Bazy Danych

w wykładzie wykorzystano:

1. Lech Banachowski, SYSTEMY BAZ DANYCH, Polsko-Japońska Wyższa Szkoła Technik Komputerowych, Warszawa 2004, http://edu.pjwstk.edu.pl/wyklady/sbd/scb/index.html
2. materiały dydaktyczne przygotowane w ramach projektu
   Opracowanie programów nauczania na odległość na kierunku studiów wyższych – Informatyka
   http://wazniak.mimuw.edu.pl

Programowanie aplikacji baz danych

po stronie serwera - PL/SQL

Krzysztof Regulski
WIMiIP, KISiM,
regulski@agh.edu.pl
B5, pok. 408

PL/SQL

- PL/SQL — wbudowany język proceduralny programowania systemu Oracle, przeznaczony do tworzenia procedur składowanych
- Inne bazy danych zwykle mają odpowiedniki języka proceduralnego, np.: PL/pgSQL w PostgreSQL
- PL/SQL rozszerza standard języka relacyjnych baz danych SQL poprzez umożliwienie korzystania z takich konstrukcji, jak: pętle, instrukcje IF-THEN czy złożone struktury danych.
- Podobny język jest określony w Standardzie SQL:1999. Nosi on tam nazwę SQL/PSM.

PL/SQL

- Język nie jest przeznaczony do interakcji z użytkownikiem, stąd brak w nim wielu mechanizmów, obecnych w innych językach programowania, a pozwalających np. na pobieranie informacji od użytkownika czy też wyrafinowane formatowanie wyników, wypisywanych przez program na ekranie.
- W programie PL/SQL można umieszczać polecenia DML, natomiast nie jest możliwe bezpośrednie wykonanie w programie poleceń z grupy DDL (poleceń tworzenia nowych obiektów, np. relacji) i DCL (poleceń sterowania przebiegiem sesji).

PL/pgSQL

- W PostgreSQL program obsługi języka PL/pgSQL jest dołączany w dystrybucji jako współdzielona biblioteka. Aby zainstalować język PL/pgSQL dla bazy danych można skorzystać z polecenia CREATE LANGUAGE w psql i załadować współdzieloną bibliotekę lub wykorzystać skrypt pomocniczy wywołany z konta użytkownika postgres.

```
createlang plpgsql nazwa_bazy_danych
```

- Jest to język blokowo-strukturalny, podobny do języka Pascal lub C, z deklaracjami zmiennych i zakresami bloków. Każdy blok ma opcjonalną etykietę, może posiadać kilka deklaracji zmiennych i zamyka instrukcje tworzące blok pomiędzy słowami kluczowymi BEGIN oraz END.
- W języku PL/pgSQL wielkość liter nie ma znaczenia.

Korzyści ze stosowania PL/SQL

- Łatwość wykonania trudniejszych zadań niż w SQL.
- Zwiększenie wydajności w przypadku gdy aplikacja generuje wiele zapytań do bazy danych.
- Mechanizmy niedostępne w SQL (dawniej), np. zmiennych do przechowywania tymczasowych wyników pewnych operacji, stałych, różnorodnych struktur sterujących (np. pętli, warunków), procedur obsługi błędów, jakie mogą pojawić się przy dostępie do bazy danych czy podczas działania programu.
- PL/SQL jest w pełni przenaszalny pomiędzy wszystkimi platformami.
- Możliwość wykorzystania bogatej biblioteki predefiniowanych programów, które Oracle dostarcza w postaci zbioru pakietów. Np. pakiet UTL_FILE pozwala na wykonywanie wewnątrz programu PL/SQL operacji na plikach, pakiet DBMS_SQL umożliwia dynamiczną konstrukcję poleceń SQL w programie, itd.

Blok *PL/SQL*

- Podstawową jednostką logiczną w programie PL/SQL, odpowiednikiem konstrukcji bloku (i programu) występujących w językach programowania jest w PL/SQL *blok PL/SQL* nazywany też *blokiem anonimowym.* nie posiada nazwy i jest wykonywany natychmiast po utworzeniu.
- Ma on następującą składnię:

```
DECLARE
   deklaracje obiektów PL/SQL jak zmienne, stałe,
wyjątki, procedury, funkcje
BEGIN
   ciąg instrukcji do wykonania
EXCEPTION
   obsługa wyjątków (błędów)
END;
```

- Deklaracje i obsługa wyjątków są opcjonalne. Bloki mogą być zagnieżdżone. Jedynymi instrukcjami języka SQL, które mogą się pojawić w bloku PL/SQL, są instrukcje SELECT, INSERT, UPDATE, DELETE, COMMIT i ROLLBACK.

Typy danych

- w PL/SQL możemy użyć wszystkich tych typów danych, które są obecne w SQL,
- dodatkowo PL/SQL posiada swoje własne typy danych, m.in:
  - typ logiczny BOOLEAN, posiadający dwa zdefiniowane literały: TRUE (prawda) i FALSE (fałsz)
  - oraz typ RECORD, umożliwiający zdefiniowanie zmiennej rekordowej.

Zmienne

- Zmienna jest elementem, służącym do krótkotrwałego przechowywania danych wewnątrz bloku PL/SQL.
- Zmienna przed użyciem musi zostać zadeklarowana w sekcji DECLARE bloku.
- Deklaracja zmiennej wymaga podania nazwy zmiennej i określenia jej typu (również długości jeśli typ tego wymaga).
- Dobrym nawykiem jest tworzenie nazw zmiennych przez użycie przedrostka „v_” w nazwie. Taka konwencja nazewnicza znacznie poprawia czytelność programu.
- Wyróżniamy dwa rodzaje zmiennych: zmienne proste i złożone.
  - Zmienna prosta służy do przechowywania wartości podstawowych typów danych: liczb, ciągów znaków, dat i wartości logicznych.
  - Zmienna złożona posiada wewnętrzną strukturę. Przykładami zmiennych złożonych są tablice, rekordy i obiekty.

