym **VALUES** w instrukcji **INSERT INTO**.

*Przykład V.7.4*

```
DECLARE
   v_test emp%rowtype; v_ename emp.ename%type;
   v_sal emp.sal%type;
BEGIN
SELECT * INTO v_test FROM emp WHERE empno = 7788;
   v_ename := v_test.ename;
   v_sal := v_test.sal;
   dbms_output.put_line(v_ename || ' zarabia ' || v_sal);
   -- albo wprost
   dbms_output.put_line(v_test.ename || ' zarabia ' || v_test.sal);
END;
```

Jeżeli wzorcem zmiennej wierszowej jest instrukcja **SELECT** definiująca kursor, przy definiowaniu zmiennej wierszowej odwołujemy się do nazwy kursora:

*nazwa_zmiennej nazwa_kursora%rowtype*

### f) Zmienne wierszowe definiowane przez użytkownika

Typy zmiennych wierszowych mogą być definiowane przez użytkownika:

**TYPE** *Nazwa_Typu_rekordowego*
**IS RECORD** ( *nazwa_zmiennej Typ_danych,(….)* );

*Przykład V.7.5*

```
DECLARE
TYPE typ_dane IS RECORD (
                         naz Varchar2(20),
                         plac Number);
   dane typ_dane;
   v_ename dane.naz%Type;
   v_sal dane.plac%Type;
BEGIN
   SELECT ename, sal INTO dane FROM emp WHERE empno = 7788;
   v_ename := dane.naz;
   v_sal := dane.plac;
dbms_output.put_line(v_ename || ' zarabia ' || v_sal);
END;
```

### g) Zasięg działania zmiennych

Zasięg zmiennej (lub stałej) w Pl/SQL to obszar bloków, z których można się do tej zmiennej odwołać. Ponieważ bloki mogą być zagnieżdżane, zatem zasięg zmiennych jest zależny od tego, w jakim bloku została zadeklarowana.

Zmienna zadeklarowana w bloku jest dostępna w blokach w nim zagnieżdżonych (wewnętrznych względem bloku deklaracji). Zmienna zadeklarowana w bloku nie jest dostępna w bloku zewnętrznym w stosunku do bloku jej deklaracji, czyli bloku, w którym blok jej deklaracji jest zagnieżdżony.

Zmienne zadeklarowane w bloku nie są dostępne w blokach „sąsiednich", utworzonych na tym samym poziomie. Odwołanie do zmiennej znajdującej się poza zasięgiem wywołuje błąd. Zmienne w blokach zagnieżdżonych mogą mieć te same nazwy, ale są traktowane jako różne zmienne!

*Przykład V.7.6*

```
END;
DECLARE
   v_tytul Varchar2(20);
BEGIN
   v_tytul := 'ORACLE jest latwy';
      DECLARE
        v_tytul2 Varchar2(20);
        v_tytul Varchar2(20);
      BEGIN
        v_tytul2 := 'Pl/SQL jest latwy';
        v_tytul := 'Pl/SQL jest TRUDNY';
      END;
   dbms_output.put_line(v_tytul);
   --dbms_output.put_line(v_tytul2); zmienna poza zasięgiem wygeneruje błąd
END;
```

*Przykład V.7.7*

*Zmienne o tej samej nazwie, zadeklarowane w zagnieżdżonych blokach, to dwie różne zmienne;*

```
DECLARE
   v_tytul Varchar2(20);
BEGIN
   v_tytul := 'ORACLE jest latwy';
```

```
    DECLARE
      v_tytul Varchar2(20);
    BEGIN
      v_tytul := 'Pl/SQL jest TRUDNY';
      dbms_output.put_line(v_tytul);
    END;
  dbms_output.put_line(v_tytul);
END;
```

*Przykład V.7.8*

*Do zmiennej zadeklarowanej w bloku zewnętrznym możemy odwołać się zarówno z bloku zewnętrznego jak i wewnętrznego:*

```
DECLARE
  v_tytul Varchar2(20);
BEGIN
  v_tytul := 'ORACLE jest latwy';
  BEGIN
    v_tytul := 'Pl/SQL jest TRUDNY';
    dbms_output.put_line(v_tytul);
  END;
  dbms_output.put_line(v_tytul);
END;
```

### h) Zmienne podstawienia i zmienne wiązania

Oprócz zmiennych deklarowanych w bloku Pl/SQL, mogą jeszcze występować zmienne z aplikacji korzystających z bloku Pl/SQL. Nazywane są **zmiennymi wiązania**, a w odwołaniach ich nazwa poprzedzana jest dwukropkiem. Deklarowane są poleceniem SQL*Plus VARIABLE, wypisywane poleceniem PRINT.

Drugim rodzajem zmiennych są **zmienne podstawienia** SQL*Plus, które mogą występować wyłącznie w wyrażeniach po prawej stronie instrukcji przypisania. W odwołaniach ich nazwa poprzedzana jest znakiem &.

Oba te rodzaje zmiennych mogą służyć do wprowadzania wartości z klawiatury i przekazania ich do wykonania w bloku Pl/SQL. Dalsze dwa przykłady pokazują użycie zmiennych wiązania i zmiennych podstawienia do wykonania operacji w bloku Pl/SQL.

**UWAGA:** zmienne podstawienia i zmienne wiązania są rozwiązaniem pochodzącym z SQL*Plus. **Sqldeveloper** prawidłowo implementuje ich użycie, natomiast **DataGrip** ani **Dbeaver** nie dopuszczają ich użycia, podobnie jak innych poleceń pochodzących z SQL*Plus.

*Przykład V.7.9*

*Użycie zmiennej podstawienia.*

```
ACCEPT year_sal PROMPT 'Podaj roczne zarobki'

DECLARE
  sal NUMBER(8,2) := &year_sal;
BEGIN sal := sal/12;
  dbms_output.put_line(sal);
END;
```

W wyniku uruchomienia skryptu wyświetlane jest okno dialogowe (Input Box) pozwalające wprowadzić wartość zmiennej podstawienia, która po wprowadzeniu użyta zostaje w bloku.

*Przykład V.7.10*

*Użycie zmiennej wiązania.*

```
ACCEPT year_sal PROMPT 'Podaj roczne zarobki'
VARIABLE b_sal NUMBER;
BEGIN
:b_sal := &year_sal /12;
END;
/
PRINT b_sal
```

Ten skrypt działa podobnie do poprzedniego, ale w tym rozwiązaniu dane przekazywane są do bloku poprzez zmienną wiązania, jej wartość ulega zmianie wewnątrz bloku i jest odczytywana poza blokiem.

Znak „/" po końcu bloku jest poleceniem jego uruchomienia i kontynuacji działania skrypty (pełni rolę zbliżoną do GO w środowisku MS SQL Server'a).

### i) Wybrane zmienne systemowe

- **SQL%Rowcount** – zwraca liczbę wierszy przetworzonych przez ostatnią instrukcję SQL,
- **SQL%Found = TRUE** jeśli został znaleziony rekord,
- **SQL%Notfound = TRUE** jeśli nie został znaleziony żaden rekord
- **SQLErrm –** tekst komunikatu o błędzie
- **SQLCode –** numer błędu

Dwie ostatnie zmienne mogą być użyte wyłącznie w sekcji EXCEPTION.

## 7. Instrukcje warunkowe

```
IF warunek THEN
     Ciąg_instrukcji_1;
END IF;
```

Instrukcje po THEN są wykonywane wtedy, gdy wartością warunku jest TRUE.

```
IF warunek THEN
     Ciąg_instrukcji_1;
ELSE
     Ciąg_instrukcji_2;
END IF;
```

Instrukcje po **ELSE** są wykonywane wtedy, gdy wartością warunku jest **FALSE** lub **NULL**.

```
IF warunek_1 THEN
     Ciąg_instrukcji_1;
ELSIF_2 warunek_2 THEN
     Ciąg_instrukcji_2;
END IF;
```

Jeżeli warunek_1 jest **TRUE** wykonywany jest Ciąg_instrukcji_1, jeżeli jest **FALSE** lub **NULL** sprawdzany jest warunek_2 i.t.d.

**UWAGA !!!:**

W PL/SQL **NIE MOŻNA** użyć konstrukcji znanych z T-SQL

```
IF EXISTS (SELECT …)
```

Ani też

```
IF nazwa_zmiennej = (SELECT …) THEN …END IF;
```

## 8. Instrukcje iteracji (pętli)

Jakkolwiek PL/SQL implementuje kilka rozwiązań umożliwiających wykonanie powtarzającej się sekwencji poleceń (iteracji, pętli), to w rzeczywistości są to różne warianty użycia jedenej instrukcji z różnymi rozwiązaniami sterowania tą instrukcją.

### a) Składnia podstawowa

```
LOOP
  ciąg_instrukcji
END LOOP;
```

Instrukcje wewnątrz pętli wykonywane są w każdej iteracji. Wyjście z pętli następuje po wywołaniu instrukcji **EXIT** albo **EXIT WHEN** warunek albo podniesieniu błędu.

*Przykład V.8.1*

```
DECLARE
x NUMBER := 0;
BEGIN
   LOOP
      DBMS_OUTPUT.PUT_LINE('Wewnątrz: x = ' || TO_CHAR(x));
      x := x + 1;
      IF x > 3 THEN
         EXIT;
      END IF;
   END LOOP;
DBMS_OUTPUT.PUT_LINE('Po wyjściu: x = ' || TO_CHAR(x));
END;
```

### b) Składnia pętli FOR

```
FOR i IN [REVERSE] wartość1..wartość2
LOOP
   ciąg_instrukcji
END LOOP;
```

Instrukcje wewnątrz pętli wykonywane są dla wartości zmiennej kontrolnej `i` w przedziale od `wartość1` do `wartość2` powiększanej o 1 w każdym cyklu pętli. Użycie opcji **REVERSE** zmienia kierunek inkrementacji – wykonywana jest od `wartość2` do `wartość1` z pomniejszaniem zmiennej kontrolnej `i` o 1 w każdym cyklu pętli.

*Przykład V.8.2*

```
BEGIN
DBMS_OUTPUT.PUT_LINE ('w_dolna < w_górna');
FOR k IN 1..3 LOOP
   DBMS_OUTPUT.PUT_LINE (k);
END LOOP;
DBMS_OUTPUT.PUT_LINE ('w_dolna = w_górna');
FOR i IN 2..2 LOOP
   DBMS_OUTPUT.PUT_LINE (i);
END LOOP;
```

```
END;
```

*Przykład V.8.3*

```
BEGIN
DBMS_OUTPUT.PUT_LINE ('w_dolna < w_górna');
FOR k IN 3..1 LOOP
   DBMS_OUTPUT.PUT_LINE (k);
END LOOP;
DBMS_OUTPUT.PUT_LINE ('w_dolna < w_górna');
FOR i IN REVERSE 1..2 LOOP
   DBMS_OUTPUT.PUT_LINE (i);
END LOOP;
END;
```

### c) Skkładnia pętli WHILE

```
WHILE warunek
LOOP
   ciąg_instrukcji
END LOOP;
```

Iteracja wykonywana jest tak długo, jak długo `warunek` ma wartość **TRUE**.

