	Bazy Danych laboratorium	**Laboratorium BD8**

Zagadnienie: Programowanie serwera bazodanowego Oracle przy pomocy języka PL/SQL

I. Język PL/SQL - informacje podstawowe

PL/SQL jest skrótem od "Procedural Language extension to SQL" (proceduralne rozszerzenie języka SQL) i jest standardowym językiem programowania dostępnym w relacyjnych bazach danych Oracle.

Udostępnia blokową strukturę wykonywalnych modułów kodu, dzięki czemu konserwacja kodu jest znacznie łatwiejsza. Do podstawowych obiektów PL/SQL należą:

- zmienne, stałe i typy,
- struktury sterujące, takie jak instrukcje warunkowe i pętle,
- moduły wielokrotnego użytku, takie jak funkcje, procedury, wyzwalacze i pakiety.

Do zalet języka PL/SQL należą:

- integrowanie konstrukcji proceduralnych z językiem SQL - język SQL nie posiada konstrukcji warunkowych ani pętli w pełnym tego słowa znaczeniu, więc przy pomocy PL/SQL jest możliwa realizacja bardziej skomplikowanych algorytmów,
- zwiększenie wydajności serwera, gdyż obiekty PL/SQL znajdują się bezpośrednio na serwerze bazodanowym i tam są wywoływane i wykonywane,
- obiekty programowe są na trwałe zapamiętane na serwerze bazodanowym i mogą być wielokrotnie wykonywane,
- umożliwia obsługę wyjątków (exception) występujących w trakcie wykonywania kodu,
- oprogramowanie utworzone w PL/SQL może być przenoszone między schematami (kontami), jak również między serwerami Oracle różnych wersji.

II. Typy obiektów PL/SQL i ich budowa

Do podstawowych bloków języka PL/SQL należą:

- bloki anonimowe,
- procedury,
- funkcje,
- wyzwalacze,
- pakiety programowe.

Bloki anonimowe

Są nietrwałymi obiektami programowymi czyli nie są zapamiętane w schemacie, mogą być zapamiętane w plikach tworzących skrypty wdrożeniowe (na przykład łącznie ze zdaniami SQL tworzącymi obiekty bazodanowe). Struktura takiego bloku wygląda tak:

```
DECLARE
        zmienne, kursory, wyjątki zdefiniowane przez użytkownika;
BEGIN
        zdania SQL - select, insert, update, delete;
        podstawowe instrukcje PL/SQL;
        wywołania innych obiektów PL/SQL, jak procedury czy funkcje;
EXCEPTION
        instrukcje wykonywane w przypadku wystąpienia wyjątku (błędu);
END;
```

Przykłady bloków anonimowych:

```
begin
  dbms_output.put_line ('PL/SQL');
end;
```

Sekcje DECLARE i EXCEPTION są opcjonalne. Obowiązkowe są instrukcje BEGIN i END stanowiące obszar dynamiczny kodu. Obszar ten nie może być pusty (*begin end;*), ale może zawierać *null* w swoim ciele (*begin null end;*).

Powyższy blok wyprowadza napis *PL/SQL* w środowisku SQL Developer lub SQL Plus.

Inny blok:

```
declare
   v_liczba_zawodnikow integer;
begin
  select count ( * )  into v_liczba_zawodnikow
  from bd3_zawodnicy;
  dbms_output.put_line ( 'Ewidencja zawiera ' || v_liczba_zawodnikow || ' zawodników' );
end;
```

jest przykładem zanurzenia zdania SQL w kod programowy.

Procedury

Są trwałymi obiektami programowymi i ich rolą jest wykonanie pewnych czynności w bazie, na przykład utworzenie raportu analitycznego, wprowadzenie nowego wiersza do tabeli lub skasowanie wierszy spełniających pewne kryterium. Mogą być sparametryzowane lub też nie.

Budowa procedury:

```
CREATE OR REPLACE PROCEDURE nazwa_procedury
                ( lista parametrów wejściowych i wyjściowych )
AS
        zmienne, kursory, wyjątki zdefiniowane przez użytkownika;
BEGIN
        zdania SQL - select, insert, update, delete;
        podstawowe instrukcje PL/SQL;
        wywołania innych obiektów PL/SQL, jak procedury czy funkcje;
EXCEPTION
        instrukcje wykonywane w przypadku wystąpienia wyjątku (błędu);
END;
```

Jak widać obszar dynamiczny między BEGIN i END jest dokładnie taki sam jak w bloku anonimowym oraz brak jest sekcji DECLARE.

```
create or replace procedure pr_insert_nowy_klub
                ( v_nr_klubu number,
                  v_nazwa_klubu varchar2)
as
begin
        insert into bd3_kluby ( nr_klubu, nazwa_klubu )
        values ( v_nr_klubu, v_nazwa_klubu );
        commit;
end;
```

Należy zauważyć, że definicje parametrów formalnych procedury nie zawierają precyzji czyli występuje *number*, a nie *number* (3). Analogicznie jest dla typu *varchar2*.

Deklaracji zmiennych lokalnych dokonuje się w obszarze między słowami kluczowymi *as* i *begin*.

Procedura uruchamiana jest przez swoją nazwę, na przykład:

```
declare
  v_nr_klubu integer default seq_nr_klubu.nextval;
  v_nazwa_klubu varchar2 ( 40 );
begin
  v_nazwa_klubu := 'Nowy Klub';
  pr_insert_nowy_klub (  v_nr_klubu, v_nazwa_klubu );
end;
```

Funkcje

Są trwałymi obiektami programowymi i ich podstawową rolą jest sprawdzanie stanu bazy danych oraz wykonywanie obliczeń lub znajdowanie w bazie danych konkretnych danych, na przykład sprawdzenie czy w ewidencji znajduje się zawodnik o określonych parametrami funkcji argumentach, obliczenie liczby punktów zdobytych przez danego zawodnika czy znalezienie w ewidencji najstarszego zawodnika. Mogą być sparametryzowane lub też nie.

