	Bazy Danych laboratorium	**Laboratorium BD9**

Zagadnienie: Konstruowanie i zastosowanie automatów programowych w PL/SQL

Na potrzeby zajęć zostanie wykorzystany model bazy danych opisany i zaimplementowany w ramach Laboratorium BD7.

I. Czynności wstępne

Przy pomocy zdań SQL należy zmodyfikować zaimplementowany model sprzedaży zawierający trzy tabele z prefixem BD4 wykonując poniższe czynności:

1. W tabeli BD4_PRODUKT dodać kolumnę ILOSC_W_MAGAZYNIE, która będzie przechowywała ilość danego produktu w magazynie. Ustawić jako domyślną wartość 0.

2. Dla wszystkich produktów wprowadzić konkretną wartość do tej kolumny - dla uproszczenia można wszystkim produktom nadać tę samą wartość.

3. W tabeli BD4_RACHUNEK dodać kolumnę STATUS_RACHUNKU i określić dla niej przy pomocy definicji CHECK dopuszczalne wartości: OTWARTE, ZAMKNIĘTE i ANULOWANE obrazujące stan realizacji danego zamówienia.

II. Wyzwalacze (triggers) jako automaty bazodanowe

Wyzwalacze są blokami programowymi, które są uruchamiane automatycznie (niejawnie) w wyniku zajścia określonego zdarzenia. Najczęściej stosowane są w sytuacji wystąpienia w kodzie programowym zdania DML (*insert*, *update*, *delete*) do pewnej tabeli. W przypadku, gdy dla tej tabeli skonstruowany jest odpowiedni wyzwalacz – jest on uruchamiany przed lub po zdaniu DML.
Jego rolą może być uzupełnienie zdania DML, które wywołało wyzwalacz, realizacja zdania DML w innej tabeli lub walidacja danych czyli sprawdzenie poprawności danych wejściowych w procedurze lub bezpośrednio w zdaniu DML.

Są dwa typy wyzwalaczy: wyzwalacze zdania i wyzwalacze wiersza.
Wyzwalacz zdania (*for statement*) jest uruchamiany raz dla zdania DML (zdarzenia wyzwalającego) bez względu na liczbę wierszy objętych działaniem tego DML, nawet jeśli liczba wierszy wynosi zero.

Wyzwalacz wiersza (*for each row*) jest uruchamiany dla każdego wiersza objętego działaniem zdarzenia wyzwalającego. Jeśli żaden wiersz nie jest objęty jego działaniem, nie dochodzi w ogóle do uruchomienia wyzwalacza.

Przykładowo, jeśli dla jakiejś tabeli został zdefiniowany wyzwalacz typu *for statement* reagujący na zdanie *update* do tej tabeli, to w momencie uruchomienia zdania:

```
update tabela
        set kolumna = nowa_wartosc
where kolumna_inna <= 3;
```

scenariusz będzie taki:

1. *update* dla kolumna_inna = 1,
2. *update* dla kolumna_inna = 2,
3. *update* dla kolumna_inna = 3,
4. wywołanie (odpalenie - fired) wyzwalacza. -- jednorazowe uruchomienie