*Przykład V.8.4*

```
DECLARE
   done BOOLEAN := FALSE;
BEGIN
WHILE done LOOP
   DBMS_OUTPUT.PUT_LINE ('Ta linia nie zostanie wypisana.');
   done := TRUE; -- To podstawienie nie zostanie wykonane.
END LOOP;
WHILE NOT done LOOP
   DBMS_OUTPUT.PUT_LINE ('Hello, world!');
   done := TRUE;
END LOOP;
END;
```

## 9. Kursory

Zasada działania i użycia kursorów Pl/SQl jest taka sama jak T-SQl (patrz rozdział III.10). Różnice są wyłącznie składniowe. Ponieważ Pl/SQL nie przewiduje możliwości wyprowadzenia danych z bloku (procedury) w postaci Result Set, zatem znacznie częściej zachodzi konieczność użycia kursora.

### a) Składnia kursora w PL/SQL

Zdefiniowanie kursora:

```
CURSOR nazwa_kursora IS instrukcja_SELECT;
```

Otwarcie kursora:

```
OPEN nazwa_kursora
```

Podstawienie kolejnych pobranych wierszy na wcześniej zadeklarowane zmienne:

```
FETCH nazwa_kursora Into zmienna1, zmienna2 (…);
```

Wyjście z pętli po odczytaniu wszystkich wierszy:

```
EXIT WHEN nazwa_kursora%NOTFOUND;
```

Zamknięcie kursora (zwolnienie blokad):

```
CLOSE nazwa_kursora;
```

### b) Zmiene stanu kursora

`nazwa_kursora%FOUND` = **TRUE** jeśli z bazy sprowadzono kolejny rekord,

`nazwa_kursora%NOTFOUND` = **TRUE** jeśli kolejny rekord nie został znaleziony – koniec sprowadzania rekordów,

`nazwa_kursora%ROWCOUNT` zwraca liczbę sprowadzonych dotąd rekordów,

`nazwa_kursora%ISOPEN` = **TRUE** jeżeli kursor jest otwarty

```
IF NOT nazwa_kursora%ISOPEN THEN
    OPEN nazwa_kursora;
END IF;
LOOP FETCH nazwa_kursora INTO …
```

*Przykład V.9.1*

```
SET SERVEROUTPUT ON
DECLARE
   v_dname Varchar2(10);
   v_deptno INT;
   v_dsal Number(8,2) := 0;
   CURSOR Cur_Dept IS
   SELECT deptno, dname FROM dept;
BEGIN
OPEN Cur_Dept;
LOOP
   FETCH Cur_Dept INTO v_deptno, v_dname;
   EXIT WHEN Cur_Dept%NOTFOUND;
   SELECT NVL(SUM(sal), 0) INTO v_dsal
   FROM emp WHERE deptno = v_deptno;
   dbms_output.put_line('Budżet dzialu ' || v_dname || ' wynosi ' || v_dsal);
   v_dsal := 0;
END LOOP;
CLOSE Cur_Dept;
END;
```

Powyższy trywialny przykład ma za zadanie jedynie prezentację składni kursora. Identyczny wynik można uzyskać prostym poleceniem SELECT.

### c) Aliasy kolumn kursora

W deklaracji kursora, na liście SELECT mogą znajdować się dowolne wyrażenie, przy czym jeśli nie są to nazwy kolumn, muszą być stosowane aliasy (etykiety) wyrażeń.

*Przykład V.9.2*

```
SET SERVEROUTPUT ON;

DECLARE
   CURSOR Budzety_dzialow IS
   SELECT d.Deptno, Dname, Sum(sal) AS Budzet_dzialu --Tutaj musi zostać użyty
   alias
   FROM emp e
   INNER JOIN Dept d
   ON e.deptno = e.deptno
   GROUP BY d.Deptno, Dname;
   v_deptno Int;
   v_dname Varchar2(10);
   v_budzet Number(10,2);
   v_info Varchar2(100);
BEGIN
OPEN Budzety_dzialow;
LOOP
   FETCH Budzety_dzialow INTO v_deptno, v_dname, v_budzet;
   EXIT WHEN Budzety_dzialow%NotFound;
   v_info := 'Budzet dzialu ' || v_dname || ' wynosi ' || v_budzet;
   dbms_output.put_line(v_info);
END LOOP;
CLOSE Budzety_dzialow;
END;
```

### d) Iteracja z kursorem

W instrukcji iteracji (pętli) możeny odwołać się do kursora działającego niejako „w tle". W takim przypadku instrukcje OPEN, FETCH, CLOSE wykonywane są niejawnie, a kursor zakończy się i zostanie zamknięty po podstawieniu na zmienne wszystkich odczytanych rekordów.

*Przykład V.9.3*

```
SET serveroutput on

DECLARE
   CURSOR kursor IS SELECT dname FROM dept;
BEGIN
   FOR kursorek IN kursor
   LOOP
      dbms_output.put_line(kursorek.dname);
   END LOOP;
END;
```

W powyższym przykładzie „`kursorek`" to dowolna nazwa, niejako alias kursora.

### e) Kursor z parametrami

W instrukcji SELECT definiującej kursor mogą występować parametry. Ten sam kursor może być otwarty wielokrotnie – z różnymi wartościami parametrów.

```
CURSOR nazwa_kursora(parametr typ_danych, ....)
IS instrukcja_SELECT … ;
```

*Przykład V.9.4*

```
SET serveroutput on
```

```
DECLARE
   CURSOR Dept_job (v_deptno Int, v_job Varchar2)
   IS SELECT Empno, Ename, sal
   FROM emp WHERE deptno = v_deptno AND job = v_job;
   v_ees Dept_job%rowtype;
BEGIN
   OPEN Dept_job(10, 'CLERK');
   LOOP
      FETCH Dept_job INTO v_ees;
      EXIT WHEN Dept_job%NOTFOUND;
      dbms_output.put_line(v_ees.empno || ' ' || v_ees.ename || ' ' || v_ees.sal);
   END LOOP;
   CLOSE Dept_job;
END;
```

### f) Aktualizacja rekordów za pomocą kursora

Przy wykonywaniu instrukcji **SELECT** definiującej kursor mogą być zakładane blokady na wiersze, w celu ich modyfikacji.

Dyrektywa …**FOR UPDATE** [**OF** *kolumna, kolumna*…] określa tabele, lub kolumny tabel, w których rekordy mają zostać zablokowane w celu umożliwienia modyfikacji. Stowarzyszona z nią w instrukcji **UPDATE** lub **DELETE** klauzula …**WHERE CURRENT OF** nazwa_kursora umożliwia modyfikację lub usunięcie wiersza aktualnie sprowadzonego przez kursor z danej tabeli.

*Przykład V.9.5*

```
SET serveroutput on

DECLARE
   CURSOR ename_sal IS SELECT ename, sal FROM emp FOR UPDATE OF sal;
   v_rekosoba ename_sal%ROWTYPE;
   v_newsal NUMBER(8,2);
   v_info Varchar2(100);
BEGIN
OPEN ename_sal;
LOOP
   FETCH ename_sal INTO v_rekosoba;
   EXIT WHEN ename_sal%NOTFOUND;
   IF v_rekosoba.sal < 1000 THEN
      v_newsal := v_rekosoba.sal*1.1;
      UPDATE emp SET sal = v_rekosoba.sal * 1.1
      WHERE CURRENT OF ename_sal;
      V_info := 'Podniesiono place: ' || ' ' || v_rekosoba.ename || ' do ' ||
      v_newsal;
      dbms_output.put_line(info);
   END IF;
END LOOP;
CLOSE ename_sal;
COMMIT;
END;
```

# 10. Obsługa błędów

## a) Ogólne zasady obsługi błędów

Jeżeli blok, w którym wystąpił błąd, zawiera obsługę tego błędu, to po dokonaniu obsługi, sterowanie jest w zwykły sposób przekazywane do bloku go zawierającego (nadrzędnego). Jeśli nie zawiera obsługi błędów, następuje przekazanie błędu do bloku w którym dany blok jest zagnieżdżony (czyli do bloku nadrzędnego) i albo tam nastąpi jego obsługa, albo błąd przechodzi do środowiska zewnętrznego.

Błąd, który wystąpił w sekcji wykonawczej bloku (między BEGIN i END) jest obsługiwany w sekcji EXCEPTION tego samego bloku. Błędy, które wystąpią w sekcji deklaracji lub w sekcji wyjątków są przekazywane do bloku nadrzędnego.

Dobra praktyka programistyczna wymaga, aby każdy błąd został obsłużony – w ostateczności w klauzuli WHEN OTHERS THEN najbardziej zewnętrznego bloku. Aby móc stwierdzić, która instrukcja SQL spowodowała błąd:

- można używać podbloków z własną obsługą błędów, albo
- można używać licznika, zwiększającego się o jeden po wykonaniu każdej instrukcji SQL.