Budowa funkcji:

```
CREATE OR REPLACE FUNCTION nazwa_funkcji
                ( lista parametrów wejściowych )
RETURN typ_funkcji
AS
        zmienne, kursory, wyjątki zdefiniowane przez użytkownika;
BEGIN
        zdania SQL - select, insert, update, delete;
        podstawowe instrukcje PL/SQL;
        wywołania innych obiektów PL/SQL, jak procedury czy funkcje;
RETURN zmienna_odpowiedniego_typu;
EXCEPTION
        instrukcje wykonywane w przypadku wystąpienia wyjątku (błędu);
END;
```

Funkcja musi być określonego typu zdefiniowanego w nagłówku definicji (RETURN ...) oraz w ciele funkcji musi, co najmniej raz, wystąpić instrukcja RETURN wraz ze zwracaną wartością.

Przykład funkcji:

```
create or replace function fn_najstarszy_zawodnik_klubu
                        ( v_nr_klubu number )
return varchar2
as
        v_nazwisko varchar2 ( 40 );
begin
        select nazwisko into v_nazwisko from bd3_zawodnicy
        where data_urodzenia = ( select min ( data_urodzenia )
                                from bd3_zawodnicy
                                where nr_klubu = v_nr_klubu );
return v_nazwisko;
end;
```

Funkcja zwraca wartość znakową i dlatego musi być typu varchar2.

Sposoby wywołania funkcji:

Pierwszy: w zdaniach SQL na tych samych zasadach jak funkcje standardowe języka SQL[1].

Zdanie:

```
select nr_klubu, fn_najstarszy_zawodnik_klubu ( nr_klubu ) "Nazwisko"
from bd3_kluby;
```

utworzy zbiór wyników:

NR_KLUBU	Nazwisko
1	Cisłak
2	Borowski
3	Werthein
4	Mańkowski
5	(null)
6	Dembiński
7	Kowalczyk

.....

Drugi: w bloku PL/SQL:

```
declare
        v_nr_klubu integer default :Nr_Klubu;
        v_nazwisko varchar2 ( 40 );
begin
        v_nazwisko := fn_najstarszy_zawodnik_klubu ( v_nr_klubu );
        dbms_output.put_line ( 'Najstarszym zawodnikiem w klubie ' || v_nr_klubu
                                || ' jest ' || v_nazwisko );
end;
```

Uruchomienie tego kodu z argumentem :Nr_Klubu = 1 da wynik:

```
Najstarszym zawodnikiem w klubie 1 jest Cisłak
```

Funkcja musi stać po prawej stronie instrukcji podstawienia i musi być takiego samego typu jak zmienna stojąca po lewej stronie.

Uwagi:

1. Bloki anonimowe nie są trwałym obiektem bazodanowym i ich "czas życia" jest zgodny z czasem aktywności sesji. Mogą znajdować się w historii pracy w SQL Developer lub być zapamiętanymi w plikach zewnętrznych.

2. Funkcje i procedury usuwane są ze schematu zdaniami:

   ```
   drop procedure nazwa_procedury;
   drop function nazwa_funkcji;
   ```

3. Przy definiowaniu zmiennych lokalnych lub argumentów funkcji i procedur należy przestrzegać zasady, aby ich nazwy nie były tożsame z nazwami kolumn w tabelach, gdyż może to prowadzić do otrzymywania niezamierzonych wyników.

[1] Język SQL nie posiada typu logicznego (boolean), dlatego nie jest możliwe uruchamianie w ten sposób funkcji logicznych czyli funkcji zwracających wartość boolean.

Rozważmy przykład:

```
declare
   nr_klubu integer;
   v_nazwa_klubu varchar2 ( 40 );
begin
   nr_klubu := 1;
   select nazwa_klubu into v_nazwa_klubu from bd3_kluby
   where nr_klubu = nr_klubu;
   dbms_output.put_line (v_nazwa_klubu);
end;
```

Próba uruchomienia powyższego kodu zakończy się błędem:

```
Error report -
ORA-01422: dokładne pobranie zwraca większą liczbę wierszy niż zamówiono
```

Warunek zawarty we frazie *where...* zdania *select...* jest warunkiem typu 1 = 1, gdyż nazwy kolumn tabeli mają priorytet nad nazwami zmiennych.

III. Definiowanie zmiennych i nazw obiektów programowych

Konwencja nadawania nazw jest następująca:

- muszą zaczynać się od litery,
- mogą składać się z liter i cyfr,
- mogą zawierać znaki specjalne, takie jak:
  $, _ , #
- ich długość nie może przekraczać 30 znaków,
- nie mogą być słowami zarezerwowanymi (np. *select* czy *begin*).

Przyjęło się, że nazwy zmiennych lub argumentów procedur i funkcji poprzedza się prefixem "v_" (v od *variable* - zmienna), np: v_nazwisko, v_nr_klubu tak, aby ustrzec się konfliktu opisanego wyżej.

Można też spotkać konwencję:

- n_nr_klubu dla zmiennych numerycznych,
- c_nazwa_klubu lub v_nazwa_klubu dla zmiennych znakowych,
- d_data_urodzenia dla zmiennych typu data.

Inicjowanie zmiennych może odbywać się na trzy sposoby.
Pierwszy polega na inicjowaniu zmiennej konkretną wartością w momencie jej deklarowania, drugi w obszarze dynamicznym instrukcją podstawienia i trzeci poprzez użycie zmiennej wiązania zarówno w sekcji DECLARE, jak i w obszarze dynamicznym.

```
DECLARE
        v_data_urodzenia date;
        v_nr_klubu number ( 3 )  := 1;
        c_napis   constant  varchar2 ( 50 )  default  'Ewidencja zawodników';
        v_nazwa_klubu varchar2 ( 40 ) := :Nazwa_klubu;
BEGIN
        v_nazwa_klubu := 'Nowy_Klub';
        v_nr_klubu := :Nr_klubu;
                .....
END;
```

Warto zauważyć, że zmienna niezainicjowana ma nieokreśloną wartość czyli *null*, jak w powyższym przykładzie zmienna v_data_urodzenia.
Zmienna c_napis ma atrybut constant (stała) czyli w przetwarzaniu nie może stać z lewej strony instrukcji podstawienia (nie może być zmieniona jej wartość).