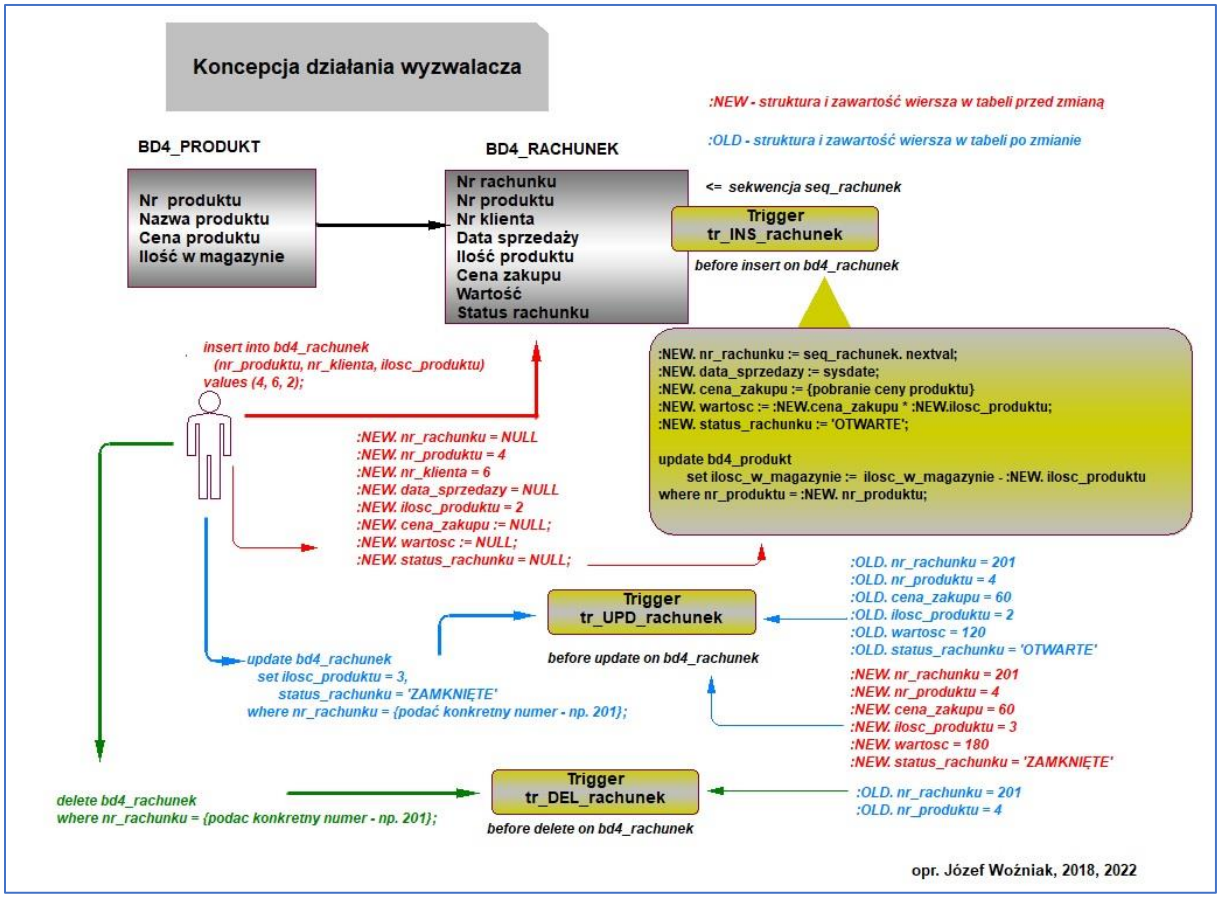
Jeśli dla tej samej tabeli został zdefiniowany wyzwalacz typu *for each row* reagujący na zdanie *update* do tej tabeli, to w momencie uruchomienia tego samego zdania scenariusz będzie inny:

1. *update* dla kolumna_inna = 1,
2. wywołanie wyzwalacza,
3. *update* dla kolumna_inna = 2,
4. wywołanie wyzwalacza,
5. *update* dla kolumna_inna = 3,
6. wywołanie wyzwalacza.

Moment uruchomienia wyzwalacza zależy od klauzuli BEFORE lub AFTER.
BEFORE – treść wyzwalacza jest wykonywana przed zdaniem DML,
AFTER – treść wyzwalacza jest wykonywana po zdaniu DML.

Powyższe dwa przykłady dotyczą klauzuli AFTER.

Przy pomocy poniższego diagramu zostanie omówiona koncepcja działania wyzwalacza bazodanowego.



Z wyzwalaczami związane są dwie struktury: :OLD i :NEW. Ich budowa jest tożsama ze strukturą tabeli, z którą dany wyzwalacz jest związany, czyli jeśli tabela ma strukturę:

nr_produktu,
nazwa_produktu,
cena_produktu,
ilosc_w_magazynie,
.....

to obie powyższe struktury mają taką samą budowę i można się do elementów tych struktur odwoływać:

:NEW.nr_produktu :OLD.nr_produktu
:NEW.nazwa_produktu :OLD.nazwa_produktu

i tak dalej.