## b) Niektóre nazwane wyjątki

- **dup_val_on_index** - powtarzająca się wartość w indeksie jednoznacznym (UNIQUE),
- **no_data_**found – instrukcja SELECT nie zwróciła wartości przeznaczonych do podstawienia na zmienne w klauzuli SELECT … INTO…
- **too_many_rows** – instrukcja SELECT zwróciła więcej niż jeden wiersz zawierający wartości przeznaczone do podstawienia na zmienne w klauzuli SELECT … INTO…
- **zero_**divide – błąd dzielenia przez zero,
- **timeout_on_resource** – przekroczony limit czasu oczekiwania na zwrot żądanych zasobów,
- **invalid_cursor** – próba wykonania niepoprawnej operacji na kursorze,
- **invalid_number** – błąd (próby) konwersji na liczbę,
- **cursor_already _open** – próba otwarcia kursora, który jest już otwarty

## c) Składnia obsługi wyjątków w bloku

```
DECLARE
    Deklaracje zmiennych I kursorów
BEGIN
  Ciąg instrukcji PL/SQL i SQL
EXCEPTION
```

```
WHEN nazwaWyjątku_1 THEN
     Ciąg_instrukcji_1
WHEN nazwaWyjątku_2 THEN
     Ciąg_instrukcji_2
     (…)
WHEN OTHERS THEN
     Ciąg_instrukcji
END;
```

*Przykład V.10.1*

```
SET serveroutput on
DECLARE
   v_ename emp.ename%TYPE;
   v_info Varchar2(100);
BEGIN
SELECT ename INTO v_ename FROM emp WHERE sal = 1100;
EXCEPTION
WHEN no_data_found THEN
   dbms_output.put_line('Brak pracowników o pensji 1100');
WHEN too_many_rows THEN
   dbms_output.put_line('Jest kilku pracowników o pensji 1100');
WHEN OTHERS THEN
   V_info := 'Błąd nr ' || SQLCODE || ', komunikat: ' || Substr(SQLERRM, 1, 50);
   dbms_output.put_line(v_info);
END;
```

### d) Deklaracja wyjątków przez programistę

Poza wyjątkami definiowanymi przez SZBD, wyjątki mogą być deklarowane i używane przez tworzącego kod dewelopera. W tym celu w sekcji DECLARE wprowadzane zostają obok nazw zmiennych i kursorów nazwy wyjątków.

```
DECLARE
Nazwa_wyjątku EXCEPTION;
```

W sekcji wykonania instrukcji tak zadeklarowane wyjątki mogą być podnoszene instrukcją

```
RAISE Nazwa_wyjątku;
```

a następnie obsługiwane w sekcji **EXCEPTION**

```
WHEN nazwa_wyjątku THEN …
```

*Przykład V.10.2*

```
SET serveroutput on
DECLARE
   v_budzet emp.sal%TYPE;
   v_info Varchar2(100);
   v_notenugh EXCEPTION;
```

```
BEGIN
   SELECT Sum(sal+100) INTO v_budzet FROM emp WHERE deptno = 30;
   IF v_budzet <= 11000 THEN
      UPDATE emp SET sal = sal + 100 WHERE deptno = 30;
      V_info := 'Dokonano podwyzki w dziale 30';
      dbms_output.put_line(v_info);
   ELSE
      RAISE v_notenugh;
   END IF;
   EXCEPTION
   WHEN v_notenugh THEN
      V_info := 'Zbyt maly budzet na podwyżki w dziale 30';
      dbms_output.put_line(v_info);
   RAISE wniosek; --błąd obsłużony w bloku nadrzędnym
END;
```

### e) Procedura podnosząca błąd

Istnieje także możliwość podniesienia wyjątku za pomocą procedury

**Raise_application_error** (*numer_błędu, komunikat*)

z przypisaniem mu numeru oraz treści komunikatu. Wyjątek taki może zostać obsłużony w tym samym bloku, albo w aplikacji zewnętrznej, w której to wywołanie zostało wykonane. jest Programiści mogą używać numerów błędów w zakresie od -20000 do -209999.

```
BEGIN
   (.....)
   Raise_Application_Error(-20100,'Błąd');
EXCEPTION
WHEN OTHERS THEN numer := SQLCODE;
IF numer = -20100 THEN
   Dbms_output.Put_line('Błąd przechwycony!');
END IF;
END;
```

## 11. Podsumowanie części V

W części IV zostały przewdstawione podstawowe elementy składni języka PL/SQL – języka proceduralnego SZBD ORACLE. Omówione zostały:

- Deklaracja zmiennych
- Podstawienie wartości na zmienne, także wartości odczytywanych przez instrukcje SELECT
- Instrukcje warunków IF …
- Instrukcje rekursji (pętli)
- Kursory
- Obsługa błędów

Wszystkie te konstrukcje będą wykorzystane w materiale części V prezentującej procedury, pakiety i wyzwalacze w środowisku SZBD ORACLE, a także w części VI zawierającej prezentację niektórych bardziej zaawansowanych konstrukcji w tym środowisku.

# VI. PL/SQL procedury składowane, funkcje, wyzwalacze

## 1. Procedury i funkcje jako obiekty bazy danych,

Procedury i funkcje to obiekty bazy danych, zapisywane jako obiekty danych. Mogą być wykorzystywane przez wszystkie procesy korzystające z bazy danych, pod warunkiem, że proces został uruchomiony przez użytkownika lub aplikację posiadającą uprawnienia do danego obiektu – tak jak w przypadku innych obiektów bazy danych.

Procedury i funkcje mogą być definiowane wewnątrz bloków Pl/SQL. Wówczas ich użycie jest ograniczone do tego bloku, w którym zostały zdefiniowane. Procedury i funkcje mogą być obiektami niezależnymi, ale mogą także być umieszczane w większych strukturach zwanych pakietami. Zasadniczą rolą tego typu obiektów jest utworzenie współdzielonego kodu, umieszczonego na serwerze. Mogą one zostać wykorzystane w zasadzie do wszystkich operacji wykonywanych na serwerze, eliminując konieczność sięgania bezpośrednio do tabel. Z uwagi na dużą elastyczność (możliwości programistyczne) użycie procedur może zastąpić widoki, a w dużej mierze także wyzwalacze.

Koncepcja i rola procedur i wyzwalaczy w PL/SQL jest taka sama jak w T-SQL. Jednak ich składnia, a także szereg ograniczeń jakim podlegają, różni znacząco te dwa środowiska i wymaga innego podejścia od programistów.

## 2. Składnia procedury

```
CREATE [OR REPLACE] PROCEDURE nazwa_procedury
    (Lista_parametrów)
{AS | IS}
    Blok PL/SQL bez słaowa kluczowego DECLARE
```

Użycie w składni deklaracji słów **OR REPLACE** oznacza, że jeśli istnieje już procedura o danej nazwie, zostanie ona zastąpiona przez nowy obiekt, ale nie zostanie podniesiony błąd (jak ma to miejsce w T-SQL). Zapis ten jest bardzo wygodny przy testowaniu i wprowadzaniu poprawek do nowo tworzonych procedur, gdzie wielokrotnie zmieniamy treść kodu. Słowa **AS IS** są wymienne (nie ma znaczenia, które zostanie użyte). Lista parametrów (formalnych) procedury, to ujęte w nawiasy i oddzielone przecinkami deklaracje:

```
Nazwa_parametru [typ] typ_danych [DEFAULT = wartość _domyślna]
```

## 3. Parametry procedury

Parametry procedury w Pl/SQL mogą być typu **IN** (domyślny), **OUT**, **IN OUT**.

Parametry typu **IN** służą do odebrania z procesu wywołującego procedurę wartości i przekazania jej do procedury. Wartość przekazana przez parametr **IN** nie ulega (nie może ulec)

zmianie w trakcie realizacji procedury lub funkcji. Parametr **IN** nie może wystąpić po lewej stronie instrukcji podstawienia.

Parametry typu **OUT** służą do przekazywania wartości wyliczonych w procedurze do procesu, który procedurę uruchomił. Warunkiem przekazania wartości jest nadanie wartości parametrowi wewnątrz procedury i poprawne (bez błędu) jej zakończeni.

Parametry typu **IN OUT** mogą zostać wykorzystane zarówno do przekazania do procedury wartości z wywołującego ją procesu, jak też do przekazania wyliczonej w procedurze wartości do tego procesu, po jej poprawnym zakończeniu.

W specyfikacji typu danych parametrów nie podaje się ich rozmiarów(!). Podawany jest tylko typ np. NUMBER, VARCHAR2, albo odwołanie do typu kolumny w tabeli np. emp.sal%TYPE.

*Przykład VI.3.1*

*Utwórz procedurę, która podniesie płace (sal) w tabeli Emp o zadany procent.*

```
CREATE OR REPLACE PROCEDURE UpdSal
   (v_up NUMBER,
    v_deptno NUMBER)
IS
BEGIN
   UPDATE emp
   SET sal = Sal * (1 + v_up/100)
   WHERE deptno = v_deptno;
END;
```

Sposób wywołania procedury zależy od środowiska, w którym procedura ma zostać uruchomiona.

- W Sqldeveloper i Dbeaver

```
CALL UpdSal (5, 20);
```

albo (tylko Sqldeveloper)

```
EXECUTE UpdSal (5, 20);
```

- W kodzie PL/SQL

```
BEGIN
   UpdSal (5, 20);
END;
```

- W SQL*Plus

```
EXECUTE UpdSal (5, 20);
```

*Przykład VI.3.2*

*Zmodyfikuj procedurę z poprzedniego przykładu tak, żeby zwracała nową wartość (po podwyżce) budżetu (sumy) płac działu poprzez parametr typu OUT.*

```
CREATE OR REPLACE PROCEDURE KAEM.UpdSal
   (v_up NUMBER,
   v_deptno NUMBER,
   v_Budzet OUT NUMBER)
/*Procedura podnosi place w dziale podanym przez v_deptno o
procent zadany przez v_up i zwraca budzet plac dzialu po
```

```
podwyżce*/
IS
BEGIN
   UPDATE emp
   SET sal = Sal * (1 + v_up/100)
   WHERE deptno = v_deptno;
   COMMIT;

   SELECT Sum(sal) INTO v_Budzet
   FROM emp
   WHERE deptno = v_deptno;
END;
```

*... i wywołanie procedury z poprzedniego bloku, tym razem w bloku Pl/SQL z powodu konieczności użycia (zatem i zadeklarowania) zmiennych:*

```
DECLARE
   v_deptbudzet NUMBER(10);
   v_Nrdzialu NUMBER;
   V_proc NUMBER;
BEGIN
   v_Nrdzialu:= 30;
   V_proc := 5;
   UpdSal (V_proc, v_Nrdzialu, v_deptbudzet);
   dbms_output.put_line('Aktualny budzet dzialu ' || v_Nrdzialu
   || ' ' || ' wynosi ' || v_deptbudzet);
END;
```

## 4. Wartości domyślne parametrów

Parametry zarówno funkcji jak i procedur mogą mieć nadawane wartości domyślne. Te wartości mogą być pomijane przy wywoływaniu procedur / funkcji, natomiast ich wartości podane w trakcie wywołania zastępują wartości domyślne. Parametry z wartościami domyślnymi umieszczane są na końcu listy parametrów.

```
CREATE OR REPLACE PROCEDURE nazwa_procedury
   (nazwa_parametru typ_danych DEFAULT wyrażenie (, ….))
IS
   (…) ;
```

*Przykład VI.4.1*

```
CREATE OR REPLACE PROCEDURE XXX
   (v_deptno emp.deptno%TYPE DEFAULT 10,
   V_job emp.job%TYPE DEFAULT 'SALESMAN')
AS
BEGIN
   dbms_Output.put_line(v_deptno || ' ' || V_job);
END;
```

Jeżeli został zadeklarowany więcej niż jeden parametr formalny z wartościami domyślnymi, przy wywołaniu procedury wartości nadawane parametrom z wartościami domyślnymi wskazuje się explicite:

```
   CALL nazwa_procedury (…, nazwa_parametru => wyrażenie);
```

Dla przykładowej procedury z poprzedniego slajdu (wartości domyślne deptno = 10, job = SALESMAN) wywołanie mogło by wyglądać następująco:

```
CALL XXX(V_job => 'MANAGER');  10 MANAGER

CALL XXX(v_deptno => 20);      20 SALESMAN

CALL XXX(v_deptno => 20, V_job => 'MANAGER'); 20 MANAGER
```

## 5. Funkcje

Składnia deklaracji funkcji jest zbliżona do deklaracji procedury:

```
CREATE [OR REPLACE] FUNCTION nazwa_funkcji
  (lista_parametrów_formalnych)
RETURN typ_danych
{AS | IS}
  blok Pl/SQL bez słowa kluczowego DECLARE
  z instrukcją RETURN wyrażenie;
```

Różnica pomiędzy funkcją a procedurą polega na sposobie zwracania wyliczonych wartości. Procedura może, ale nie musi zwracać wartości będących wynikiem wykonanych wewnątrz procedury obliczeń. Wartości zwracane przez procedurę przekazywane są „na zewnątrz" poprzez parametry **OUT**.