Zainicjalizowanie zmiennej

- Wartość zmiennej, która została zadeklarowana, jednak nie przeprowadzono jej zainicjalizowania, jest pusta (zmienna posiada wartość NULL).
- Inicjalizować zmienną można przez wykonanie operacji przypisania wartości do zmiennej przy wykorzystaniu operatora przypisania (:=) lub przez użycie słowa DEFAULT.
- Dla zainicjalizowanej zmiennej przy jej deklaracji można dodać słowo NOT NULL jeśli zależy nam na tym, aby zmienna nigdy nie miała wartości pustej.

Zmienna rekordowa

- Rekord jest strukturą, umożliwiającą przechowywanie powiązanych logicznie danych. Dane rekordu składowane są w polach, z których każde ma swoją własną nazwę i typ danych.
- Aby w języku PL/SQL zadeklarować zmienną rekordową, należy najpierw zdefiniować tzw. typ rekordowy. Definicję typu przeprowadzamy w sekcji deklaracji bloku.

```
DECLARE
     TYPE DanePracownika IS RECORD (
            nazwisko VARCHAR2(100),
            imie VARCHAR2(100));
     v_pracownik DanePracownika;
BEGIN
     v_pracownik.nazwisko := 'Kowalski';
     v_pracownik.imie := 'Jan';
```

Atrybuty %TYPE i %ROWTYPE

- Atrybut **%TYPE** – do deklarowania zmiennej prostej na podstawie typu atrybutu relacji bazy danych lub typu innej zmiennej – rozwiązanie stosuje się wszędzie tam, gdzie istnieje konieczność zadeklarowania zmiennej, która ma przechowywać dane pobrane z bazy danych.
- Atrybut **%ROWTYPE** – do deklarowania zmiennej rekordowej (wierszowej) w oparciu o schemat relacji, kursora lub typ innej zmiennej rekordowej – najczęściej atrybutu %ROWTYPE używa się, gdy potrzebujemy zmiennej rekordowej, która ma przechować cały rekord ze wskazanej relacji.

```
DECLARE
     v_nazwisko pracownicy.nazwisko%TYPE;
     v_nazwisko_szefa v_nazwisko%TYPE:='Kowalski';
     v_dane_pracownika pracownicy%ROWTYPE;
```

Rekordy odpowiadające wierszom z tabel

– Zmienne „wierszowe” (rekordowe), pozwalają zapamiętać cały wiersz pochodzący z tabeli. Dostęp do poszczególnych pól wiersza odbywa się tak, jak w przypadku rekordów, przez kropkę i podanie nazwy kolumny.

```
DECLARE
  rek_osob Emp%ROWTYPE; /* Typ wierszowy */
BEGIN
  SELECT * INTO rek_osob FROM Emp e WHERE e.Ename = 'SCOTT';
  rek_osob.Sal := 1.1*rek_osob.Sal;
  INSERT INTO Dziennik
  VALUES (rek_osob.Ename, rek_osob.Job, rek_osob.Sal, SYSDATE);
END;
/
```

– Zmiennej wierszowej nie można użyć bezpośrednio po słowie kluczowym VALUES w instrukcji INSERT INTO, to znaczy trzeba między nawiasami dokładnie wyspecyfikować wstawiane wartości jedna po drugiej.

Rekordy PL/SQL

- Rozszerzeniem typu wierszowego jest typ rekordowy – definiowany explicite ze swoimi polami w sekcji deklaracji. Ma on podobne własności i ograniczenia co typ wierszowy. Najpierw należy zdefiniować typ rekordowy, a następnie zadeklarować zmienne tego typu.
- Zdefiniujmy typ rekordu pracownika.

```
TYPE Typ_rek_prac IS RECORD
    (numer_prac NUMBER(4) NOT NULL,
     nazwisko VARCHAR2(40) NOT NULL,
     Zarobki NUMBER(8,2),
     Num_działu NUMBER(4));
rekord_prac Typ_rek_prac;
```

- Typy rekordowe mogą być zagnieżdżone. Dostęp do pól rekordu jest za pomocą notacji kropkowej. Na zmienną rekordową można przyporządkowywać wartość innej zmiennej rekordowej, ale tylko tego samego typu rekordowego. Można ich używać w klauzuli INTO, tak jak zmiennych wierszowych:

```
SELECT *
INTO rekord_prac
FROM Pracownicy p
WHERE p.Id_prac = 12;
```

- Nie można ani porównywać wartości typu rekordowego ani nie można zmiennej rekordowej użyć bezpośrednio po słowie kluczowym VALUES w instrukcji INSERT.

Stałe

- Stałą można rozumieć jako zmienną, która nie zmienia przypisanej jej podczas deklaracji wartości przez cały czas trwania programu.
- Stała może stać tylko po prawej stronie operacji przypisania – jakakolwiek próba zmiany wartości stałej powoduje przerwanie działania programu i wygenerowanie komunikatu o błędzie.
- Dobrym nawykiem jest tworzenie nazw stałych przez użycie przedrostka **c_** w nazwie. Taka konwencja nazewnicza znacznie poprawia czytelność programu.

```
DECLARE
    c_pi CONSTANT NUMBER(5,4):=3.1415;
    c_false CONSTANT BOOLEAN:=FALSE;
```

Deklaracje zmiennych i stałych

- Deklaracja zmiennej ma następującą postać:
  identyfikator typ_danych [NOT NULL] [:= *wyrażenie*];
  Opcjonalna część := *wyrażenie* umożliwia inicjalizację wartości zmiennej – w przeciwnym razie zmiennej jest przypisywane *NULL*.
- Deklaracja stałej ma następującą postać:
  identyfikator CONSTANT *typ_danych* [NOT NULL] [:= *wyrażenie*];
- przykłady:

```
zarobki NUMBER(7,2);
pi CONSTANT NUMBER(7,5) := 3.14159;
nazwisko VARCHAR2(25) := 'Kowalski';
data DATE := Sysdate;
zonaty BOOLEAN := False;
liczba_dzieci BINARY_INTEGER :=0;
```

- Do inicjalizacji wartości zmiennej zamiast operatora := można użyć klauzuli DEFAULT – używa się jej, gdy zmienna ma charakterystyczną wartość, np. `Data DATE DEFAULT Sysdate;`
- Zauważmy, że deklaracji zmiennych tego samego typu nie można łączyć razem jak w innych językach.
- Nie należy nadawać tej samej nazwy zmiennej co kolumnie w tabeli. Zmienne i stałe PL/SQL mogą występować w instrukcjach SQL, tak jak stałe SQL i kolumny.