Działanie wyzwalacza zbudowanego dla tabeli BD4_RACHUNEK reagującego na zdanie *insert*

Wprowadzenie nowej pozycji do tabeli BD4_RACHUNEK wiąże się z określeniem: numeru rachunku, numeru klienta, numeru produktu i ilości tego produktu, daty złożenia zamówienia oraz statusu rachunku. Niektóre z tych danych mogą być ustawiane automatycznie. Należą do nich: data złożenia zamówienia (*sysdate*), status rachunku (w momencie składania zamówienia status jest OTWARTE) oraz numer rachunku (klucz główny tabeli może być wyznaczany przez sekwencję). Od wprowadzającego nowe zamówienie wymagane będzie podanie tylko numeru produktu i jego ilości oraz numeru klienta, który to zamówienie składa. Pozostałe wielkości zostaną ustawione w wyzwalaczu.

Kod wyzwalacza będzie wyglądał tak:

```
create or replace trigger tr_ins_rachunek
 before insert on bd4_rachunek
for each row
begin
        :new. nr_rachunku := seq_rachunek.nextval;
        :new. data_sprzedazy := sysdate;
        :new. status_rachunku := 'OTWARTE';
end;
```

, a zdanie DML wyzwalające go:

```
insert into bd4_rachunek (nr_produktu, nr_klienta, ilosc_produktu)
values ( 4, 6, 2);
```

Struktury :NEW i :OLD przed wykonaniem się zdania *insert* będą wyglądały następująco (fragment):

	Nr rachunku	Data sprzedaży	Ilość produktu	Nr produktu	Nr klienta	Status rachunku
:NEW	Null	Null	2	4	6	Null
:OLD	Null	Null	Null	Null	Null	Null

Działanie wyzwalacza spowoduje, że struktura :NEW się zmieni (fragment):

	Nr rachunku	Data sprzedaży	Ilość produktu	Nr produktu	Nr klienta	Status rachunku
:NEW	201	sysdate	2	4	6	OTWARTE
:OLD	Null	Null	Null	Null	Null	Null

I tak skonstruowane zdanie *insert* zostanie wykonanie.
Rolą tego wyzwalacza było uzupełnienie zdania *insert* o standardowe wartości kolumn. Uzupełnienie to nastąpiło przed wykonaniem się zdania *insert* z racji określenia klauzuli BEFORE (najpierw wyzwalacz, a potem *insert*).
Drugim zadaniem wyzwalacza może być działanie w innej tabeli. W omawianym przykładzie w momencie złożenia zamówienia musi nastąpić rezerwacja żądanej ilości produktu czyli modyfikacja jego ilości w magazynie (w tabeli BD4_PRODUKT).

Można zatem rozszerzyć kod wyzwalacza o tę funkcjonalność:

```
.....
update bd4_produkt
        set ilosc_w_magazynie = ilosc_w_magazynie - :new. ilosc_produktu
where nr_produktu = :new. nr_produktu;
```

Przetestować działanie tak skonstruowanego wyzwalacza można według scenariusza:

1. Odczytać z tabeli produktów ilość danego produktu w magazynie,
2. Złożyć zamówienie na ten produkt w ilości nie przekraczającej stanu magazynu,
3. Odczytać zawartość ewidencji zamówień,
4. Sprawdzić stan zamówionego produktu w magazynie.

Uwagi:

1. W powyższym fragmencie kodu wyzwalacza struktura :NEW jest strukturą tabeli BD4_RACHUNEK, a nie tabeli BD4_PRODUKT.

2. W kodzie wyzwalacza nie można używać zdań DML do tej samej tabeli, na przykład w powyższym wyzwalaczu zabronione jest użycie zdania *update* BD4_RACHUNEK lub *delete* BD4_RACHUNEK, natomiast można używać zdania *select* ... *from* BD4_RACHUNEK.

3. W wyzwalaczu nie można używać zdań *commit* i *rollback*.

4. W przypadku konieczności zdefiniowania wewnątrz wyzwalacza zmiennych należy użyć sekcji DECLARE:

```
.....
for each row
declare
        [deklaracje zmiennych tak jak w bloku anonimowym PL/SQL]
begin
        .....
end;
```

5. Chcąc zdefiniować wyzwalacz zdania należy opuścić klauzulę *for each row*.

Działanie wyzwalacza zbudowanego dla tabeli BD4_RACHUNEK reagującego na zdanie *update*

W przypadku konieczności modyfikacji szczegółów zamówienia w tabeli BD4_RACHUNEK należy wykonać zdanie:

```
update bd4_rachunek
        set ilosc_produktu = 3  -- zmiana tylko ilości zamówionego produktu
where nr_rachunku = 201;
```

lub

```
update bd4_rachunek
        set ilosc_produktu = 3,
            nr_produktu = 5              -- zmiana asortymentu i ilości
where nr_rachunku = 201;
```

Zmiana warunków zamówienia pociąga za sobą zmianę ilości towarów w magazynie, a więc w tabeli BD4_PRODUKT.