Funkcja po słowie kluczowym **RETURN** zwraca wartość wyliczoną. Po wyliczeniu tej wartości funkcja kończy działanie i przekazuje „na zewnątrz" wyliczoną wartość pod swoją nazwą.

*Przykład VI.5.1*

*Utwórz funcję, zwracającą liczbę rekordów (czyli liczbę pracowników) tabeli emp w dziale, którego numer (deptno) będzie podawany w parametrze funkcji.*

```
CREATE OR REPLACE FUNCTION IleRek
   (Dno Int)
RETURN INT
AS
   v_ile Int;
BEGIN
   SELECT COUNT(1)
   INTO v_ile
   FROM emp
   WHERE deptno = dno;

   RETURN v_ile;
END;
```

*Skorzystamy z tej funkcji na dwa sposobu. Najpierw w składni SELECT:*

```
SELECT ilerek(30) FROM dual;
```

*... a następnie w bloku PL/SQL:*

```
DECLARE
   v_info Varchar2(50);
   v_liczba Int;
   v_deptno Int := 30;
BEGIN
   v_liczba := ilerek(v_deptno);
   v_info := 'W dziale nr ' || v_deptno || ' jest ' || v_liczba || ' pracowników';
   dbms_output.put_line(v_info);
END;
```

*Przykład VI.5.2*

*Utwórz funkcję zwracającą budżet (sumę płac) działu, którego nazwa zostanie podana w parametrze funkcji.*

```
CREATE OR REPLACE FUNCTION buddzial (v_dname Varchar2)
RETURN NUMBER
IS
   V_Budzet Number;
BEGIN
   SELECT Sum(sal)
   INTO V_Budzet
   FROM emp e JOIN dept d ON e.deptno = d.deptno
   WHERE dname = v_dname;
   RETURN V_Budzet;
END;
```

… i użycie funkcji:

```
DECLARE
   p_Sumaplac Number;
BEGIN
   p_Sumaplac := buddzial('SALES');
   dbms_output.put_line(p_Sumaplac);
END;
```

## 6. Funkcje – użycie w poleceniach SQL

Jak widać funkcje definiowane w bazie danych mogą być używane w poleceniach SQL tak jak inne funkcje implementowane przez SZBD.

Ograniczenie w ich wykorzystaniu wewnątrz poleceń SQL stanowi:

zakaz użycia w ich treści instrukcji DDL i DML (funkcje nie mogą zmieniać stanu bazy danych),

- nie mogą posiadać parametrów wyjściowych OUT,
- nie powinny korzystać ze zmiennych nie – lokalnych w pakietach,
- wszystkie parametry muszą zostać wyspecyfikowane,
- nie wolno używać w nich notacji parametr => wartość.

## 7. Przeciążenie nazw procedur i funkcji

Pl/SQL dopuszcza wielokrotne użycie tej samej nazwy funkcji czy procedury. Jest to wygodne, gdy ta sama (z logicznego punktu widzenia) procedura jest wykorzystywana różnie,

zależnie od kontekstu, np. z różną liczbą parametrów lub różnymi typami danych parametrów. Wersje tej samej procedury (funkcji) muszą różnić się liczbą parametrów, albo nazwą i typem parametrów. Jest to warunek konieczny dla rozróżnienia przez system tych wariantów i wybrania właściwej wersji.

Dopuszczalne jest zastosowanie takiego rozwiązania wyłącznie w procedurach utworzonych w ramach jednego bloku lub jednego pakietu, zatem niedopuszczalne jest przeciążanie nazw procedur niezależnych (standelone). Niedopuszczalne jest również zróżnicowanie tylko podtypów danych (np. Char i Varchar2, Int i Real itd.). Poniżej przykład użycia tak zdefiniowanych procedur w bloku anonimowym, wykorzystanych do odczytania budżetu płac działu, jeśli podawana jest jego nazwa (dname) albo numer (deptno).

*Przykład VI.7.1*

*Utwórz procedury o jednakowej nazwie, zwracające wartość budżetu (sumę płac) działu, który jest identyfikowany albo przez numer (deptno), albo przez swoją nazwę (dname).*

```
DECLARE
   v_budzet NUMBER(8,2);
   v_num INT;
   v_name Varchar2(30);
   PROCEDURE emp_dept (v_bud OUT NUMBER, v_deptno INT)
   IS
   BEGIN
      SELECT SUM(sal) INTO v_bud
      FROM emp
      WHERE deptno = v_deptno;
   END;
   PROCEDURE emp_dept (v_bud OUT NUMBER, v_dname Varchar2)
   IS
   BEGIN
      SELECT SUM(sal) INTO v_bud
      FROM emp
      WHERE deptno =
               (SELECT deptno
               FROM dept
               WHERE dname = v_dname);
   END;
BEGIN
   v_num := 10;
   v_name := 'SALES';
IF v_num IS NULL THEN
   emp_dept(v_budzet, v_name);
ELSE
   emp_dept(v_budzet, v_num);
END IF;
dbms_output.put_line(v_budzet);
END;
```

## 8. Pakiety

Z uwagi na dużą liczbę procedur, funkcji, sekwencji, jakie zwykle powstają podczas tworzenia aplikacji, dużym ułatwieniem jest możliwość grupowania ich w większe jednostki zwane pakietami (packages). W ramach pakietu możemy zdefiniować: ➢ kursory ➢ zmienne i stałe ➢ wyjątki ➢ podprogramy (funkcje i procedury) Pakiet zazwyczaj składa się z dwóch

części – specyfikacji pakietu (specification) i ciała pakietu (body). Specyfikacja (część publiczna pakietu) stanowi interfejs dla aplikacji i zawiera deklaracje typów, zmiennych, stałych, wyjątków, kursorów i podprogramów możliwych do użycia przez aplikację korzystającą z pakietu. Ciało pakietu (część prywatna) zawiera pełne definicje kursorów i podprogramów, czyli implementuje specyfikację. Czas życia zmiennych i stałych pakietu ogranicza się do sesji, która pakiet wywołała.

```
CREATE OR [REPLACE] PACKAGE nazwa_pakietu
AS | IS
  <deklaracja obiektów publicznych; w przypadku procedur i
  funkcji specyfikacja nagłówków>
END nazwa_pakietu
CREATE OR [REPLACE] PACKAGE BODY nazwa_pakietu
AS | IS
  <definicje obiektów publicznych i prywatnych>
[BEGIN]
  <instrukcje inicjalizujące>
END nazwa_pakietu;
```

Część prywatna pakietu jest opcjonalna - pakiet może zawierać tylko deklaracje. Podprogramy deklarowane w części publicznej pakietu muszą znaleźć się na końcu – muszą być poprzedzone wszystkimi innymi deklaracjami. Instrukcje inicjalizujące wykonywane są tylko jeden raz – przy pierwszym uruchomieniu pakietu w sesji.

*Przykład VI.8.1*

*Utwórz pakiet, realizujący operacje zatrudnienia (dopisania do bazy) nowego pracownika, oraz zwolnienia (usunięcia z bazy) wskazanego pracownika.*

```
CREATE OR REPLACE PACKAGE Obsluga_Prac
AS
v_zat INT; v_zwol INT;
PROCEDURE Zatrudnij (v_ename VARCHAR2
                    ,v_sal NUMBER
                    ,v_dname VARCHAR2);
PROCEDURE Zwolnij (v_empno INT);
END Obsluga_Prac;

--/ to musi wystąpić w SQL*Plus oraz Sqldeveloper

CREATE OR REPLACE PACKAGE BODY Obsluga_Prac
AS
PROCEDURE Zatrudnij (v_ename VARCHAR2
                    ,v_sal NUMBER
                    ,v_dname VARCHAR2)
AS
   v_empId INT;
BEGIN
   SELECT NVL(Max(empno ), 0) + 1 INTO v_empId
   FROM emp;

   INSERT INTO emp (empno, ename, sal, deptno, hiredate)
   SELECT v_empId, v_ename, v_sal, deptno, SYSDATE
   FROM dept
   WHERE dname = v_dname;

   COMMIT;
```

```
   v_zat := v_zat + 1;
END Zatrudnij;

PROCEDURE Zwolnij (v_empno INT)
IS
BEGIN
   DELETE FROM emp
   WHERE empno = v_empno;
   COMMIT;
   v_zwol := v_zwol + 1;
END Zwolnij;
BEGIN
   v_zwol := 0;
   v_zat := 0;
END Obsluga_Prac;
```

*Wywołanie procedury Zatrudnij:*

```
BEGIN
obsluga_prac.zatrudnij('Malinowski', 1225, 'SALES');
END;
```

*Wywołanie procedury Zwolnij:*

```
BEGIN
obsluga_prac.zwolnij(7935);
dbms_output.put_line(obsluga_prac.v_zat);
END;
```

## 9. Wyzwalacze

Wyzwalacze w Pl/SQL są to procedury wiązane z jednym z obiektów:

- tabelą
- perspektywą
- schematem (kontem użytkownika)
- całą bazą danych

Wyzwalacze są uruchamiane przez SZBD w wyniku zaistnienia odpowiedniego zdarzenia, które może być zdarzeniem systemowym, albo jedną z instrukcji INSERT, UPDATE, DELETE zachodzącym na tabeli lub perspektywie, z którą związany jest wyzwalacz. Rola wyzwalaczy w Pl/SQL, związanych z tabelami i wyzwalaczy INSTAED OFF związanych z perspektywami jest identyczna, jak rola wyzwalaczy w T-SQL. Jednak sama „filozofia" konstrukcji i działania wyzwalaczy różni się zasadniczo pomiędzy tymi językami.