SQL*Plus

- Wprowadzając blok PL/SQL przy użyciu programu SQL*Plus, należy:
  - » blok zakończyć kropką znajdującą się na osobnej linii - w celu wpisania go do bufora bieżącej instrukcji;
  - » bądź znakiem / znajdującym się na osobnej linii - w celu wpisania go do bufora bieżącej instrukcji i wykonania.
- Wykonanie instrukcji w buforze bieżącej instrukcji (instrukcji SQL lub bloku PL/SQL) realizujemy poprzez podanie znaku / na osobnej linii. Instrukcje samego SQL*Plus nie są wpisywane do bufora.

Przykład

- W bloku jest oprogramowana transakcja sprzedaży samochodu marki *Fiat*. Gdy w magazynie jest brak informacji o takiej marce samochodu, podnoszony jest błąd, który jest następnie obsługiwany w sekcji wyjątków poprzez wpisanie odpowiedniej informacji do tablicy dziennika błędów.
- Proszę zwrócić uwagę na zamieszczanie komentarzy w kodzie:
  - albo między nawiasami /* i */ albo
  - od dwóch kresek -- do końca bieżącej linii.

```
DECLARE
  ilość NUMBER(5);
BEGIN
  SELECT m.Stan INTO ilość FROM Magazyn m
  WHERE m.Produkt = 'Fiat';
  /* Gdy w kolumnie Produkt tabeli Magazyn nie ma
wartości 'Fiat' jest podnoszony wyjątek o nazwie
no_data_found. */
  IF ilość> 0 THEN
    UPDATE Magazyn SET Stan = Stan - 1
    WHERE Produkt = 'Fiat';
    INSERT INTO Zakupy
    VALUES ('Kupiono Fiata', Sysdate);
  ELSE
    INSERT INTO Zakupy
    VALUES ('Brak Fiatów', Sysdate);
  END IF;
  COMMIT;
EXCEPTION -- Początek sekcji wyjątków
WHEN no_data_found THEN
   INSERT INTO dziennik_błędów
   VALUES ('Nie znaleziono produktu FIAT');
END;
/
```

Interakcja z użytkownikiem

- PL/SQL jest nastawiony na przetwarzanie danych i jego możliwości w zakresie interakcji z użytkownikiem są nader skromne
- Jeśli zachodzi konieczność wczytania wartości do programu, można użyć tzw. **zmiennych podstawienia**.
- Jeśli program został zapisany w narzędziu SQL*Plus, przed wykonaniem programu narzędzie przegląda go w poszukiwaniu zmiennych podstawienia, jeśli je znajdzie, pyta użytkownika o wartości dla tych zmiennych.
- Podane przez użytkownika wartości zostają wstawione w miejsca zmiennych podstawienia i program zostaje wykonany.
- Zmienne podstawienia są zamieniane na wartości przed wykonaniem programu, a nie w trakcie. Stąd nie można ich użyć np. do pytania użytkownika co do przebiegu programu (np. do jego rozgałęzienia).

Zmienne podstawienia

- Jeśli istnieje konieczność wypisania komunikatu z programu PL/SQL, można do tego celu użyć procedury **PUT_LINE** z pakietu DBMS_OUTPUT.
- Parametrem procedury jest ciąg znaków, który ma zostać wyświetlony na konsoli.
- Aby komunikaty pojawiały się na konsoli, konieczne jest ustawienie w narzędziu SQL*Plus wartości zmiennej SETSERVEROUTPUT na ON.
- Należy jednak pamiętać, że komunikaty z programu, generowane przez wykonanie wspomnianej procedury, pojawią się na konsoli dopiero po wykonaniu całego programu, a nie w momencie wykonania linii zawierającej procedurę PUT_LINE.

```
SQL> SET SETSERVEROUTPUT ON

BEGIN
     DBMS_OUTPUT.put_line('Hello World');
END;
```

Wprowadzanie danych z klawiatury i wypisywanie wyników na ekran

- *zmienne podstawienia* SQL*Plus mogą wystąpić tylko w wyrażeniach i nigdy po lewej stronie w instrukcji przypisania.
- oznaczamy je: `&zmienna`
- Przy testowaniu i uruchamianiu aplikacji składających się z kodu PL/SQL wygodnie jest pobierać dane do testowania z klawiatury i wypisywać informacje o przebiegu obliczeń na ekran.
- Oprócz zmiennych deklarowanych w bloku PL/SQL mogą występować jeszcze zmienne z aplikacji korzystającej z bloku PL/SQL - poprzedza się je dwukropkiem (**:** `zmienna`) i nazywa się *zmiennymi wiązania*.

Przykład

- Program pobierający od użytkownika dwie wartości przy wykorzystaniu zmiennych podstawienia &liczba i &tekst.
- Wartości te zostają użyte do zainicjalizowania zmiennych `v_i` i `v_nazwa`. Zwróćmy uwagę, że zmienna podstawienia &tekst została ujęta w apostrofy.

```
DECLARE
     v_i NUMBER (3):=&liczba;
     v_nazwa VARCHAR2(50):='&tekst';
BEGIN
     dbms_output.put_line('Zmienna v_i:'||to_char(v_i));
     v_nazwa:=v_nazwa||'ABC';
     dbms_output.put_line(v_nazwa);
END;
```

Przykład

- Skrypt SQL*Plus obejmujący wprowadzanie przez użytkownika wartości do bloku PL/SQL - za pomocą zmiennej podstawienia `rocz_zarob`, a następnie wypisywanie wyniku na ekran - za pomocą procedury `DBMS_OUTPUT.Put_line`.
- Należy pamiętać, ze w celu uruchomienia skryptu należy go najpierw zapisać w pliku, a następnie wywołać go w SQL*Plus przy użyciu instrukcji *`start`*.

```
SET SERVEROUTPUT ON
ACCEPT rocz_zarob PROMPT 'Podaj roczne zarobki: '
DECLARE
mies NUMBER(9,2) := &rocz_zarob;
BEGIN
  mies := mies/12;
  DBMS_OUTPUT.PUT_LINE ('Miesięczne zarobki = ' ||mies);
END;
/
```

– Nie możemy użyć tego skryptu w iSQL*Plus, bo nie obsługuje on instrukcji `ACCEPT`. Na szczęście opuszczenie instrukcji `ACCEPT` też prowadzi do satysfakcjonującego rezultatu. Dlaczego?