Kod wyzwalacza będzie wyglądał tak:

```
create or replace trigger tr_upd_rachunek
 before update on bd4_rachunek
for each row
begin
        update bd4_produkt
                set ilosc_w_magazynie = ilosc_w_magazynie - :new. ilosc_produktu
                                                          + :old. ilosc_produktu
        where nr_produktu = :new. nr_produktu;
end;
```

Struktury :NEW i :OLD przed wykonaniem się zdania *update* (dotyczącego tylko zmiany ilości zamówionego produktu) będą wyglądały następująco (fragment):

	Nr rachunku	Data sprzedaży	Ilość produktu	Nr produktu	Nr klienta	Status rachunku
:NEW	201	2022/01/26	3	4	6	OTWARTE
:OLD	201	2022/01/26	2	4	6	OTWARTE

I tak skonstruowane zdanie *update* zmieni szczegóły zamówienia, a wyzwalacz zmieni ilość produktu w magazynie.

Działanie wyzwalacza zbudowanego dla tabeli BD4_RACHUNEK reagującego na zdanie *delete*

W przypadku konieczności skasowania złożonego zamówienia (wiersza w tabeli BD4_RACHUNEK) można postępować dwojako. Albo skasować to zamówienie bezpowrotnie zmieniając odpowiednio stan magazynowy produktu w tabeli BD4_PRODUKT albo nie kasować zamówienia, tylko zmienić jego status na ANULOWANE i zmienić stan magazynowy produktu.
Poniżej zostanie zaprezentowane rozwiązanie dla pierwszego wariantu.

W przypadku konieczności skasowania zamówienia w tabeli BD4_RACHUNEK należy wykonać zdanie:

```
delete bd4_rachunek
where nr_rachunku = 201;
```

Kod wyzwalacza będzie wyglądał tak:

```
create or replace trigger tr_del_rachunek
 before delete on bd4_rachunek
for each row
begin
        update bd4_produkt
                set ilosc_w_magazynie = ilosc_w_magazynie  + :old. ilosc_produktu
        where nr_produktu = :old. nr_produktu;
end;
```

Struktury :NEW i :OLD przed wykonaniem się zdania *delete* będą wyglądały następująco (fragment):

	Nr rachunku	Data sprzedaży	Ilość produktu	Nr produktu	Nr klienta	Status rachunku
:NEW	Null	Null	Null	Null	Null	Null
:OLD	201	2021/01/26	3	4	6	OTWARTE

I tak skonstruowane zdanie *delete* usunie złożone zamówienie, a wyzwalacz "zwróci" nie sprzedany a zarezerwowany produkt do magazynu.

Uwagi:

1. W przypadku wyzwalacza reagującego na zdanie *update* możliwe jest wskazanie kolumny lub kolumn, zmiana których spowoduje uruchomienie zaprojektowanego wyzwalacza.

   Na przykład wyzwalacz:

   ```
   create or replace trigger tr_upd_rachunek
   before update of ilosc_produktu on bd4_rachunek
   for each row
           ......
   ```

będzie reagował na zdanie:

```
update bd4_rachunek
        set ilosc_produktu = 3
where nr_rachunku = 201;
```

, ale nie będzie reagował na zdanie:

```
update bd4_rachunek
        set nr_produktu = 2
where nr_rachunku = 201;
```

2. Można łączyć w jednym wyzwalaczu akcje podejmowane w przypadku wykonywania różnych zdań DML, na przykład:

```
create or replace trigger tr_rachunek
before insert or delete or update on bd4_rachunek
        .......
```

W takim przypadku mogą być pomocne predykaty warunkowe: *INSERTING*, *UPDATING* i *DELETING* do budowania kodu wyzwalacza:

```
if INSERTING then ......end if;
if UPDATING then ....  .end if;
if UPDATING ( 'ilosc_produktu' )  then.....end if;
if UPDATING ( 'nr_produktu' ) then ..... end if;
if DELETING then ..... end if;
```

3. Można budować wyzwalacz w oparciu o perspektywę. Na przykład:

Załóżmy, że istnieje perspektywa bd4_rachunek_produkt zawierająca poniższe kolumny:

Nr_rachunku, Data_sprzedazy, Nazwa_produktu, Cena_produktu, Nazwisko_klienta

W oparciu o tę perspektywę można rejestrować nowe zamówienie, nawet w przypadku, gdy nazwisko klienta nie figuruje w ewidencji klientów lub brak jest w ewidencji produktu o podanej nazwie.