## 10. Wyzwalacze na tabelach bazy danych

Przy definiowaniu wyzwalacza w Pl/SQL określa się:

- z jaką sekwencją instrukcji **INSERT**, **UPDATE**, **DELETE** wyzwalacz będzie związany (czyli jakie operacje na tabeli mogą go uruchomić),
- czy działanie wyzwalacza będzie dotyczyło pojedynczych wierszy (wyzwalacz wierszowy), czy też całej tabeli,

- czy wyzwalacz zostanie uruchomiony przed (**BEFORE**), czy też po (**AFTER**) wykonaniu instrukcji, która go uruchamia.

Składnia deklaracji wyzwalacza w ORACLE wygląda następująco

```
CREATE OR [REPLACE] TRIGGER nazwa_wyzwalacza
BEFORE | AFTER specyfikacja_instrukcji_DML
ON nazwa_taneli
[FOR EACH ROW]
Blok PL/SQL
```

Specyfikacja instrukcji DML to ciąg do trzech nazw instrukcji **INSERT**, **UPDATE**, **DELETE** połączonych spójnikiem **OR**. Kolejność specyfikacji instrukcji nie ma znaczenia. W przypadku **UPDATE** można podać nazwy kolumn, których modyfikacja ma uruchamiać wyzwalacz:

```
UPDATE OF kolumna (, …)
```

### a) Dwa typy wyzwalaczy PL/SQL

PL/SQL implementuje dwa typy wyzwalaczy:

- Wyzwalacz „wierszowy" – **FOR EACH ROW**, w którym operacje wykonywane są kolejno na wszystkich zmienianych (aktualizowanych, wstawianych, usuwanych) wierszach tabeli. W wyzwalaczu wierszowym dostęp do wartości w zmienianych wierszach dostępne są przez odwołania:

  :**OLD.wyrażenie** – wartość przed zmianą

  **:NEW.wyrażenie** – wartość po zmianie

- Wyzwalacz działający w kontekście całej tabeli, a nie poszczególnych wierszy.

### b) Ograniczenia dotyczące operacji na wyzwalaczach

- W wyzwalaczach NIE WOLNO używać poleceń **COMMIT** ani **ROLLBACK**
- W wyzwalaczach wierszowych (**FOR EACH ROW**) odczytywać ani zmieniać wartości z tabeli zmienianej (poza odwołaniami **:OLD** i **:NEW**). Dopuszczalne jest wstawianie nowego wiersza poleceniem **INSERT** … **INTO** … **VALUES**. Tabela zmieniana to tabela, z którą związany jest wyzwalacz, ale także każda tabela odwołująca się do tej tabeli przez więzy referencyjne

### c) Identyfikacja instrukcji DML która uruchomiła wyzwalacz

Istnieje możliwość sprawdzenia, która instrukcja uruchomiła wyzwalacz. Służą do tego predykaty

**INSERTING**, **UPDATING**, **DELETING**

które mogą zostać użyte w instrukcji warunkowej:

```
IF {DELETING | INSERTING | UPDATING} THEN … END IF;
```

Informacja ta może być istotna, jeżeli wyzwalacz może być uruchomiony przez więcej niż jedną instrukcję DML.

### d) Kilka wyzwalaczy związanych z jedną tabelą

Z jedną tabelą można powiązać wiele wyzwalaczy. Jednak nie istnieje możliwość sterowania kolejnością ich uruchamiania (decyduje tutaj SZBD). W związku z tym nie należy tworzyć wyzwalaczy, których działanie wiąże się z określoną kolejnością uruchamiania.

Jednak w przypadku zdefiniowania kilku wyzwalaczy na jednej tabeli, ogólną zasadę kolejności ich uruchamiania można przedstawić następująco:

1. Wyzwalacz przed instrukcją,
2. wyzwalacz przed pierwszym wierszem, na którym operuje instrukcja,
3. wyzwalacz po pierwszym wierszu, na którym operuje instrukcja,
4. …
5. wyzwalacz przed ostatnim wierszem,
6. wyzwalacz po ostatnim wierszu,
7. wyzwalacz po instrukcji.

*Przykład VI.10.1*

*Utwórz na tabeli emp wyzwalacz, który po każdej zmianie dotyczącej płacy (sal) wpisze nowy rekord do tabeli BUDZET {Wartosc, DataAktualizacji}*

```
CREATE OR REPLACE TRIGGER KAEM.emp_UpdBudzet
AFTER INSERT OR UPDATE OF sal OR DELETE
ON emp

DECLARE
   V_budzet Number(10,2);
   V_info Varchar2(50);
   v_data DATE;

BEGIN
   v_data := Sysdate;
   SELECT Sum(sal) INTO V_budzet FROM emp;
   INSERT INTO Budzet (Wartosc, DataAktualizacji)
   VALUES (V_budzet, Sysdate);
   v_info := 'Budzet na dzień ' || v_data || ' wynosi ' || v_budzet || ';';
   dbms_output.put_line(v_info);
END;
```

*W tym wyzwalaczu w trakcie jego działania odczytywana jest wartość SUM(sal) ze zmienianej tabeli emp. Jest to dopuszczalne, gdyż nie jest to wyzwalacz FOR EACH ROW.*

A teraz będzie „antyprzykład”.

*Przykład VI.10.2*

*Utwórz na tabeli emp wyzwalacz, który nie dopuści do takiej indywidualnej podwyżki płac, która przekraczała by o więcej niż 1.5 razy wartość aktualnej średniej płacy w dziale, w którym jest zatrudniony pracownik, któremu podwyższana jest płaca.*

```
CREATE OR REPLACE TRIGGER control_sal
BEFORE INSERT OR UPDATE OF SAL
ON EMP
FOR EACH ROW
DECLARE
   v_sal NUMBER(6,2);
   v_empno Int;
   v_deptno Int;
   v_avsal NUMBER(7,2);
```

```
BEGIN
   v_sal := :NEW.sal;
   v_empno := :NEW.empno;
   v_deptno := :NEW.deptno;
   SELECT AVG(sal) INTO v_avsal
   FROM emp
   WHERE deptno = v_deptno;

   IF v_sal > 1.5 * v_avsal THEN
      :NEW.sal := :OLD.sal;
   END IF;
END;
```

*I wszystko wydaje się być poprawne, wyzwalacz się kompiluje, ale przy próbie wykonania podwyżki:*

```
UPDATE emp SET sal = sal*2 WHERE deptno = 10;
```

*podnoszony jest błąd*

SQL Error [4091] [42000]: ORA-04091: table KAEM.EMP is mutating, trigger/function may not see it

ORA-06512: at "KAEM.CONTROL_SAL", line 10

ORA-04088: error during execution of trigger 'KAEM.CONTROL_SAL'

*wskazujący na to, że tabela ulega zmianie i wyzwalacz (FOR EACH ROW) nie może operować w trakcie tych zmian, , a cała operacja zostaje wycofywana.*

*Przykład VI.10.3*

*Utwórz na tabeli EMP wyzwalacz, który nie dopuści do takiej zmiany płacy (sal), która zmieniała by aktualną grupę zarobkową pracownika (grade).*

```
CREATE OR REPLACE TRIGGER In_Grade_Trigg
BEFORE UPDATE ON emp
FOR EACH ROW
DECLARE
   v_Ograde salgrade.grade%type;
   v_Ngrade salgrade.grade%type;
BEGIN
   SELECT grade INTO v_Ograde FROM Salgrade
   WHERE :old.sal Between Losal AND Hisal;

   SELECT grade INTO v_Ngrade FROM Salgrade
   WHERE :new.sal Between Losal AND Hisal;

   IF v_Ograde != v_Ngrade THEN
      :new.sal := :old.sal;
   dbms_output.put_line('Nie zmieniamy grupy zarobkowej!');
END IF;
END;
```

**UWAGA:**

Wyzwalacze z obu powyższych przykładów działają na tej samej tabeli *emp*, przy czym wyzwalacz z przykładu VI.10.2 jest nieprawidłowy i może blokować działanie wyzwalacza z przykładu VI.10.3, wobec czego nieprawidłowy wyzwalacz należy albo usunąć poleceniem:

```
DROP TRIGGER nazwa_wyzwalacza;
```

albo zablokować (wyłączyć) jego działanbie poleceniem:

```
ALTER TRIGGER nazwa_wyzwalacza DISABLE];
```

Wyzwalacz może zostać ponownie włączony poleceniem

```
ALTER TRIGGER nazwa_wyzwalacza ENABLE;
```

## 11. Wyzwalacze INSTEAD OF

ORACLE oferuje możliwość prowadzenia operacji DML poprzez perspektywy utworzone na więcej niż jednej tabeli (patrz możliwość aktualizacji danych poprzez perspektywy). Realizacja tego typu operacji możliwa jest przy użyciu wyzwalaczy INSTEAD OF definiowanych dla perspektyw. W przeciwueństwie do MS SQL Server w ORACLE wyzwalacze INSTEAD OF nie mogą być łączone z tabelami.

Składnia tego typu wyzwalaczy wygląda następująco:

```
CREATE [OR REPLACE] TRIGGER nazwa_wyzwalacza
INSTEAD OF specyfikacja_instrukcji_DML
ON nazwa_perspektywy [FOR EACH ROW]
blok Pl/SQL
```

Blok instrukcji Pl/SQL wraz z instrukcjami SQL pozwala na wykonanie niezależnych operacji DML na wszystkich tabelach, do których odwołuje się perspektywa, podczas gdy perspektywa izoluje szczegóły tych operacji (zatem także strukturę tabel) od użytkownika.

W części wykładu dotyczącej MS SQL Server został zprezentowany przykład utworzenia wyzwalacza INSTEAD OF. Pozostawiam czytelnikowi przepisanie tego wyzwalacza i uruchomienie go w środowisku ORACLE.

## 12. Wyzwalacze systemowe (bazy danych)

ORACLE pozwala zdefiniować wyzwalacze uruchamiane nie przez operacje DML związane z tabelą lub perspektywą, lecz przez zdarzenia zachodzące na bazie danych (zdarzenia bazodanowe) i zdarzenia DDL.