– Zamiast zmiennych podstawienia można użyć zmiennych wiązania z SQL*Plus, np.

```
ACCEPT rocz_zarob PROMPT 'Podaj roczne zarobki: '
VARIABLE mies NUMBER
BEGIN
  :mies := &rocz_zarob/12;
END;
/
PRINT Mies
```

Zmienne systemowe

– Jest pewna liczba zmiennych zadeklarowanych w systemie, z których można korzystać w kodzie PL/SQL (ale nie w SQL) – ich wartości dotyczą ostatnio wykonanej instrukcji SQL.

`SQL%ROWCOUNT`	liczba wierszy przetworzonych przez ostatnią instrukcję SQL;
`SQL%FOUND=`*`True`*	jeśli został znaleziony (przetworzony) przynajmniej jeden wiersz;
`SQL%NOTFOUND=`*`True`*	jeśli nie znaleziono (przetworzono) żadnego wiersza;
`SQLERRM`	tekstowa informacja o błędzie;
`SQLCODE`	kod błędu.

– Obu zmiennych `SQLERRM` i `SQLCODE` można używać tylko w sekcji `EXCEPTION`.

– Przykład użycia zmiennej `SQL%ROWCOUNT` (do obliczenia liczby usuwanych działów o numerze 50) przedstawia następujący blok:

```
DECLARE usunięte NUMBER;
BEGIN
  DELETE FROM Dept WHERE Deptno = 50;
  usunięte := SQL%ROWCOUNT;
  INSERT INTO dziennik VALUES ('Dział', usunięte, Sysdate);
END;
/
```

Instrukcja SELECT w PL/SQL

- Wewnątrz bloku PL/SQL instrukcja SELECT nie wypisuje wyników na ekran i w zasadzie nie można jej użyć jako takiej bez dodatkowych konstrukcji.
- Instrukcja SELECT ma w PL/SQL swoją specjalną postać. Wynik zapytania nie jest wyświetlany na ekranie, tylko zostaje zapisany na zmiennych – zamieszczanych w klauzuli **INTO**, która jest wymaganą klauzulą instrukcji SELECT w PL/SQL.

```
SELECT wyrażenie, wyrażenie,...
INTO zmienna, zmienna,...
FROM tabela, tabela,...
[WHERE ...][GROUP BY ...][HAVING ...][FOR
UPDATE OF ...];
```

- Aby instrukcja była poprawna, instrukcja SELECT musi zwracać dokładnie jeden wiersz wyników.

– Przykład

```
SELECT e.Ename
INTO nazwisko
FROM Emp e
WHERE e.Empno = 1030;
```

– Wartości, na których operujemy w bloku PL/SQL, pochodzą na ogół z bazy danych, gdzie został określony ich typ danych. W związku z tym, wygodnie jest w bloku PL/SQL określać typ danych przez odniesienie do kolumny w bazie danych jako ''typ danych wymienionej kolumny'' np. nazwisko `Emp.Ename%TYPE`

DML w PL/SQL

```
DECLARE
   v_suma_plac NUMBER(6,2);
   v_ilu_pracownikow NUMBER(5);
   v_zespol zespoly%ROWTYPE;
BEGIN
   SELECT * INTO v_zespol FROM zespoly
   WHERE nazwa = 'ADMINISTRACJA';
   SELECT sum(placa_pod), count(*)
   INTO v_suma_plac, v_ilu_pracownikow
   FROM pracownicy WHERE id_zesp = v_zespol.id_zesp;
   dbms_output.put_line('Suma płac: ' || to_char(v_suma_plac));
   dbms_output.put_line('Pracowników: ' || to_char(v_ilu_pracownikow));
END;
```

DML w PL/SQL

- Zastosowanie poleceń INSERT, UPDATE i DELETE w programie PL/SQL w podstawowej postaci nie różni się niczym od postaci stosowanej w SQL.
- Opcjonalnie do poleceń można dodać klauzulę `RETURNING INTO`, która pozwala na zapisanie we wskazanej zmiennej:
  - » wartości atrybutów rekordu, wstawionego przez zlecenie INSERT,
  - » wartości atrybutów rekordu, zmodyfikowanego przez zlecenie UPDATE,
  - » wartości atrybutów rekordu, usuniętego przez zlecenie DELETE.

```
DECLARE
   v_id_prac pracownicy.id_prac%TYPE;
BEGIN
   INSERT INTO pracownicy (id_prac, imie, nazwisko)
   VALUES (400, 'Jacek','Kowalski')
   RETURNING id_prac INTO v_id_prac;
   dbms_output.put_line('Id nowego pracownika: ' || to_char(v_id_prac));
END;
```

Użycie zmiennych rekordowych w poleceniach DML

- W przypadku polecenia `INSERT` zmienną rekordową można podać bezpośrednio w klauzuli `VALUES` polecenia.
- Dla polecenia `UPDATE` istnieje możliwość modyfikacji wszystkich atrybutów rekordu relacji do wartości, jakie zawiera zmienna rekordowa. Do tego celu należy użyć konstrukcji `SET ROW = v_zmienna_rekordowa`.

```
DECLARE
   v_zespol zespoly%ROWTYPE;
BEGIN
   v_zespol.id_zesp := 70;
   v_zespol.nazwa := 'SIECI';
   v_zespol.adres := 'PIOTROWO 3A';
   INSERT INTO zespoly VALUES v_zespol;
   v_zespol.nazwa := 'SIECI KOMPUTEROWE';
   v_zespol.adres := 'WIENIAWSKIEGO';
   UPDATE zespoly SET row = v_zespol
   WHERE id_zesp = 70;
END;
```

Kursory

- Każde polecenie SQL, umieszczone w programie PL/SQL, w trakcie wykonania zostaje skojarzone z obszarem pamięci, w którym przechowywane są informacje o przetwarzanym poleceniu: m.in. status wykonania polecenia i zbiór odczytanych z bazy danych rekordów w przypadku zapytania.
- Dostęp w programie PL/SQL do tego obszaru pamięci, nazywanego również obszarem roboczym, jest możliwy za pomocą specjalnej struktury, tzw. kursora.
- Każdy kursor posiada swoją własną nazwę i może służyć np. do odczytania liczby rekordów, jakie zmodyfikowało polecenie UPDATE w programie PL/SQL, pobierania po kolei rekordów, jakie z bazy danych odczytało zapytanie, czy też sprawdzenia, czy polecenie DELETE usunęło rekordy.