Nagłówek wyzwalacza będzie wyglądał tak:

```
create or replace trigger tr_ins_rachunek_view
instead of insert on  bd4_rachunek_produkt
        .....
```

Realizując zdanie:

```
insert into bd4_rachunek_produkt (Nr_rachunku, Data_sprzedazy,
                                  Nazwa_produktu,..... )
 values (.............);
```

należy poprzez wyzwalacz zapewnić, aby w tabeli BD4_PRODUKT znalazł się produkt występujący w powyższym zdaniu, jeśli go tam jeszcze nie ma. Analogicznie należy postępować w przypadku nowego klienta.

Kod takiego wyzwalacza zawierać może takie zdania *insert* i/lub *update* do różnych tabel, które zapewnią skuteczne wprowadzenie nowego zamówienia do tabeli BD4_RACHUNEK.

Walidacja danych przy pomocy wyzwalacza

Walidacja wprowadzanych danych do tabeli czyli kontrola ich poprawności może być realizowana przy pomocy wyzwalacza.
Przykładowo chcąc wpisać nowy produkt do tabeli BD4_PRODUKT można kontrolować, czy produkt o zadanej nazwie już występuje w tabeli. Ponieważ nazwa produktu nie jest kluczem głównym, więc należy dokonać dodatkowej programowej kontroli. Można to zrealizować poprzez wyzwalacz reagujący na zdanie *insert* do tabeli BD4_PRODUKT. Jego zadaniem będzie stwierdzenie, czy podana nazwa produktu już w tabeli istnieje i jeśli tak dać komunikat typu *raise* czyli przerwanie wykonywania się zdania *insert*.

Procedura wpisująca nowy produkt do tabeli BD4_PRODUKT może wyglądać tak:

```
create or replace procedure pr_insert_produkt
            (
             v_nazwa_produktu varchar2,
             v_cena_produktu numeric,
             v_rok_produkcji numeric default null,
             v_ranking number default 5,
             v_ilosc_w_magazynie numeric) AS
begin
  insert into bd4_produkt
    values (seq_produkt.nextval,, v_nazwa_produktu, v_cena_produktu, v_rok_produkcji,
              v_ranking,   v_ilosc_w_magazynie);

  dbms_output.put_line ( ' Dalsze przetwarzanie...' );
end;
```

W ciele procedury instrukcja *dbms_output.put_line* symbolizuje dalsze (ewentualne) przetwarzanie danych.

Można opracować wyzwalacz reagujący na zdanie *insert* do tabeli produktów o postaci:

```
create or replace trigger tr_ins_produkt
before insert on bd4_produkt
for each row
declare
  v_ile integer;
begin
  select count ( * ) into v_ile
  from bd4_produkt
  where nazwa_produktu = :new. nazwa_produktu;

  if v_ile = 1 then
        raise_application_error (-20001,
                                  'Produkt '||:new. nazwa_produktu||' już jest  zarejestrowany');
  end if;
end;
```

Zakładając, że w tabeli BD4_PRODUKT nie ma produktu o nazwie Canon 6D Mark II i uruchamiając powyższą procedurę, na przykład blokiem:

```
begin
   pr_insert_produkt (‘Czapka męska’, 120, 2021, 4, 15);
end;
```

w panelu Dbms Output zostanie wyświetlony komunikat *Dalsze przetwarzanie...* pochodzący z procedury, co oznacza, że nowy produkt został wpisany do tabeli:

NR_PRODUKTU	NAZWA_PRODUKTU	CENA_PRODUKTU	ROK_PRODUKCJI	RANKING	ILOSC_W_MAGAZYNIE
20	Czapka meska	120	2021	4	15
1	Koszulka polo	150	2019	3	15
2	Spodnie	180	2019	2	15
3	Żakiet	150	2019	1	15
4	Bluzka	60	2019	3	15

...