Można inicjować również parametry formalne procedur i funkcji. Na przykład:

```
create or replace procedure pr_insert_nowy_klub
                    ( v_nr_klubu number default seq_nr_klubu.nextval,
                      v_nazwa_klubu varchar2)
                   .....
```

lub

```
create or replace function fn_liczba_zawodnikow
                                ( v_plec varchar2 := 'M' )
                   .....
```

Sposób wywołania takiego kodu może uwzględniać wartości domyślne lub nie.

Na przykład dla powyżej zdefiniowanej procedury wywołanie jej w notacji pozycyjnej wygląda tak:

```
begin
  pr_insert_nowy_klub ( 50, 'Nowy Klub' );
end;
```

a dla notacji nominalnej:

```
begin
  pr_insert_nowy_klub ( v_nazwa_klubu =>  'Nowy Klub' ); -- numer klubu określi sekwencja
end;
```

Typy zmiennych

Podstawowymi typami zmiennych języka PL/SQL są:

- *number* dla zmiennych numerycznych,
- *varchar2* dla zmiennych znakowych,
- *date* i *timestamp* dla dat i czasu,
- *boolean* dla zmiennych logicznych.

Lista możliwych typów jest dłuższa. Poza typami skalarnymi istnieje możliwość definiowania typów złożonych, takich jak rekordy czy tablice, jak również typy dotyczące dużych obiektów (takich jak zdjęcia, pliki multimedialne). Zagadnienia te nie będą w tym dokumencie omawiane.
Dodatkowo w stosunku do typów numerycznych można stosować znane z innych języków programowania typy, takie jak *integer*, *real* czy *float*.

Atrybut %TYPE

Atrybut ten służy do deklarowania zmiennej zgodnie z definicją kolumny w tabeli lub inną zadeklarowaną zmienną.

```
DECLARE
  v_nazwisko          bd3_zawodnicy.nazwisko%type;
  v_liczba_kobiet     integer;
  v_liczba_mezczyzn   v_liczba_kobiet%type;
```

Można go również używać do definiowania argumentów procedur i funkcji, na przykład:

```
create or replace function fn_liczba_zawodnikow
                                ( v_plec bd3_zawodnicy.plec%type  := 'M' )
```

Ten sposób deklarowania zmiennych zwiększa stabilność oprogramowania, gdyż typy zmiennych są dynamicznie dziedziczone w momencie uruchamiania kodu. Oprogramowanie jest mniej narażone na błędy fatalne, które mogą wystąpić na skutek zmiany struktur tabel bazodanowych.

Zmienne logiczne nie występują w języku SQL (nie ma kolumny typu *boolean*), a w języku PL/SQL - tak. Można im przypisać wyłącznie wartości TRUE, FALSE i NULL.
Do zwracania wartości logicznych można używać wyrażeń arytmetycznych, znakowych i dat zbudowanych w postaci zdań logicznych.

```
DECLARE
  v_old number  := 400;
  v_new number := 500;
  flag boolean;
BEGIN
  flag := v_old < v_new;  -- wartość zmiennej flag zostanie ustawiona na TRUE
END;
```

Typ *boolean* może być również użyty do definiowania funkcji logicznych. Przykładowo chcemy opracować funkcję decyzyjną, która będzie odpowiadała na pytanie: “Czy zawodnik o zadanym numerze startował w jakichkolwiek zawodach?”.

Kod takiej funkcji może wyglądać tak:

```
create or replace function fn_czy_zawodnik_startowal
                    (v_nr_zawodnika bd3_wyniki.nr_zawodnika%type)
return boolean
as
if_exists integer;
how_many integer;
begin
   select count(*) into if_exists
          from bd3_zawodnicy
          where nr_zawodnika = v_nr_zawodnika;
    select count(*) into how_many
           from bd3_wyniki
          where nr_zawodnika = v_nr_zawodnika;

   case
     when if_exists = 0 then return null;
     when how_many = 0 then return false;
     when how_many > 0 then return true;
  end case;
end;
```

Uruchomienie takiej funkcji w zdaniu SQL nie jest możliwe z uwagi na brak typu *boolean* w implementacji tego języka. Natomiast jej wywołanie w kodzie PL/SQL typu procedura czy blok anonimowy jest możliwe I spełnia taką samą rolę jak zmienna logiczna użyta na przykład w funkcji sterującej *if* lub *case*.

Zdanie:

```
select fn_czy_zawodnik_startowal (v_nr_zawodnika => 333) from dual;
```

zgłosi błąd wykonania:

```
SQL Error: ORA-00902: niepoprawny typ danych
```

Natomiast można tej funkcji użyć do sterowania przetwarzaniem:

```
declare
v_nr_zaw bd3_wyniki.nr_zawodnika%type := :Nr_Zawodnika;
```

```
begin
if fn_czy_zawodnik_startowal (v_nr_zawodnika => v_nr_zaw) then
        /*  gdy TRUE */
        dbms_output.put_line ('Zawodnik o numerze '||v_nr_zaw||' startował w sezonie');

elsif not fn_czy_zawodnik_startowal (v_nr_zawodnika => v_nr_zaw) then
        /*  gdy FALSE */
        dbms_output.put_line ('Zawodnik o numerze '||v_nr_zaw||' nie startował w sezonie');

else    /*  gdy NULL */
        dbms_output.put_line ('Zawodnika o numerze '||v_nr_zaw||' brak w ewidencji');
end if;
end;
```

```
Zawodnik o numerze 50 nie startował w sezonie

Zawodnik o numerze 333 startował w sezonie

Zawodnika o numerze 1333 brak w ewidencji
```

IV. Podstawowe konstrukcje programowe w PL/SQL

Język PL/SQL jest językiem strukturalnym zawierającym szereg, znanych z innych języków programowania, konstrukcji programowych, takich jak instrukcje warunkowe i pętle. I z tego względu nie będą one dokładnie omawiane w tym rozdziale. Zostaną podane tylko przykłady ich zastosowania w odniesieniu do baz danych. Więcej informacji można będzie znaleźć w dokumentacji Oracle lub na rekomendowanym wcześniej serwisie Tech on the Net.