Kursory

- Wyróżniamy dwa rodzaje kursorów: jawne i niejawne.
- Kursory jawne są deklarowane przez programistę i służą do odczytu zbioru rekordów z bazy danych (jak pamiętamy z poprzedniego ćwiczenia, zwykłe polecenie SELECT umieszczone w programie PL/SQL może odczytać z bazy danych tylko jeden rekord).
- Z kolei kursory niejawne są tworzone automatycznie dla każdego z poleceń UPDATE, INSERT, DELETE i SELECT INTO, jakie zostaje umieszczone w programie. Kursory niejawne najczęściej służą do sprawdzenia stanu polecenia, dla którego kursor niejawny został utworzony.

Kursory jawne

- Do zapytań, które mają zwrócić więcej niż jeden wiersz.
- Jawnie tworzone przez programistę
- Sposób użycia:
  1. zadeklarowanie w sekcji DECLARE
  2. otwarcie – wykonanie zapytania związanego z kursorem, odczytane z bazy danych rekordy trafiają do pamięci
  3. pobieranie kolejnych rekordów
  4. zamknięcie – zwolnienie obszaru pamięci kursora.

```
DECLARE
  CURSOR cur_zespoly IS SELECT * FROM zespoly;
  v_zespol zespoly%ROWTYPE;
BEGIN
  OPEN cur_zespoly;
  FETCH cur_zespoly INTO v_zespol;
  ...
  CLOSE cur_zespoly;
```

Obsługa wyjątków

- W PL/SQL standardowe błędy (wyjątki) mają przyporządkowane nazwy. Oto najczęściej używane nazwy błędów (wyjątków):
  - ***dup_val_on_index*** - powtórzenie tej samej wartości w indeksie jednoznacznym,
  - ***no_data_found*** - instrukcja SELECT nie zwróciła wartości dla zmiennych w klauzuli INTO,
  - ***too_many_rows*** - instrukcja SELECT zwróciła więcej niż jeden wiersz wartości dla zmiennych w klauzuli INTO,
  - ***zero_divide*** - dzielenie przez zero,
  - ***timeout_on_resource*** - zbyt długie oczekiwanie na zasoby,
  - ***invalid_cursor*** - niepoprawna operacja na kursorze,
  - ***login_denied*** - niepoprawna nazwa użytkownika/hasło,
  - ***invalid_number*** - niepoprawna konwersja na liczbę,
  - ***storage_error*** - brak pamięci,
  - ***value_error*** - błąd związany z działaniem na wartościach,
  - ***cursor_already_open*** - kursor już został otwarty,
  - ***program_error*** - błąd w interpreterze PL/SQL.

Przykład

```
DECLARE
  nazwisko Emp.Ename%TYPE;
  stanowisko Emp.Job%TYPE;
  komunikat VARCHAR2(100);
BEGIN
  SELECT e.Ename, e.Job
  INTO nazwisko, stanowisko
  FROM Emp e
  WHERE e.Hiredate BETWEEN '01-JAN-93' AND '01-JAN-94';
EXCEPTION
WHEN no_data_found THEN
  INSERT INTO Dziennik
  VALUES ('0 zatrudnionych w 93');
  DBMS_OUTPUT.Put_line('0 zatrudnionych w 93');
WHEN too_many_rows THEN
  INSERT INTO Dziennik VALUES (''Więcej niż 1 w 93');
  DBMS_OUTPUT.Put_line('Więcej niż 1 w 93');
WHEN OTHERS THEN -- Obsługa pozostałych błędów
  komunikat := 'Błąd nr.= ' || SQLCODE|| ',komunikat= ' ||
Substr(SQLERRM,1,100);
   -- SQLCODE i SQLERRM nie mogą wystąpić w instrukcji SQL!
  INSERT INTO Dziennik VALUES (komunikat);
  DBMS_OUTPUT.Put_line(‘Wystąpił inny błąd ');
END;
/
```

Obsługa wyjątków

- Jeśli 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, następuje propagacja błędu do zawierających go bloków 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 od razu propagowane do bloku zawierającego dany blok.
- Dobra praktyka programistyczna wymaga, aby każdy błąd został obsłużony – ewentualnie w klauzuli WHEN OTHERS THEN najbardziej zewnętrznego bloku. Aby móc stwierdzić, która instrukcja SQL spowodowała błąd, można używać podbloków z własną obsługą błędów, albo użyć licznika, zwiększającego się o jeden po wykonaniu każdej instrukcji SQL.