Zdarzenia bazodanowe to:

- **SERVERROR**,
- **LOGON**,
- **LOGOFF**,
- **STARTUP**,
- **SHUTDOWN**.

Zdarzenia DDL to nazwy instrukcji DDL i DCL takie jak:

- **CREATE**,
- **ALTER**,
- **DROP**,
- **GRANT**,
- **REVOKE**.

Zdarzenia mogą być łączone w jednym wyzwalaczu systemowym za pomocą operatora OR. Składnia tego typu wyzwalaczy wygląda następująco:

```
CREATE [OR REPLACE] TRIGGER nazwa_wyzwalacza
[BEFORE | AFTER | INSTEAD OF][zdarzenie bazodanowe|DDL]
ON [DATABASE | SCHEMA]
blok Pl/SQL
```

# VII. PL/SQL – konstrukcje (nieco) bardziej zaawansowane

Podobnie jek w przypadku T-SQL przdstawiam kilka rozwiązań które rozszerzają narzędzia dostępne w programowaniu w środowisku DB ORACLE, jednak nie są niezbędne przy wykonywaniu podstawowych operacji.

## 1. Nazwy w ORACLE Database

- Wszystkie nazwy (identyfikatory) w ORACLE, także nazwy obiektów, muszą spełniać poniższe warunki:
- Maksymalna długość nazwy wynosi 30 znaków. Dotyczy to także nazw więzów referencyjnych tworzonych domyślnie jako złożenie nazw powiązanych tabel.
- Pierwszym znakiem musi być litera (alfabetu łacińskiego), kolejnymi mogą być litery, cyfry (0-9), znak dolara ($), podkreślenia „podłogi” (_), oraz hash (#).
- PL/SQL jest case-insensitive, czyli w przypadku nazw nie rozróżniane są małe i wielkie litery
- Dopuszcza się tworzenie nazw ujętych w podwójne cudzysłowy i zawierające inne znaki, niż wyżej wymienmione, np.:

```
DECLARE
   "A B C " Varchar2(10);
BEGIN
   "A B C " := 'Alamakota';
   dbms_output.put_line("A B C ");
END;
```

ORACLE zapisuje wszystkie nazwy w postaci wielkich liter (uppercase), chyba że zastosujemy wariant z podwójnym cudzysłowem. Wówczas nazwa jest przechowywana tak, jak została zapisana, ale też w ten sam sposób musi być przywoływana. Ta właściwość dotyczy wszystkich nazw w ORACLE.

```
DECLARE
   "abc" Varchar2(10);
BEGIN
"abc" := 'Alamakota';
   dbms_output.put_line("abc"); -- to polecenie zostanie wykonane poprawnie
   dbms_output.put_line("ABC"); -- to polecenie wywoła błąd: brak deklaracji
                                -- zmiennej
END;
```

## 2. Autoinkrementacja kolumn w tabelach ORACLE.

W bazie danych ORACLE od wersji 12c (r. 2014) istnieje możliwość zrealizowania autoinkrementacji wartości w kolumnie tabeli poprzez wykorzystanie IDENTITY, podobnie

jak na to niejscce w MS SQL Server. Ta metoda jest szeroko wykorzystywana do generowania wartości kluczy głównych w tabelach baz danych. W wersjach wcześniejszych można było to uzyskać wykorzystując w tym celu sekwencję (Sequence) lub wyzwalacz (Trigger). Oczywiście te dwie możliwości są nadal dostępne. Metoda IDENTITY występuje w ORACLE w trzech wariantach, co czyni ją bardziej elastyczną, niż ma to miejsce w SQL Server. Te warianty to:

- ALWAYS AS IDENTITY
- BY DEFAULT AS IDENTITY
- BY DEFAULT ON NULL AS IDENTITY

Każda z tych metod daje nieco inne możliwości niż pozostałe. Przedstawimy to na przykładach.

### a) ALWAYS AS IDENTITY

Używając opcji ALWAYS AS IDENTITY wymuszamy użycie nowej wartości wyliczonej przez serwer, bez możliwości „ręcznej" ingerencji w tą wartość. Zatem mamy przypadek takiego działania, jak ma to miejsce w MS SQL Server.

*Przykład VII.2.1*

```
DROP TABLE emp; --Table EMP dropped.
DROP TABLE DEPT; --Table DEPT dropped.
```

```
CREATE TABLE DEPT (
   Deptno Integer GENERATED ALWAYS AS IDENTITY START WITH 10 INCREMENT BY 10,
   Dname Varchar2(20) NOT NULL,
   Loc Varchar2(20) NOT NULL,
   PRIMARY KEY (Deptno)
                 ); --Table DEPT created.
```

```
INSERT INTO DEPT (Dname, Loc)
VALUES ('ACCOUNTING', 'NEW YORK'); --1 row inserted.
```

```
INSERT INTO DEPT (Deptno, Dname, Loc)
VALUES (20, 'RESEARCH', 'DALLAS');
```

Próba wykonania tego ostatniego polecenia kończy się błędem:

```
SQL Error [32795] [99999]: ORA-32795: cannot insert into a generated always
identity column
```

### b) BY DEFAULT AS IDENTITY

Opcja BY DEFAULT AS IDENTITY pozwala na wybranie opcji w poleceniu INSERT INTO - czy serwer ma wygenerować automatycznie nową wartość w kolumnie, czy też ma zaakceptować wartość podaną explicite w instrukcji INSERT.

*Przykład VII.2.2*

```
DROP TABLE DEPT; --Table DEPT dropped.
```

```
CREATE TABLE DEPT (
   Deptno Integer GENERATED BY DEFAULT AS IDENTITY START WITH 10 INCREMENT BY 10,
   Dname Varchar2(20) NOT NULL,
   Loc Varchar2(20) NOT NULL,
   PRIMARY KEY (Deptno)
                 ); --Table DEPT created.
```

```
INSERT INTO DEPT (Dname, Loc)
VALUES ('ACCOUNTING', 'NEW YORK'); --1 row inserted.

INSERT INTO DEPT (Deptno, Dname, Loc)
VALUES (50,'RESEARCH', 'DALLAS'); --1 row inserted.

INSERT INTO DEPT (Dname, Loc)
VALUES ('SALES', 'CHICAGO'); --1 row inserted.

SELECT * FROM DEPT ;

  10 ACCOUNTING NEW YORK
  20 SALES CHICAGO
  50 RESEARCH DALLAS
```

Ale generator nie kontroluje wartości wstawionych explicite, co może prowadzić do próby wstawienia wartości już istniejącej i podniesienia błędu, co czyni użycie tego wariantu dość iluzorycznym.

### c) BY DEFAULT ON NULL AS IDENTITY

Wybór opcji BY DEFAULT ON NULL AS IDENTITY spowoduje, że serwer wygeneruje nową wartość wtedy, kiedy w poleceniu INSERT INTO albo nie pojawi się odwołanie do kolumny z metodą IDENTITY, albo jako wartość do wypełnienia tej kolumny pojawi się NULL.

*Przykład VII.2.3*

```
DROP TABLE DEPT ; --Table DEPT dropped.

CREATE TABLE dept (
  Deptno Integer GENERATED BY DEFAULT ON NULL AS IDENTITY
  START WITH 10 INCREMENT BY 10,
  Dname Varchar2(20) NOT NULL,
  Loc Varchar2(20) NOT NULL,
  PRIMARY KEY (Deptno)
                );  --Table DEPT created.

INSERT INTO dept (Dname, Loc)
VALUES ('ACCOUNTING','NEW YORK'); --1 row inserted.

INSERT INTO dept (Deptno, Dname, Loc)
VALUES (50,'RESEARCH','DALLAS'); --1 row inserted.

INSERT INTO dept (Deptno, Dname, Loc)
VALUES (NULL,'OPERATIONS','BOSTON'); --1 row inserted.

SELECT * FROM dept;

  10 ACCOUNTING NEW YORK
  50 RESEARCH DALLAS
  20 SALES CHICAGO
```

### d) Ograniczenia dla opcji IDENTITY

Użycie opcji IDENTITY w ORACLE ma swoje ograniczenia:

- Tylko jedna kolumna w tabeli może używać tej opcji,
- Kolumna musi być typu numerycznego i nie może być to typ definiowany przez użytkownika,
- Taka kolumna nie może używać więzów DEFAULT,

- Tabela tworzona automatycznie na podstawie odczytu danych z innej tabeli (tabel) nie dziedziczy właściwości IDENTITY
- ORACLE nie implementuje mechanizmu umożliwiającego łatwy odczyt ostatnio wygenerowanej wartości przez metodę IDENTITY (tak jak ma to miejsce w MS SQL Server – zmienna sysytemowa @@Identity lub funkcja Scope_Identity).

#### e) Inne niż IDENTITY rozwiązanie autoinkrementacji wartości kolumny

W wersjach serwera ORACLE starszych niż 12c rozwiazać problem możemy albo tworząc obiekt(y) klasy sekwencja (Sequence), albo tworząc wyzwalacz wyspecjalizowany do wykonania tych operacji. Oczywiście te rozwiązania mogą być użyte także w nowszych wersjach.

*Przykład VII.2.4*

```
DROP TABLE DEPT ; --Table DEPT dropped.

CREATE TABLE DEPT (
DEPTNO NUMBER(2) PRIMARY KEY,
DNAME VARCHAR2(14),
LOC VARCHAR2(13)); --Table DEPT created.

CREATE SEQUENCE seq_dept
START WITH 10 INCREMENT BY 10; --Sequence SEQ_DEPT created.

INSERT INTO DEPT VALUES
(seq_dept.nextval, 'ACCOUNTING', 'NEW YORK'); --1 row inserted.

BEGIN
DBMS_OUTPUT.put_line
('Ostatnio dodany rekord tabeli dept ma wartość deptno = ' || seq_dept.currval);
END;
```

Użycie sekwencji ma jednak pewne zalety w stusunku do użycia IDENTITY, zwłaszcza że przy użyciu sekwencji można generować wartości w więcej niż jednej kolumnie tabeli. We wszystkich powyższych przykładach, zarówno z użyciem IDENTITY jak też w sekwencji, użyłem możliwości zdefiniowania wartości początkowej (START WITH) oraz wartości inkrementacji (INCREMENT BY). Są to elementy opcjonalne. Jeżeli zostaną pominięte, zarówno jako wartość początkowa, jak też wartość inkrementacji zostanie użyte 1. Sekwencje posiadają także inne opcje (m.in. możliwość użycia cyklu), ale tutaj dociekliwych odsyłam już do dokumentacji ORACLE Database.