Instrukcja warunkowa if ... then ..... else .... end if;

Steruje wykonywaniem się odpowiednich partii kodu programowego w zależności od badanego warunku.
Na przykład:

```
.....................
select count ( * )  into v_liczba_zawodnikow
from bd3_zawodnicy
where plec = 'K' and nr_klubu = v_nr_klubu;

if v_liczba_zawodnikow = 0 then
   dbms_output.put_line ( ' W klubie o numerze ' || v_nr_klubu || ' brak jest zawodników ');
else
   dbms_output.put_line ( ' W klubie o numerze ' || v_nr_klubu || ' jest '
                                | | v_liczba_zawodnikow || ' zawodników ' );
end if;
.........................
```

Innymi odmianami tej konstrukcji są:

```
if ..... then ..... end if;
if ..... then ..... elsif ..... then ..... end if;
if ..... then ..... elsif ..... then ..... else ..... end if;
```

Należy zwrócić uwagę na zachowanie się instrukcji warunkowej przy obsłudze wartości *null*.

Przykład pierwszy:

```
.....
x := 5;
y := NULL;
...
IF x <> y THEN  -- wartością nie jest TRUE, lecz NULL
……. --  ta część algorytmu nie zostanie wykonana
END IF;
```

Przykład drugi:

```
.....
a := NULL;
b := NULL;
...
IF a = b THEN  -- wartością nie jest TRUE, lecz NULL
 ……. --  ta część algorytmu nie zostanie wykonana
ELSE
……. --  ta część algorytmu zostanie wykonana
END IF;
```

Wyrażenie i instrukcja *case*

W języku PL/SQL konstrukcja *case* występuje w dwóch postaciach, jako wyrażenie *case* i jako instrukcja *case*.

Wyrażenie *case* ma dwie postacie.

Pierwsza postać:

```
declare
identyfikator number ( 3 )   := upper ( :zmienna_wejsciowa);
v_zmienna_wyjsciowa varchar2 (40);

begin

v_zmienna_wyjsciowa :=
        case identyfikator                      -- identyfikator badany jest na równość z wartościami
                when 200  then ' Wspaniale'     -- zawartymi we frazach when
                when 100  then ' Bardzo dobrze '
                when   50  then ' Wystarczająco '
                else  ' Brak danych '           -- jeśli brak frazy else i żaden z powyższych
                                                -- warunków nie jest spełniony - case zwraca null
        end;

dbms_output.put_line ( v_zmienna_wyjsciowa );
end;
```

Druga postać:

```
declare
v_suma_pkt integer;
v_medal varchar2(40);

begin

select sum (  nvl (punkty_globalne, 0 )) into v_suma_pkt
from bd3_wynikiwhere nr_zawodnika = :Podaj_numer_zawodnika; -- suma punktów dla zawodnika
                                                            niestartującego ani razu jest null
```

```
v_medal :=
        case
                when v_suma_pkt = 0      then ' Brak medalu '  -- zawodnik startował i nie zdobył
                                                                  punktów
                when v_suma_pkt < 100    then ' Medal pocieszenia '
                when v_suma_pkt < 130    then ' Brązowy medal '
                when v_suma_pkt < 160    then ' Srebrny medal '
                when v_suma_pkt >= 160 then ' Złoty medal '
                else  ' Zawodnik nie startował w sezonie '         -- zawodnik ani razu nie startował
        end;

dbms_output.put_line ( v_medal );
end;
```

W każdej frazie *when* może wystąpić dowolne zdanie logiczne (niekoniecznie zawierające tę samą zmienną). Pierwsze zdanie logiczne dające wartość TRUE kończy przeszukiwanie konstrukcji *case*. Jeśli żaden warunek nie jest spełniony zwracana jest wartość frazy *else*, a jeśli jej nie ma - wartość *null*.

Natomiast instrukcja *case* realizuje różne warianty algorytmu (wykonuje instrukcje) w zależności od wartości zmiennej stanowiącej selector:

```
.....
case v_action
        when  ' DEL '      then delete bd3_kluby_kopia;
        when  ' COUNT ' then select count ( * ) into v_liczba from bd3_zawodnicy;
        when  ' SUM '     then .....................
        .....................
end case;
.....
```

Pętle sterujące

Pętle to konstrukcje, które wielokrotnie powtarzają wykonywanie instrukcji lub sekwencji instrukcji.
Są trzy rodzaje pętli:

- pętla podstawowa
- pętla *while*
- pętla *for*

Pętla podstawowa *loop*:

```
loop
   {przetwarzane instrukcje PL/SQL i SQL}
   exit {warunek wyjścia z pętli}
end loop;
```

Warunek wyjścia z pętli musi być tak oprogramowany, aby dawał możliwość wyjścia z niej po skończonej liczbie iteracji.
W przypadku braku frazy *exit* lub nie spełnienia powyższego postulatu pętla będzie nieskończona.

Przykład:

```
.....
counter integer default 1;
max_iteracja integer := :Ile_razy;

begin
.....
loop
```

```
        insert into tabela_wartosci (numer_wiersza)
        values (counter * 100);
        exit when counter = max_iteracja;
        counter := counter + 1;
end loop;
.....
end;
```

Tak zbudowana pętla wykona się co najmniej raz.