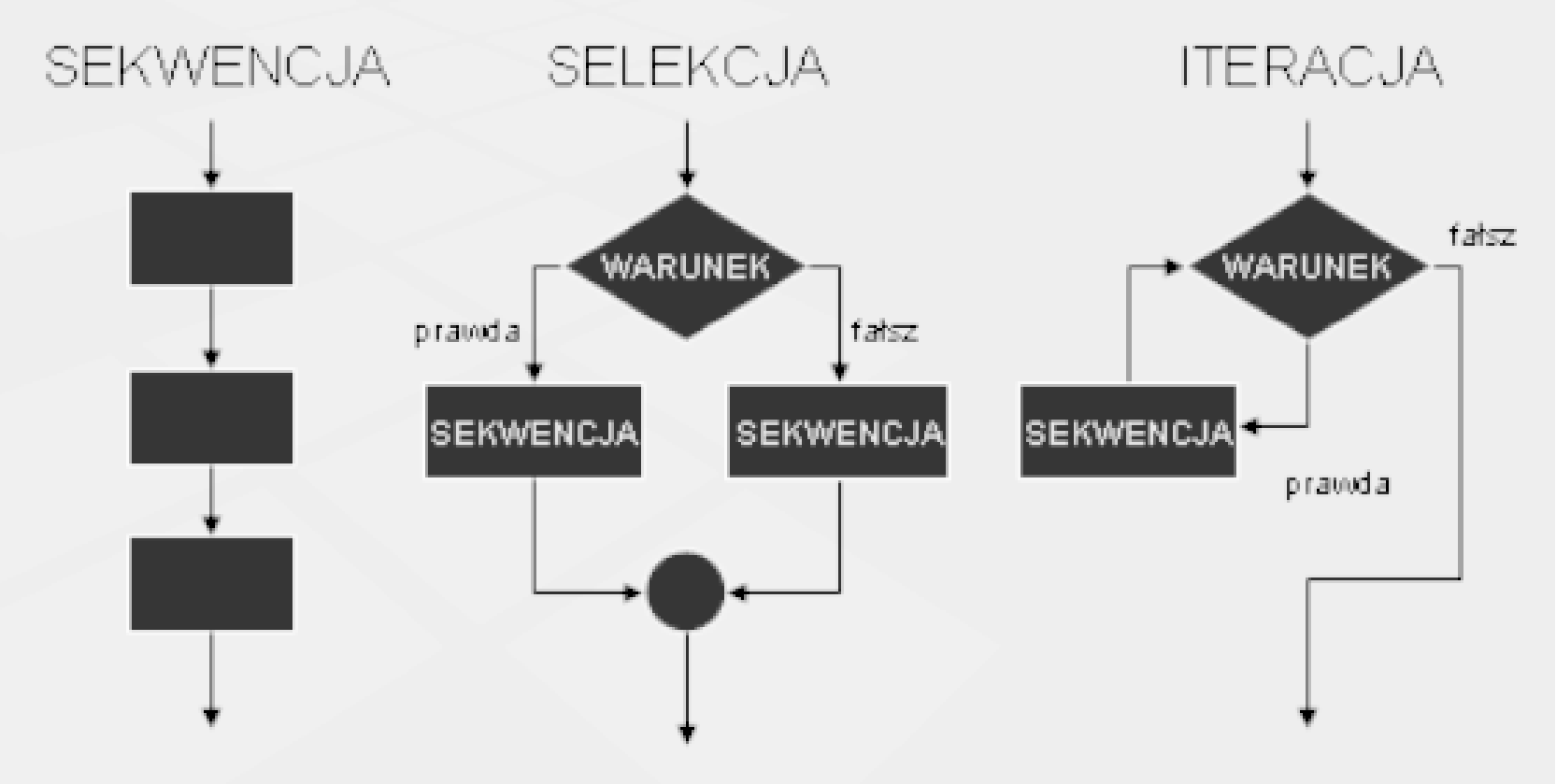
Wyjątki definiowane przez programistę

– Rozważmy obsługę błędu naruszenia więzów klucza obcego o numerze -2292.

```
DECLARE
  bl_klucz_o EXCEPTION;
  PRAGMA EXCEPTION_INIT (bl_klucz_o, -2292);
  v_deptno Dept.Deptno%TYPE := &p_deptno;
BEGIN
  DELETE FROM Dept
  WHERE Deptno = v_deptno;
  COMMIT;
EXCEPTION
WHEN bl_klucz_o THEN
  DBMS_OUTPUT.Put_line('Nie można usunąć działu ' ||
TO_CHAR(v_deptno) || ', w którym są pracownicy.');
END;
/
```

– PRAGMA oznacza dyrektywę wykonywaną przez kompilator. W tym przypadku dyrektywa o nazwie EXCEPTION_INIT powoduje przypisanie błędowi o numerze –2292 nazwy wyjątku *bl_klucz_o*. Do nazwy tej można następnie odwołać się w sekcji obsługi wyjątków w taki sam sposób, jak do każdego innego nazwanego wyjątku.

Instrukcje sterujące



Instrukcje sterujące

- Język PL/SQL posiada większość typowych instrukcji sterujących występujących w typowych językach programowania:

```
IF warunek THEN ciąg_instrukcji END IF;

IF warunek THEN ciąg_instrukcji
ELSE ciąg_instrukcji END IF;
```

- Jest jeszcze jedna postać instrukcji IF.

```
IF warunek THEN ciąg_instrukcji
ELSIF warunek THEN ciąg_instrukcji END IF;
```

- Drugim rodzajem operacji selekcji jest instrukcja `CASE`. Pozwala ona na testowanie wielu warunków, jej zapis jest bardziej zwięzły niż zapis instrukcji `IF ... THEN`.

Pętle

– Oto możliwe postacie iteracji:

» LOOP
ciąg instrukcji (w tym EXIT lub EXIT WHEN *warunek*)
END LOOP;

» FOR *zmienna* IN *wartość1* .. *wartość2*
LOOP
ciąg instrukcji
END LOOP;

» WHILE *warunek* LOOP
ciąg instrukcji
END LOOP;

Instrukcja NULL

- Czasami jest konieczne użycie instrukcji pustej – w języku PL/SQL jest to Null – na przykład, w sytuacji gdy obsługa wyjątku jest pusta.
- Nie realizuje ona żadnej akcji i jest właściwie swego rodzaju wypełniaczem, wykorzystywanym podczas testowania programów, których nie wszystkie elementy zostały jeszcze zdefiniowane.

```
DECLARE
    v_czy_zapłacona BOOLEAN := true;
BEGIN
    IF NOT v_czy_zapłacona THEN
        NULL;
    ELSE
        dbms_output.put_line('Faktura opłacona!');
    END IF;
END;
```