## 3. Tabele tymczasowe

ORACLE DB przewiduje możliwość użycia dwóch rodzajów tabel tymczasowych: GLOBAL (dostępne od wersji 8i) i PRIVATE (od wersji 18c), Koncepcja wykorzystania tych tabel różni się znacznie od rozwiązań znanych z MS SQL Server.

#### a) Global Temporary Tables

Tabela tymczasowa Global (GTT) jest TRWAŁYM obiektem bazy danych, przechowującym dane na dysku i widocznym dla wszystkich sesji. Jednakże dane przechowywane w GLOBAL TEMPORARY TABLE są danymi prywatnymi sesji, która te dane w niej umieściła. Inaczej mówiąc, każda sesja ma dostęp wyłącznie do własnych danych umieszczonych w tego typu tabeli.

Jakkolwiek tabela tymczasowa GTT jest obiektem trwałym, to dane w niej umieszczone są w niej przechowywane albo do końca transakcji, albo do końca sesji.

Polecenie utworzenia tabeli tymczasowej:

```
CREATE GLOBAL TEMPORARY TABLE nazwa_tabeli (
      Nazwa_kolumny_1 Typ_danych Więzy_integralności
      (…)
      Więzy_integralności
      )
ON COMMIT [DELETE ROWS] | [PRESERVE ROWS];
```

Klauzula ON COMMIT wskazuje, czy dane zapisane w tabeli są trwałe w obrębie transakcji, czy sesji.

- Opcja ON COMMIT DELETE ROWS decyduje, że dane są danymi specyficznymi dla transakcji i SZBD wykona operacje Truncate na tabeli (usunie wszystkie wiersze) po zakończeniu transakcji przez COMMIT.
- Opcja ON COMMIT PRESERVE ROWS decyduje, że dane są danymi specyficznymi dla sesji i oznacza wykona operacje Truncate na tabeli po zakończeniu sesji, a nie po zatwierdzeniu transakcji.

Opcja ON COMMIT DELETE ROWS jest opcją domyślną i zostanie zastosowana, jeśli klauzula ON COMMIT zostanie pominięta przy definicji tabeli tymczasowej.

- Na tabelach tymczasowych GTT mogą być zakładane indeksy.
- Kilka sesji na raz może korzystać z jednej tabeli tymczasowej GTT, mając dostęp wyłącznie do „swoich" danych.
- Operacje DDL (za wyjątkiem TRUNCATE) na tabelach tymczasowych GTT nie mogą być wykonane, jeżeli jakakolwiek sesja z nich korzysta.
- Operacja **TRUNCATE TABLE** *tabela_tymczasowa* powoduje usunięcie z tabeli wyłącznie „własnych" rekordów sesji.
- Tabele tymczasowe (wraz z danymi) nie podlegają operacji BACKUP, czyli nie zostaną odtworzone przez RESTORE.

*Przykład VII.3.1*

*Utwórzmy tabelę tymczasową Global w aktualnej sesji 1 (zakładce Dbeaver lub Sqldeveloper)*

```
CREATE GLOBAL TEMPORARY TABLE temp1 (
Id Int,
Info Varchar2(50)
) ON COMMIT PRESERVE ROWS;
```

*I dopiszmy do niej jeden wiersz*

```
INSERT INTO temp1 (Id, Info)
VALUES (1, 'Dane sesji 1');
```

*A następnie odczytajmy zawartość tabeli w tej sesji*

```
SELECT * FROM Temp1;
```

```
1|Dane sesji 1|
```

*Otwórzmy nową sesję 2 (nową zakładkę Dbeaver lub Sqldeveloper). Teraz polecenie zwraca*

```
   SELECT * FROM Temp1;
```

```
0 row(s) fetched.
```

*Pozostając w nowej sesji dopiszmy kolejny wiersz*

```
INSERT INTO temp1 (Id, Info)
VALUES (1, 'Dane sesji 2');
```

*I odczytajmy zawartość tabeli*

```
   SELECT * FROM Temp1;
```

```
1|Dane sesji 2|
```

*Pozostając w sesji 2*

```
TRUNCATE TABLE Temp1;
SELECT * FROM Temp1;
```

```
0 row(s) fetched.
```

*Wróćmy do sesji 1*

```
SELECT * FROM Temp1;
```

```
1|Dane sesji 1|
```

*Spróbujmy usunąć tabelę*

```
DROP TABLE Temp1;
```

```
SQL Error [14452] [72000]: ORA-14452: attempt to create, alter or drop an
index on temporary table already in use
```

*Zamknijmy zatem sesję 2 i ponówmy próbę usunięcia tabeli*

```
DROP TABLE Temp1;
```

```
SQL Error [14452] [72000]: ORA-14452: attempt to create, alter or drop an
index on temporary table already in use
```

*Zamknijmy zatem sesję 1 i spróbujmy odwołać się do tabeli temp1 z kolejnej otwartej sesji, która dotąd nie dowoływała się do tej tabeli*

```
SELECT * FROM Temp1;
```

```
SQL Error [942] [42000]: ORA-00942: table or view does not exist
```

*Jak widać tabela została usunięta wraz z końcem ostatniej sesji, która z niej korzystała.*

### b) Private Temporary Tables (od wersji 18c)

W przypadku tabel **Private** (PTT) zarówno definicja tabeli jak też dane w niej przechowywane są faktycznie tymczasowe i zostają usunięte wraz z końcem transakcji lub sesji – zależnie od opcji z którą zostały utworzone. Tabele tego typu są przechowywane w pamięci RAM i są widoczne wyłącznie dla sesji, w których zostały utworzone.

Nazwy tabel PTT muszą zostać poprzedzone prefixem **ORA$PTT_**

```
CREATE PRIVATE TEMPORARY TABLE nazwa_tabeli (
     Nazwa_kolumny_1 Typ_danych Więzy_integralności
     (…)
     )
ON COMMIT [DROP] | [PRESERVE] DEFINITION;
```

Znaczenie opcji w klauzuli ON COMMIT jest identyczne jak w przypadku tabel **GTT**.

Ograniczenia dla tabel **PTT**:

- Nazwa tabeli musi być poprzedzona prefiksem ORA$PTT_,
- Trwałe obiekty bazy danych nie mogą bezpośrednio odwoływać się do tabel PTT,
- Na tabelach PTT nie mogą być zakładane indeksy ani na ich podstawie nie mogą być tworzone perspektywy zmaterializowane,
- Columny tabel PTT nie mogą mieć przypisanych wartośći domyślnych (Default)
- Tabele PTT nie są dostępne z innych baz danych.

*Przykład VII.3.2*

```
CREATE PRIVATE TEMPORARY TABLE ora$ptt_my_temp_table (
id NUMBER,
description VARCHAR2(20)
)
ON COMMIT DROP DEFINITION;

Private TEMPORARY created.

INSERT INTO ora$ptt_my_temp_table VALUES (1, 'ONE');

1 row inserted.

SELECT COUNT(*) FROM ora$ptt_my_temp_table;

1| 1|

COMMIT;

Commit complete.

SELECT COUNT(*) FROM ora$ptt_my_temp_table;

ORA-00942: tabela lub perspektywa nie istnieje
```

**UWAGA!**

Powyższy przykład został wykonany w Sqldeveloper; w Dbeaver nie działa!!!

## 4. Common Table Expression z rekursją, CASE, skorelowany Update

Ponieważ w tych zagadnieniach w zasadzie nie ma istotnych różnic pomiędzy MS SQL Server a ORACLE DB, pomijam ich omówienbie w składni ORACLE, a zainteresowanych odsyłam do poświęconych tym zaganieniom rozdziałów w części dotyczącej MS SQL Server.

## 5. Funkcje "rankingowe" ORACLE

Funkcje "rankingowe" SQL Server to funkcje rozszerzające możliwości klauzuli ORDER BY i umożliwiające numerowanie rekordów wynikowych poleceń SELECT. Dla ORACLE omawiam tylko funkcje:

- ROW_NUMBER
- RANK
- DENSE_RANK
- LISTAGG

### a) Funkcja ROW_NUMBER

Funkcja **ROW_NUMBER** numeruje rekordy wynikowe polecenia **SELECT**, umożliwiając też utworzenie partycji (grup rekordów), dla których tworzone są oddzielne sekwencje numeracji.

Składnia funkcji:

```
ROW_NUMBER() OVER ( [ PARTITION BY wyrażenie , ... [ n ] ]
klauzula ORDER BY )
```

Numeracja zaczyna się od 1, zgodnie z kierunkiem sortowania. Jeżeli została użyta opcjonalna klauzula PARTITION BY, numeracja odbywa się w obrębie każdej partycji (grupy rekordów).

*Przykład VII.5.1*

*Poniższe przykłady bez komentarza*

```
SELECT ename,
       deptno,
       row_number() OVER(ORDER BY ename) Nr
FROM emp;
```

```
SELECT ename,
       deptno,
       row_number() OVER(PARTITION BY deptno ORDER BY ename) Nr_w_dziale
FROM emp;
```

*Przykład VII.5.2*

*Przypisz każdemu pracownikowi jego "pozycję" płacową na jego stanowisku*

```
SELECT ROW_NUMBER() Over (PARTITION BY job ORDER BY sal Desc) Pozycja,
       job Stanowisko,
       ename Nazwisko,
       sal Płaca
FROM emp;
```

*Przykład VII.5.3*

*Wypisz nazwiska, stanowiska, płace wszystkich pracowników wraz z sumą płac, średnią, minimalną i maksymalną płacą na ich stanowiskach.*

```
SELECT ename,
       job,
       sal,
       Sum(sal) OVER (PARTITION BY job) job_Sumsal,
       Avg(sal) OVER (PARTITION BY job) job_Avsal,
       Min(sal) OVER (PARTITION BY job) job_Minsal,
       Max(sal) OVER (PARTITION BY job) job_Maxsal
FROM emp;
```

### b) Funkcja agregująca RANK

```
RANK (wyr1 [,wyr2,..wyrN]) WITHIN GROUP
(ORDER BY wyr1 [,wyr 2, ...wyrN])
```

Funkcja zwraca pozycję danej wartości w grupie wartości. W poniższym przykładzie sprawdzamy, jaką pozycję „od góry" zajmuje płaca o wysokości 1600 z prowizją 300. Dla równych sobie wartości funkcja zwraca tą samą pozycję.

*Przykład VII.5.4*

```
SELECT RANK(1600, 300) WITHIN GROUP (ORDER BY sal, comm Desc)
FROM emp;
```

### c) Funkcja analityczna RANK

```
RANK ( ) OVER ( [ klauzula PARTITION BY ] klauzula ORDER BY )
```

Funkcja działa podobnie jak Row_number, ale RANK to 1 + liczba rekordów poprzedzających w danej grupie , o mniejszej wartosci. Inaczej mówiąc, rekordy z tą samą wartością w klauzuli ORDER BY w tej samej grupie PARTITION BY otrzymają tą samą wartość RANK.