Pętla *while*:

```
while {warunek wyjścia z pętli}
loop
   {przetwarzane instrukcje PL/SQL i SQL}
end loop;
```

Przykład:

```
.....
counter integer default 1;
max_iteracja integer := :Ile_razy;

begin
.....
while counter <= max_iteracja loop
        insert into tabela_wartosci (numer_wiersza)
        values (counter * 100);
        counter := counter + 1;
end loop;
.....
end;
```

Warunek wyjścia z pętli umieszczony jest przy wejściu do kolejnej iteracji i dlatego tak zbudowana pętla może nie wykonać się ani razu.

W pętlach *loop* i *while* nie jest znana liczba iteracji przed rozpoczęciem jej wykonywania, gdyż warunek wyjścia może być dynamicznie zmieniany w trakcie iteracji.

Pętla *for*:

```
for counter in low_value..high_value
loop
        {przetwarzane instrukcje PL/SQL i SQL}
end loop;
```

Przykład:

```
.....
max_iteracja integer = :Ile_razy;
begin
.....
for counter in 1 .. max_iteracja loop
        insert into tabela_wartosci (numer_wiersza)
        values (counter * 100);
end loop;
.....
```

Liczba iteracji określona jest przy wejściu do pętli na podstawie zmiennych low_value i high_value. Licznik pętli (counter) nie musi być definiowany i nie może być miejscem docelowym żadnej instrukcji podstawienia.

V. Wykonywanie zdań języka SQL w PL/SQL

Zdanie *select:*

Budowa zdania *select* zanurzonego w PL/SQL rozszerzona jest o frazę *into* określającą zmienną lub zmienne, w których będą umieszczone dane pobrane z bazy.

```
select kolumna1, kolumna2, ......
   into zmienna1, zmienna2, ......
from {tabela lub tabele}
where .....
having ....
```

Przykład:

```
create or replace procedure pr_liczba_zawodnikow
          (v_nr_klubu bd3_zawodnicy.nr_klubu%type)
as
v_nazwa_klubu bd3_kluby.nazwa_klubu%type;
v_licznosc integer;

begin
  select nazwa_klubu, count ( * )
     into v_nazwa_klubu, v_licznosc
  from bd3_zawodnicy z join bd3_kluby k on z.nr_klubu = k.nr_klubu
  where z.nr_klubu = v_nr_klubu
  group by nazwa_klubu;

  dbms_output.put_line ('Klub '|| v_nazwa_klubu ||' ma zarejestrowanych ' ||
                                  v_licznosc || ' zawodników' );
end;
```

Wywołanie tej procedury:

```
begin
   pr_liczba_zawodnikow ( 3 );
end;
```

da wynik:

```
Klub KB Gymnasion Warszawa ma zarejestrowanych 104 zawodników
```

Zmienne użyte we frazie *into* są typu skalarnego (liczby, napisy czy daty) i dlatego zdanie *select* musi być tak zbudowane, aby zwracało wartości skalarne a nie zbiory. Innymi słowy warunek we frazie *where* musi określać dokładnie jeden wiersz.

Aby się o tym przekonać wystarczy w ciele procedury zmienić filtr we frazie *where* na

```
.....
where z.nr_klubu > v_nr_klubu
.....
```

i ponownie ją wywołać. Otrzymamy błąd uruchomienia:

```
Error report -
ORA-01422: dokładne pobranie zwraca większą liczbę wierszy niż zamówiono
```

Zdanie *select* zwraca więcej niż jeden wiersz i nie można tych zbiorów umieścić w zmiennych skalarnych.

Podobnie, jeśli zdanie *select* nie zwraca żadnego wiersza (na przykład podany numer klubu nie istnieje), wtedy także zwracany jest błąd:

```
begin
   pr_liczba_zawodnikow ( 0 );
end;
```

```
Error report -
ORA-01403: nie znaleziono danych
```

Efektem wystąpienia tych błędów jest przerwanie się wykonania kodu w aplikacji, co jest zjawiskiem niepożądanym. Aby temu przeciwdziałać należy te błędy wychwytywać i odpowiednio obsługiwać. Realizuje się to w sekcji EXCEPTION kodu PL/SQL, co nie będzie omówione w tym materiale.

Zdania DML (*insert*, *update*, *delete*):

W celu zmiany stanu bazy danych poprzez wprowadzanie danych do tabel, ich modyfikacje lub kasowanie używane są zdania SQL: *insert*, *update* i *delete*. Mogą być używane bezpośrednio w skryptach SQL lub w kodzie PL/SQL. Istotne jest to, że w przeciwieństwie do zdania *select* (które w PL/SQL zmienia swoją postać) zdania DML mają taką samą budowę.
Na przykład chcąc wprowadzić dane o nowych zawodach do tabeli BD3_ZAWODY można napisać procedurę:

```
create or replace procedure pr_insert_nowe_zawody
            (v_nr_zawodow numeric,
             v_nazwa_zawodow varchar2,
             v_data_zawodow date,
             v_podtytul varchar2 default null)
as
begin
          insert into bd3_zawody (nr_zawodow, nazwa_zawodow, data_zawodow, podtytul)
          values (v_nr_zawodow, v_nazwa_zawodow, v_data_zawodow, v_podtytul);
          commit;
end;
```

Wywołując ją blokiem anonimowym:

```
begin
          pr_insert_nowe_zawody (5, 'Puchar Warszawy', '2018/11/10');
          pr_insert_nowe_zawody (6, 'Puchar Warszawy', '2018/11/11', 'Bieg Niepodległości');
end;
/
select * from bd3_zawody;
```

otrzymujemy wynik:

```
NR_ZAWODOW NAZWA_ZAWODOW                          DATA_ZAW PODTYTUL
---------- ---------------------------------------- -------- --------------------
         5 Puchar Warszawy                          18/11/10
         6 Puchar Warszawy                          18/11/11 Bieg Niepodległości
         1 Cztery Pory Roku - Jesień                10/10/22
         2 Cztery Pory Roku - Zima                  11/02/19
         3 Cztery Pory Roku - Wiosna                11/05/21
         4 Cztery Pory Roku - Lato                  11/09/03 im.S.Dankowskiego

6 rows selected.
```

Uwagi:

1. Znak "/" między kodem PL/SQL i zdaniem SQL nakazuje systemowi zarządzania baza danych przełączyć się z kontekstu PL/SQL na kontekst SQL lub odwrotnie.