Kursory: dostęp do obszarów roboczych instrukcji SELECT

- Używając dotychczas wprowadzonych konstrukcji języka PL/SQL nie było możliwe przeglądanie kolejno wszystkich wierszy będących wynikiem zapytania. Do tego celu jest potrzebny obiekt PL/SQL o nazwie *kursor*, który stanowi bufor, do którego są zapisywane, kolejno sprowadzane z bazy danych, wiersze wynikowe zapytania.
- W sekcji deklaracji definiujemy kursor przyporządkowując mu instrukcję SELECT:
  ```
  CURSOR nazwa_kursora
  IS instrukcja_SELECT; -- (bez INTO!)
  ```
- Następnie otwieramy go za pomocą instrukcji OPEN:
  ```
  OPEN nazwa_kursora;
  ```
- co oznacza wykonanie instrukcji SELECT przyporządkowanej kursorowi. Po czym możemy w pętli pobierać, przy użyciu instrukcji
  ```
  FETCH nazwa_kursora INTO zmienna, …;
  ```
  kolejny wiersz wyników zapytania i przypisywać go na zmienne PL/SQL
- Standardowo, instrukcja FETCH jest umieszczana w pętli i towarzyszy jej instrukcja
  ```
  EXIT WHEN nazwa_kursora%NOTFOUND;
  ```
  powodująca wyjście z pętli po sprowadzeniu wszystkich wierszy wynikowych.
- Na koniec, należy zamknąć kursor za pomocą instrukcji
  ```
  CLOSE nazwa_kursora;
  ```
  aby zwolnić zasoby systemu przyporządkowane kursorowi.

Przykład

```
DECLARE
    zarobki REAL:=0;
    CURSOR kursor_osoba IS
       SELECT * FROM Emp;
    rek_osoby kursor_osoba%ROWTYPE;
  BEGIN
    OPEN kursor_osoba;
    LOOP
      FETCH kursor_osoba INTO rek_osoby;
      EXIT WHEN kursor_osoba%NOTFOUND;
      zarobki := zarobki + NVL(rek_osoby.Sal,0);
    END LOOP;
    DBMS_OUTPUT.Put_line('W sumie zarobki = '||zarobki);
    CLOSE kursor_osoba;
  END;
  /
```

użycie funkcji NVL (zamienia wartości null np. na 0) gwarantuje poprawne sumowanie zarobków w przypadku, gdy zarobki niektórych pracowników nie zostały określone (nieokreślone zarobki interpretujemy jako równe 0) – w takim przypadku bez zastosowania funkcji NVL otrzymalibyśmy wynik *Null* (reprezentowane pustym miejscem przy wyświetlaniu).

Przykład

– Zastosujmy blokowanie wierszy poprzez kursor do zaprogramowania podwyższenia zarobków o 10% najmniej zarabiającym pracownikom oraz zmniejszenia zarobków o 10% najwięcej zarabiającym pracownikom.

```
DECLARE
  CURSOR kursor_osoba IS
    SELECT e.Ename, e.Sal FROM Emp e
    FOR UPDATE OF e.Sal;
  rek_osoby kursor_osoba%ROWTYPE;
BEGIN
  OPEN kursor_osoba;
  LOOP
    FETCH kursor_osoba INTO rek_osoby;
    EXIT WHEN kursor_osoba%NOTFOUND;
    IF rek_osoby.Sal < 10000 THEN
      UPDATE Emp SET Sal = Sal * 1.1
      WHERE CURRENT OF kursor_osoba;
    ELSIF rek_osoby.Sal > 100000 THEN
      UPDATE Emp SET Sal = Sal * 0.9
      WHERE CURRENT OF kursor_osoba;
    END IF;
    /* zamiast modyfikować, możemy też usunąć wiersz, np.
       DELETE Emp WHERE CURRENT OF kursor_osoba; */
  END LOOP;
  CLOSE kursor_osoba;
  COMMIT;
END;
/
```

Dynamiczny SQL

- Do tej pory mieliśmy do czynienia wyłącznie z sytuacją, w której instrukcje SQL są w całości znane w chwili kompilacji jednostki programowej. W praktyce są jednak sytuacje, gdy w chwili kompilacji programu nie znamy pełnej postaci instrukcji SQL. Szczegóły mogą być ustalane w trakcie działania programu, jak np. z której tabeli wyświetlić wiersze.

```
SQLsource VARCHAR2(5000);
SQLprepped VARCHAR2(5000);
.....
/* Wczytaj na zmienną SQLsource tekst instrukcji SQL, którą chcesz wykonać */
EXEC SQL PREPARE SQLprepped FROM SQLsource;  -- skompiluj
EXEC SQL EXECUTE SQLprepped;                 -- wykonaj
......
```

- Zmienna SQLsource zawiera tekst instrukcji do wykonania. Natomiast zmienna SQLprepped służy do przechowania skompilowanej postaci instrukcji SQL. Instrukcja SQL PREPARE kompiluje ustaloną dynamicznie instrukcję SQL. Natomiast instrukcja SQL EXECUTE wykonuje ją.
- Po wykonaniu instrukcji SQL na systemowych zmiennych SQLSTATE i SQLCODE znajduje się informacja o statusie wykonania tej instrukcji.
- Standard języka SQL dostarcza zbioru instrukcji jak SQL PREPARE i SQL EXECUTE noszących razem nazwę dynamicznego SQL.

Dynamiczny SQL w Oracle

- Oracle ma swoją wersję dynamicznego SQL zrealizowaną za pomocą pakietu PL/SQL o nazwie DBMS_SQL. np.:

1. `DBMS_SQL.OPEN_CURSOR` - do otworzenia kursora,
2. `DBMS_SQL.PARSE` - do sparsowania wyznaczonej instrukcji,
3. `DBMS_SQL.EXECUTE` - do wykonania instrukcji,
4. `DBMS_SQL.CLOSE_CURSOR` - do zamknięcia kursora.
5. `DBMS_SQL.BIND_VARIABLE(`
   `c IN INTEGER,`
   `name IN VARCHAR2,`
   `value IN` *`typ danych`*`)` - związanie zmiennych gdzie
   - » *c* jest identyfikatorem kursora,
   - » *name* jest nazwą zmiennej wiązania,
   - » *value* jest wartością przypisywaną zmiennej wiązania.