*Przykład VII.5.5*

*Dla każdego pracownika na stanowisku SALESMAN i MANAGER podaj „pozycję" płacową na jego stanowisku.*

```
SELECT ename,
       job,
       sal,
       RANK() OVER (PARTITION BY job ORDER BY sal Desc) Pozycja
FROM emp
WHERE job IN ('SALESMAN', 'MANAGER');
```

### d) Funkcja analityczna DENSE_RANK

```
DENSE_RANK() OVER ([klauzula PARTITION BY] klauzula ORDER BY)
```

Ta funkcja działa podobnie jak Rank, ale zwraca pozycję (liczba) wgrupie utworzonej według klauzuli PARTITION BY zgodnie z sortowaniem ORDER BY:

*Przykład VII.5.6*

```
SELECT
ename,
sal,
deptno,
DENSE_RANK() OVER (PARTITION BY deptno ORDER BY sal DESC)
FROM emp;
```

### e) Funkcja LISTTAGG

```
LISTAGG (kolumna [, 'separator']) WITHIN GROUP (klauzula ORDER
BY) [OVER klauzula PARTITION BY)]
```

*Przykład VII.5.7*

```
SELECT
LISTAGG (Rtrim(ename), ', ') WITHIN GROUP (ORDER BY ename) "Pracownicy dzialu 10"
FROM emp
WHERE deptno = 10;
```

```
SELECT
DISTINCT job, LISTAGG (Rtrim(ename), '; ') WITHIN GROUP
(ORDER BY ename) OVER (PARTITION BY job) "Pracownicy działów"
FROM emp;
```

## 6. MERGE Pl/SQL

Koncepcja i rola operacji MERGE w PL/SQL jest identyczna jak w przypadku T-SQL (rozdział IV.10) – synchronizacja danych w źródle danych i tabeli docelowej. Działanie w składni PL/SQL przedstawimy na przykładzie identycznym jak w T-SQL.

Składnia **MERGE** w PL/SQL:

```
MERGE INTO  nazwa_tabeli_docelowej
USING       instrukcja SELECT definiujaca źródło danych)
ON          warunek złączenia)
WHEN MATCHED THEN UPDATE SET kolumna = wyrażenie,(…)
                  [WHERE warunek]
                  [DELETE WHERE warunek]
WHEN NOT MATCHED THEN INSERT(nazwa_kolumny(,…))
                      VALUES(wyrażenie(,…))
                      [WHEREwarunek];
```

Rekordy w tabeli docelowej spełniające warunek powiązania zostaną zaktualizowane, rekordy ze źródła danych niespełniające warunku powiązania zostaną dopisane do tabeli docelowej. Rekordy z tabeli docelowej spełniające warunek powiązania oraz warunek **WHERE** dla **DELETE** zostaną z niej usunięte.

*Przykład VII.6.1*

*Tworzymy tabelę **Budzety_dzialow** w której będą zapisywane kwoty przeznaczone na płace pracowników poszczególnych działów. Operacja może być wykonywana cyklicznie (np. raz na*

*miesiąc), lub też po każdej zmianie danych mających wpływ na dane w tabeli* ***Budzety_dzialow****.*

*Tworząc procedurę, która będzie taką operację realizowała, musimy założyć, możliwość wystąpienia zmian zarówno w wysokości płac pracowników, zmiany zatrudnienia, a także zmiany w działach (tabela dept).*

*Zagadnienie to rozwiążemy tworząc instrukcję* ***MERGE****, porównującą stan danych w tabeli docelowej i źródle danych (tabeli, widoku) i na podstawie warunku złączenia identyfikującej rekordy „dopasowane" i „niedopasowane".*

```
CREATE TABLE BUDZETY_DZIALOW (
   DEPTNO NUMBER(2, 0),
   BUDZET NUMBER,
   Data_aktualizacji DATE,
   CONSTRAINT FK_Dept_budzet FOREIGN KEY (deptno) References Dept
);
```

```
MERGE INTO BUDZETY_DZIALOW BD
   USING (SELECT SUM(sal) SS, deptno, Sysdate
   FROM emp
   GROUP BY deptno, Sysdate) SBD
ON (NVL(SBD.deptno, -1) = NVL(BD.deptno, -1))
WHEN MATCHED THEN UPDATE SET BD.Budzet = SS
   DELETE WHERE BD.deptno IS NULL
WHEN NOT MATCHED THEN INSERT (Budzet, deptno, Data_aktualizacji)
                              VALUES (SS, SBD.deptno, Sysdate);
```

*Przed pierwszym uruchomieniem instrukcji tabela Budzety_dzialow jest pusta. Po uruchomieniu instrukcji MERGE instrukcja*

```
SELECT * FROM BUDZETY_DZIALOW;
```

*zwraca wynik uwzględniający rekord, w którym nr działu deptno jest NULL (jeden z pracowników nie jest przyp[iany do żadnego działu). Dopisujemy kolejny wiersz do tabeli emp*

```
INSERT INTO emp (empno, ename, sal, job, deptno)
VALUES (9876 , 'GREEN', 1250, 'SALESMAN', 40);
```

*i ponownie uruchamiamy instrukcję MERGE. Teraz pojawia się dział 40 (przypisaliśmy do niego nowego pracownika), zniknął natomiast dział „NULL" usunięty przez klauzulę DELETE w instrukcji MERGE. Gdyby po raz kolejny wykonać MERGE nie zmieniając wartości NULL w kolumnie deptno jednego z pracowników, dział „NULL" ponownie pojawi się w tabeli Budzety_dzialow, bo klauzula DELETE usuwa tylko rekordy w tej tabeli istniejące. Aby nie doprowadzić do swoistego oscylatora (znikające i ponownie pojawiające się rekordy), można uzupełnić całe polecenie dodatkową klauzulą*

```
... WHERE SBD.deptno IS NOT NULL;
```

## 7. Funkcje PL/SQL

Rola funkcji PL/SQL definiowanych przez użytkownika jest identyczna jak funkcji w T-SQL. Składnia też jest zbliżona (ale nie identyczna), natomiast PL/SQL implementuje tylko podstawowy wariant funkcji, o składni:

```
CREATE [OR REPLACE] FUNCTION nazwa_funkcji (lista parametrów)
RETURN typ_zwracany
```

```
IS
Deklaracja zmiennych (bez słowa DECLARE)
Blok PL/SQL
```

*Przykład VII.7.1*

*Utwórz funkcję zwracającą średnia płacę w dziale, którego numer (deptno) zostanie podany w parametrze funkcji.*

```
CREATE OR REPLACE FUNCTION Srednia_w_dziale (p_deptno IN Number)
RETURN Number
IS
   V_srednia Number(8,2);
BEGIN
   SELECT AVG(sal) INTO V_srednia
   FROM emp
   WHERE deptno = p_deptno;
RETURN (V_srednia);
END;

-- i jej wywołanie

SELECT srednia_w_dziale(20) FROM dummy;
```

*I jeszcze drugi sposób wykonania tej samej funkcji, tym razem w bloku Pl/SQL z użyciem zmiennej podstawienia (przykład zrealizowany w Sqldeveloper, w Dbeaver nie działa):*

```
SET Verify Off
SET serveroutput on
ACCEPT p_deptno PROMPT 'Podaj numer działu';

DECLARE
   v_avg Number(8,2);
   v_deptno Int := &p_deptno;
BEGIN
   v_avg := srednia_w_dziale(v_deptno);
   dbms_output.put_line('Srednia płaca w dziale ' || v_deptno || ' to ' || v_avg);
END;
```

## 8. Dynamiczny SQL

Uwagi dotyczące koncepcji oraz użycia dynamicznego SQL w środowisku DB Oracle są identyczne jak w przypadku MS SQL Server. Z jednej strony jest możliwość napisania elastycznego, łatwo modyfikowanego kodu, z drugiej niebezpieczeństwo podatności tego kodu na ataki SQL Injection. Oczywiście decyzja o stosowaniu (lub nie) należy do programisty i jest zależna od konkretnego przypadku.

ORACLE oferuje dwa podejścia do dynamicznego konstruowania kodu SQL. Pierwsze (starsze) to pakiet Pl/SQL o nazwie DBMS_SQL. Jest to rozwiązanie „nieco" skomplikowane i nie będzie omawiane w tym wykładzie. Zainteresowanych odsyłam do rozdziału 5 podręcznika „Systemy zarządzania bazami danych" autorstwa Lecha Banachowskiego i Krzysztofa Stencla, Wydawnictwo PJWSTK.

Rozwiązanie drugie, to naturalny dynamiczny SQL, którego instrukcje można bezpośrednio umieszczać w kodzie Pl/SQL. Składnia wygląda następująco:

```
EXECUTE IMMEDIATE instrukcja_SQL
```

```
[USING lista_parametrów_wiązania]
```

gdzie `instrukcja_SQL` to wyrażenie napisowe Varchar2 definiujące przeznaczoną do wykonania instrukcję SQL, opcjonalnie dostarczając do zmiennych wiązania w `instrukcja_SQL` wartości parametrów wiązania.

*Przykład VII.8.1*

*Korzystając z dynamicznego SQL utwórz nowy dział w tabeli dept, przenieś do nowo utworzonego działu pracownika o podanym w parametrze empno i wypisz stosowny komunikat o zmianie przynależności do działu.*

```
DECLARE
   sql_stmt Varchar2(100);
   plsql_block Varchar2(200);
   my_deptno INT := 60;
   my_dname Varchar2(30) := 'ACCOUNTS';
   my_loc Varchar2(30) := 'DETROIT';
   my_empno INT := 7788;
   emp_rec emp%rowtype;
BEGIN
sql_stmt := 'INSERT INTO Dept (deptno, dname, loc)
             VALUES (:1, :2, :3)';
EXECUTE IMMEDIATE sql_stmt
USING my_deptno, my_dname, my_loc;

sql_stmt := 'UPDATE emp
             SET deptno = :did
             WHERE empno = :eid';
EXECUTE IMMEDIATE sql_stmt
USING my_deptno, my_empno;

sql_stmt := 'SELECT * FROM emp WHERE empno = :eid';
EXECUTE IMMEDIATE sql_stmt INTO emp_rec
USING my_empno;

dbms_output.put_line('Pracownik ' || emp_rec.ename ||
                     ' pracuje teraz w dziale ' || emp_rec.deptno);
END;
```