2. Należy zwrócić uwagę na atrybut formalny v_podtytul w specyfikacji procedury. Ponieważ została ustawiona wartość domyślna (w tym przypadku *null*) wywołanie tej procedury może mieć dwojaką postać: z wprowadzeniem podtytułu lub nie, co widać w powyższym bloku uruchamiającym tę procedurę dwa razy. Zapoznając się ze strukturą tabeli BD3_ZAWODY można stwierdzić, że kolumna PODTYTUL może zawierać wartości *null* i dlatego taka specyfikacja procedury jest możliwa. Zmiana specyfikacji nagłówka tej procedury:

   ```
   create or replace procedure pr_insert_nowe_zawody
     (v_nr_zawodow number,
      v_nazwa_zawodow varchar2 default null,
      v_data_zawodow date,
      v_podtytul varchar2 default null)
   ```

   oraz próba uruchomienia jej:

   ```
   begin
           pr_insert_nowe_zawody (v_nr_zawodow   => 7,
                                  v_data_zawodow =>  '2018/11/20');
   end;
   ```

   zakończy się błędem, gdyż kolumna NAZWA_ZAWODOW w tabeli nie może przyjmować wartości *null.*

3. Usuwając wartości domyślne w specyfikacji procedury:

   ```
   create or replace procedure pr_insert_nowe_zawody
     (v_nr_zawodow number,
      v_nazwa_zawodow varchar2,
      v_data_zawodow date,
      v_podtytul varchar2 )
   ```

   i wywołując ją tak jak poprzednio:

   ```
   begin
           pr_insert_nowe_zawody (7, 'Puchar Warszawy', '2018/11/20');
   end;
   ```

   otrzymamy błąd niepoprawnej liczby argumentów, mimo, że kolumna PODTYTUL może przyjmować wartości *null*.

Jak wcześniej było napisane wykonanie zdanie *select* w kodzie PL/SQL może spowodować przerwane wykonywania kodu PL/SQL (a więc aplikacji), w przypadku gdy zbiór wynikowy zdania *select* jest zbiorem pustym.

Weźmy pod uwagę zdanie *select* wykonane interaktywnie:

```
select nazwisko, imie
from bd3_zawodnicy
where nr_zawodnika = 1000;
```

Zakładając, że nie ma zawodnika o takim numerze otrzymamy odpowiedź:

```
no rows selected
```

lub

NAZWISKO	IMIE

w zależności od sposobu uruchomienia, ale brak jest błędu.

Jeśli tak skonstruowane zdanie umieszczone zostanie w kodzie PL/SQL:

```
declare
v_nazwisko bd3_zawodnicy.nazwisko%type;
v_imie bd3_zawodnicy.imie%type;
begin
        select nazwisko, imie
        into v_nazwisko, v_imie
        from bd3_zawodnicy
        where nr_zawodnika = 1000;
dbms_output.put_line ( 'Zdanie zostało wykonane' );
end;
```

i wykonane, to kod "załamie się" w zdaniu *select* i dalsze przetwarzanie (wyświetlenie komunikatu) zostanie przerwane.

W przypadku użycia zdań DML w kodzie PL/SQL sytuacja jest inna.

Załóżmy, że chcemy skasować wszystkie wyniki zawodów o określonym numerze, którego brak w ewidencji zawodów.

Wykonując blok:

```
begin
        delete bd3_wyniki
        where nr_zawodow = 10;
dbms_output.put_line ( 'Żądanie kasowania wyników zostało wykonane' );
end;
```

i zakładając, że danych o zawodach o numerze 10 nie ma otrzymamy wynik:

```
Żądanie kasowania wyników zostało wykonane
```

, co oznacza, że zdanie DML (w tym przypadku *delete*) nie zostało przerwane, mimo, że żaden wiersz w tabeli BD3_WYNIKI nie spełniał kryterium kasowania.

Podobnie jest ze zdaniem *update*. Jeśli żaden wiersz nie spełnia kryteriów modyfikacji stan bazy się nie zmieni, a działanie aplikacji nie zostanie przerwane.

Do analizy liczby wierszy poddanych działaniu zdań *delete* lub *update* pomocnym może być niejawny kursor SQL%ROWCOUNT. Z każdym zdaniem DML wykonywanym przez serwer skojarzony jest indywidualny kursor zarządzany przez kod PL/SQL. Jego wartość ustawiana jest po każdym wykonaniu zdania *select*, *insert*, *update* i *delete* i określa liczbę wierszy objętych ostatnio wykonanym zdaniem SQL.

Na przykład poniższy kod:

```
begin
        delete bd3_wyniki
        where nr_zawodow = 10;
dbms_output.put_line ( 'Skasowano '|| sql%rowcount || ' wierszy' );
end;
```

po uruchomieniu wyświetli komunikat:

```
Skasowano 0 wierszy
```

a zmieniając numer zawodów na istniejący, np. 1 wynik będzie inny:

```
Skasowano 274 wierszy
```

Należy mieć świadomość, że z tabeli wyników zostały usunięte wiersze, ale nie w sposób trwały. Chcąc wycofać skutki działania tego kodu należy wycofać transakcję zdaniem SQL *rollback* według scenariusza:

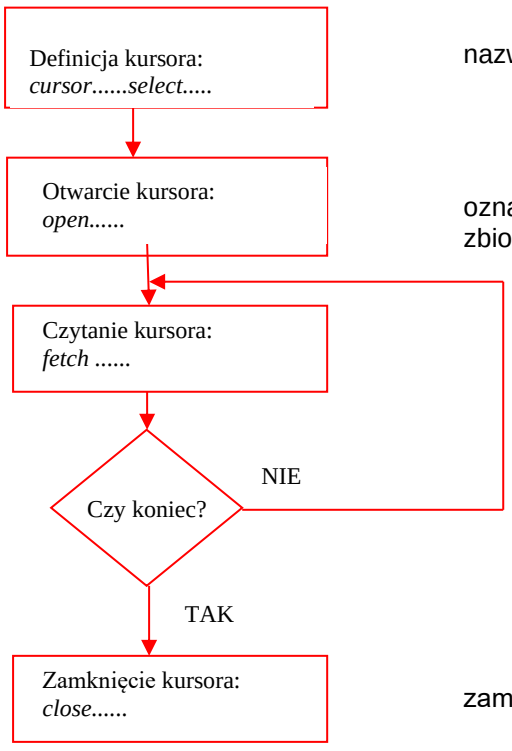
```
select count ( * ) from bd3_wyniki
where nr_zawodow =1;          -- sprawdzenie liczby wierszy przed wycofaniem
                              -- transakcji (zbiór jest pusty)

rollback;                     -- wycofanie transakcji

select count ( * ) from bd3_wyniki   -- sprawdzenie liczby wierszy po wycofaniu
where nr_zawodow =1;                 -- transakcji (zbiór liczy 274 wiersze)
```

VI. Zastosowanie kursorów w PL/SQL

Kursorem w języku PL/SQL określana jest technika polegająca na wykonaniu zdania *select* i utworzeniu zbioru wynikowego, a następnie na możliwości czytania i przetwarzania wiersz po wierszu tego zbioru.



nazwany dowolny *select* sparametryzowany lub nie

oznacza wykonanie *select* na serwerze i utworzenie zbioru wynikowego

czytanie i przetwarzanie kolejnego wiersza zbioru wynikowego pętlą *loop*

badanie przy pomocy *%notfound* końca czytanego zbioru wynikowego

zamknięcie kursora i zwolnienie obszaru na serwerze

W ujęciu klasycznym użycie kursora obejmuje następujące fazy:

- zdefiniowanie kursora zdaniem *select*,
- otwarcie kursora instrukcją *open*,
- czytanie kursora instrukcją *fetch*,
- sprawdzanie końca zbioru wynikowego funkcją *%notfound*,
- zamknięcie kursora instrukcją *close*.

Przykład zastosowania kursora niesparametryzowanego:

```
declare
v_nazwisko bd3_zawodnicy.nazwisko%type;
v_data_urodzenia bd3_zawodnicy.data_urodzenia%type;
cursor cur_zawodnik is
        select nazwisko, data_urodzenia
        from bd3_zawodnicy
         order by data_urodzenia desc;

begin
open cur_zawodnik;
loop
        fetch cur_zawodnik into v_nazwisko, v_data_urodzenia;
        exit when cur_zawodnik%notfound;
        dbms_output.put_line ( v_nazwisko || '      ' ||
                to_char ( v_data_urodzenia, 'DD.MM.YYYY' ) );
end loop;
close cur_zawodnik;
end;
```

Procedura drukowania w powyższym przykładzie symbolizuje przetwarzanie otrzymanych kursorem danych. Dane te można przetwarzać na różny sposób w zależności od potrzeb biznesowych.

Tak zdefiniowany kursor ma charakter statyczny, gdyż za każdym razem przy jego uruchomieniu zbiór wynikowy będzie taki sam.

Definicję kursora można sparametryzować przy pomocy zmiennej wiązania albo poprzez atrybut procedury.

Na przykład:

```
create or replace procedure pr_zawodnicy_punkty
          (v_nr_klubu bd3_zawodnicy.nr_klubu %type)
as
v_zawodnik varchar2(100);
v_suma integer;
cursor cur_agregat is
   select nazwisko || '  ' || imie zawodnik, sum ( nvl(punkty_globalne, 0 )) suma
   from bd3_zawodnicy z join bd3_wyniki w
      on z.nr_zawodnika = w.nr_zawodnika
   where z.nr_klubu = v_nr_klubu
   group by nazwisko || '  ' || imie
   order by suma desc;

begin
open cur_agregat;
dbms_output.put_line ('Osiągnięcia zawodników');
dbms_output.put_line ('====================');
dbms_output.put_line ( CHR ( 10 ));
loop
   fetch cur_agregat into v_zawodnik, v_suma;
```

```
    exit when cur_agregat%notfound;
    dbms_output.put_line (v_zawodnik || ' zdobył '|| v_suma || ' pkt');
end loop;
dbms_output.put_line ('==============================');
close cur_agregat;
end;
```

Definicja kursora jest dynamiczna, gdyż numer klubu określany jest w momencie uruchomienia procedury, na przykład blokiem:

```
begin
  pr_zawodnicy_punkty ( :nr_klubu );
end;
```

lub w ciele innej procedury.

Kolejnym sposobem przetwarzania danych przy pomocy kursora jest *kursorowa pętla for*, budowa której oparta jest o klasyczną pętlę *for*. Występuje w dwóch wariantach.

Przykład pierwszego wariantu:

```
declare
cursor cur_klub is
   select nr_klubu, nazwa_klubu
   from bd3_kluby
   order by nazwa_klubu;
begin
   for tmpRec in cur_klub
   loop
   dbms_output.put_line ( 'Klub ' || tmpRec.nazwa_klubu ||
                          ' ma numer ' || tmpRec.nr_klubu);
   end loop;
end;
```

Tak skonstruowana metoda użycia nie wymaga otwierania, zamykania ani badania końca zbioru kursora. Należy w sekcji DECLARE zdefiniować sparametryzowany lub nie kursor, a w części dynamicznej zastosować pętlę *for*.
Zmienna *tmpRec* nie wymaga wcześniejszego definiowania i jej struktura (skalarna lub rekordowa) jest taka, aby można było w niej umieścić to co zwraca kursor przy jednym obrocie pętli, czyli w powyższym przykładzie numer i nazwę klubu.