Funkcje

– Język PL/pgsql umożliwia definiowanie funkcji, które można wykorzystać wewnątrz bazy PostgreSQL. Aby zdefiniować własną funkcję należy użyć polecenia CREATE FUNCTION. Ogólna postać definicji funkcji:

```
CREATE FUNCTION nazwa_funkcji ( [typ_arg1 [,typ_arg2 [,...]]] )
RETURNS typ_wyniku
AS  'BEGIN wnętrze_funkcji
END;'

LANGUAGE 'nazwa_języka'
```

– Przykład : Stwórz funkcję, która dla przyjmowanego id pracownika z tabeli `pracownicy` będzie zwracała jego nazwisko

```
CREATE FUNCTION nazwisko_pracownika ( int4 )
RETURNS text
AS' BEGIN
DECLARE n;
SELECT `nazwisko` FROM `pracownicy` WHERE `id`=$1;
RETURN n;
END;'
LANGUAGE 'nazwa_języka'
```

Deklaracje zmiennych dla funkcji zapisuje się w sekcji DECLARE definicji funkcji lub w bloku wewnątrz funkcji. Zmienne zadeklarowane w bloku są widoczne tylko wewnątrz niego oraz w innych blokach, które się w nim znajdują. Zmienna zadeklarowana w bloku wewnętrznym, posiadająca tę samą nazwę jak zmienna na zewnątrz bloku, przesłania zmienną zewnętrzną.

Zlecanie zadań do wykonania - pakiet DBMS_JOB

- Na ogół, kod PL/SQL jest wykonywany w ramach pewnej otwartej sesji. Można także zaplanować periodyczne wykonywanie kodu PL/SQL na serwerze bazy danych poza jakąkolwiek sesją. Tego typu złożone do wykonania zadania są realizowane na serwerze bazy danych za pomocą specjalnych procesów działających w tle o nazwie SNP.

```
DBMS_JOB.SUBMIT(
  job OUT BINARY_INTEGER,
  what IN VARCHAR2,
  next_date IN DATE DEFAULT SYSDATE,
  interval IN VARCHAR2 DEFAULT 'null',
  no_parse IN BOOLEAN DEFAULT FALSE)
```

gdzie:

- *job* - numer identyfikujący przekazane zadanie,
- *what* – procedura PL/SQL do periodycznego wykonywania,
- *next_date* - kiedy ma być wykonane zadanie,
- i*nterval* - albo null (tylko jedno wykonanie) albo następny punkt czasowy,
- *no_parse* - parsowanie dopiero przy pierwszym wykonaniu.

Procedury wyzwalane – wyzwalacze (TRIGGER)

- W przeciwieństwie do pozostałych rodzajów podprogramów, wyzwalacze nie są uruchamiane na żądanie użytkownika, ale automatycznie na skutek zajścia określonych zdarzeń w bazie danych.
- Zdarzenia te mogą być zdefiniowane dla relacji lub perspektywy (np. wstawienie, usunięcie lub modyfikacja rekordu), określonego schematu (np. utworzenie nowej relacji w schemacie) lub całej bazy danych (np. przyłączenie użytkownika do bazy danych).

```
CREATE [OR REPLACE] TRIGGER nazwa_procedury_wyzwalanej
    <moment uruchomienia>
    <zdarzenie uruchamiające> ON { relacja | perspektywa }
    [ WHEN warunek ]
    [ FOR EACH ROW ]
[ DECLARE <deklaracje stałych, zmiennych, kursorów> ]
BEGIN
    <ciało procedury wyzwalanej>
END;
```

Wyzwalacze - cele

- Celem wyzwalacza jest np. wymuszanie złożonych reguł biznesowych, np. zależności rekordów jednej relacji od rekordów innej relacji.
- Kolejne zastosowania to zaawansowane śledzenie operacji, realizowanych przez użytkowników bazy danych, wymuszanie złożonych polityk bezpieczeństwa (np. uniemożliwianie pracy użytkownikom w odpowiednich porach), wypełnianie atrybutów relacji wartościami domyślnymi przy wstawianiu nowych rekordów.
- Bardzo ważnym zastosowaniem jest umożliwianie modyfikacji złożonych perspektyw relacyjnych (problem ten zostanie omówiony w dalszej części ćwiczenia).

Wyzwalacz polecenia

- Wyzwalacz taki wykonywany jest zawsze jednokrotnie, niezależnie od liczby rekordów, jakie przetworzyło polecenie.
- Ograniczeniem takiego wyzwalacza jest niemożność bezpośredniego odwołania w ciele wyzwalacza do danych relacji lub perspektywy, na której założono wyzwalacz (takie odwołanie jest możliwe w w przypadku wyzwalaczy wierszowych).

```
CREATE TRIGGER ZapiszOperacjeInsert
  AFTER INSERT ON pracownicy
BEGIN
  INSERT INTO log (data, relacja, operacja)
  VALUES(sysdate, 'PRACOWNICY', 'INSERT');
END;
```

Wyzwalacz wierszowy

- Wyzwalacz wierszowy jest wykonywany jednokrotnie dla każdego rekordu, przetworzonego przez polecenie uruchamiające wyzwalacz.
- Aby wyzwalacz był wyzwalaczem wierszowym, należy w jego definicji podać klauzulę `FOR EACH ROW`.
- Wyzwalacz wierszowy ma jedno poważne ograniczenie – w jego ciele nie może zostać wykonana żadna operacja odczytu lub modyfikacji danych relacji lub perspektywy, dla której zdefiniowano wyzwalacz. Gdyby realizacja takich operacji była dopuszczalna w wyzwalaczu wierszowym, SZBD nie mógłby zagwarantować spójności operacji.
- Za to w wyzwalaczu wierszowym można bezpośrednio odwołać się do wartości atrybutów rekordu relacji lub perspektywy, dla którego wyzwalacz został uruchomiony.

```
CREATE TRIGGER WstawIdentyfikator
  BEFORE INSERT ON pracownicy
  FOR EACH ROW
BEGIN
  IF :NEW.id_prac IS NULL THEN
    SELECT seq_pracownicy.nextval INTO :NEW.id_prac
    FROM dual;
  END IF;
END;
```

PL/SQL

- PL/SQL umożliwia ponadto:
  - Komunikacja z innymi sesjami - pakiet DBMS_PIPE
  - Komunikacja przez system plików - pakiet UTL_FILE
  - Biblioteki (obsługa obiektów typu LIBRARY)
  - koda Java składowany w bazie danych (obsługa dzięki sterownikowi JDBC)
  - obsługa osadzonego SQL

Więcej np.: Bill Pribyl, Steven Feuerstein, *Oracle PL/SQL. Wprowadzenie*, Helion 2002