Przykład drugiego wariantu:

```
begin
   for tmpRec in ( select nazwisko, sum ( punkty_globalne ) suma
                   from bd3_zawodnicy z join bd3_wyniki w
                   on z.nr_zawodnika = w.nr_zawodnika
                   group by nazwisko
                   having sum ( punkty_globalne ) >= :Próg_punktowy
                   order by suma desc )
   loop
   dbms_output.put_line ( 'Zawodnik ' || tmpRec.nazwisko ||
                          ' zdobył ' || tmpRec.suma || ' pkt');
   end loop;
end;
```

Ta metoda dodatkowo nie wymaga definiowania kursora. Zdanie *select* jest użyte bezpośrednio w konstrukcji pętli *for*.

Zadania do samodzielnego wykonania

1. Opracować procedurę wyświetlającą datę zawodów oraz liczbę startujących w nich zawodników. Argumentem wejściowym jest data zawodów. W przypadku, gdy w zadanym dniu nie było zawodów należy wyświetlić komunikat o braku zawodów w tym dniu.

2. Opracować procedurę, która wpisuje do tabeli klubów nowy klub. Argumentem wejściowym ma być nazwa klubu. Wartość numeru klubu obliczać przy pomocy funkcji max (nr_klubu) + 1. Procedura ma sprawdzać, czy taka nazwa klubu już istnieje w tabeli. Jeśli tak należy wyświetlić stosowny komunikat.

3. Opracować procedurę, która wyświetla zadaną liczbę najlepszych zawodników w danych zawodach w następujący sposób: Nazwisko, Nazwa Klubu, czas_min:czas_sek. Argumentami wejściowymi ma być numer zawodów oraz liczba wyświetlanych zawodników.

4. Opracować funkcję, która na wejściu posiada atrybuty: numer klubu oraz Action mogący przyjmować następujące wartości: COUNT, SUMA, AVG w postaci napisu.
   W zależności od tego atrybutu funkcja będzie zwracała:
   - COUNT - liczbę aktywnych zawodników czyli tych, którzy startowali w sezonie chociaż raz,
   - SUMA - sumę zdobytych punktów przez zawodników tego klubu,
   - AVG - średnią wieku zawodników tego klubu.

   Do tego celu opracować odpowiednią perspektywę lub perspektywy i jej / ich używać w ciele funkcji. Użyć tej funkcji w kodzie programowym do wyświetlania wyników (wedle uznania).

5. Napisać procedurę, która przyjmuje jako dane wejściowe numer istniejącego klubu oraz nowy numer tego klubu. Zadaniem jej jest zmiana numeru klubu na nowy w ewidencji klubów, jak również zmiana numeru klubu w ewidencji zawodników.

6. Opracować funkcję logiczną, która stwierdza czy zawodnik o zadanym numerze startował w zawodach o określonym numerze. W przypadku nieistnienia zadanego numeru zawodnika lub numeru zawodów funkcja powinna zwracać *null*. Zaprezentować jej działanie.

7. Słownik kategorii wiekowych (BD3_KATEGORIE) zawiera przedziały czasowe w postaci roczników, które decydują o przynależności każdego zawodnika do odpowiedniej kategorii:

NR_KATEGORII	NAZWA_KATEGORII	DOLNY_PROG	GORNY_PROG
1	I	1998	2017
2	II	1988	1997
3	III	1978	1987
4	IV	1968	1977
5	V	1958	1967
.....			
11	XI	1	1927
20	K-I	1998	2017
21	K-II	1988	1997
22	K-III	1978	1987
23	K-IV	1968	1977
.....			

, przy czym nr_kategorii = 1 oznacza najmłodszą kategorię męską,
= 11 oznacza najstarszą kategorię męską,
= 20 oznacza najmłodszą kategorię żeńską,
= 30 oznacza najstarszą kategorię żeńską.

Po zakończeniu rocznego cyklu biegowego czyli na początku każdego roku kalendarzowego musi nastąpić modyfikacja przynależności zawodników do odpowiednich kategorii wiekowych. Odbywało się to według następującego scenariusza:

- Modyfikacja tabeli BD3_KATEGORIE polegająca na przesunięciu okna czasowego dla każdej kategorii o jeden rok, czyli przykładowo na początku roku 2018 dla kategorii o numerze 1 nowe wartości okna czasowego powinny wynosić 1999 .. 2018, a dla kategorii o numerze 2 – 1989 .. 1998 itd. Dla najstarszych kategorii obojga płci wartości w kolumnie *dolny_prog* mają charakter symboliczny i nie muszą podlegać modyfikacji.
- Dla wszystkich zawodników (ale osobno dla mężczyzn i osobno dla kobiet) następowało powtórne obliczenie przynależności do odpowiednich kategorii wiekowych przy pomocy zdań SQL:

```
update bd3_zawodnicy
   set nr_kategorii = (select nr_kategorii
                       from bd3_kategorie
                       where extract (year from data_urodzenia)
                             between dolny_prog and gorny_prog
                       and nr_kategorii between 1 and 11 )
where plec = 'M';

update bd3_zawodnicy
   set nr_kategorii = (select nr_kategorii
                       from bd3_kategorie
                       where extract (year from data_urodzenia)
                             between dolny_prog and gorny_prog
                       and nr_kategorii between 20 and 30 )
 where plec = 'K';
```

- Zatwierdzenie transakcji zdaniem *commit*;

Należy opracować procedurę, która zautomatyzuje ten proces oraz uniemożliwi powtórną modyfikację tabel BD3_KATEGORIE i BD3_ZAWODNICY w tym samym roku kalendarzowym. Ta druga funkcjonalność może być zapewniona przez sprawdzenie wartości w kolumnie *gorny_prog* dla kategorii o numerze 1. W tym polu zawsze będzie znajdował się bieżący rok, jeśli nastąpiło już przeliczenie na jego początku.

Dodatkowo można zautomatyzować wykonanie się tej procedury poprzez opracowanie zadania, które może być uruchamiane raz w roku (np. każdego 1 stycznia o godzinie 8:00). Tworzenie zadań (*jobs*) zostało opisane w Laboratorium BD9.