, a wyzwalacz nie wykazał powtórzonej nazwy.

Jeśli ponownie będziemy chcieli wprowadzić do tabeli produkt o tej samej nazwie i nowym numerze produktu:

```
begin
   pr_insert_produkt (‘Czapka męska’, 100, 2021, 6, 20);
end;
```

otrzymamy komunikat zgłoszony przez *raise_application_error* w wyzwalaczu:

```
Error report -
ORA-20001: Produkt Czapka męska już jest  zarejestrowany
ORA-06512: przy "A2022.TR_INS_PRODUKT", linia 9
ORA-04088: błąd w trakcie wykonywania wyzwalacza 'A2022.TR_INS_PRODUKT'
```

, a w panelu Dbms Output nie pojawi się komunikat *Dalsze przetwarzanie...*, co oznacza, że działanie procedury zostało przerwane czyli **cała (!!!)** procedura się nie wykonała i ten sam produkt nie został wpisany do tabeli.

Procedura *raise_application_error* pozwala projektantowi tworzyć własne komunikaty błędu i umieszczać je w kodzie PL/SQL.
Pierwszy argument może przyjmować wartości z zakresu -20000 do -20999, a drugi zawiera komunikat, który ma być przekazany w wyniku wystąpienia błędu (wyjątku - *exception*).

Jeśli taki kod PL/SQL będzie umieszczony w aplikacji utworzonej w jakimś języku programowania, to sposób dalszej obsługi takiego komunikatu może być różny. Temat ten nie będzie omawiany w tym materiale.

Zadania do samodzielnego wykonania

1. Zmodyfikować tabelę BD4_KLIENT dodając do niej kolumnę STATUS_KLIENTA. Kolumna ta będzie określała status klienta na podstawie sumarycznej wielkości przeprowadzanych transakcji według zasady:

   Jeśli suma zakupów przekroczy wartość HIGH klient otrzymuje status BARDZO WAŻNY, jeśli ta suma jest w zakresie LOW i HIGH to klient otrzymuje status WAŻNY, w przeciwnym przypadku nie ma żadnego statusu.

   Opracować kod PL/SQL (blok anonimowy lub procedurę), który jednorazowo na podstawie zawartości tabeli BD4_RACHUNEK dokona modyfikacji zawartości kolumny STATUS_KLIENTA.

   Opracować wyzwalacz, który na bieżąco będzie dokonywał modyfikacji tej kolumny w momencie rejestrowania nowego zamówienia w tabeli BD4_RACHUNEK.

2. Opracować jeden zbiorczy wyzwalacz łączący w sobie omówione w materiale wyzwalacze zbudowane w oparciu o tabelę BD4_RACHUNEK i reagujący na zdania *insert*, *update* i *delete*. Wykorzystać predykaty warunkowe *INSERTING*, *UPDATING* i *DELETING*.

3. Utworzyć tabelę słownikową BD4_RABATY zawierającą wartości procentowe rabatów przyznawanych klientom o odpowiednim statusie przy składaniu kolejnych zamówień. Tabela może składać się z dwóch kolumn: nazwa_statusu (HIGH i LOW) oraz wartości progowych (np. 10% i 5%).
   Utworzyć odpowiednią relację między tabelami BD4_RABATY i BD4_KLIENCI.
   Zmodyfikować tabelę BD4_RACHUNEK dodając do niej kolumnę: RABAT. Kolumna RABAT otrzymuje wartość zgodną ze statusem klienta, a wartość rachunku obliczana jest na podstawie ilości zamówionego produktu, ceny tego produktu i przydzielonego rabatu.
   Zmodyfikować, opracowany w punkcie 2, wyzwalacz uwzględniający przyznawanie rabatu klientowi przy składaniu zamówień.

III. Zadania (jobs) jako automaty czasowe

Omawiane do tej pory automaty typu wyzwalacze działały w ten sposób, że były wyzwalane zdarzeniem jakim mogło być jedno ze zdań DML (*insert*, *update* czy *delete*).

Automaty czasowe zwane zadaniami (częściej "jobami") są wyzwalane czasem. Można na przykład określić zadanie polegające na wydruku raportu analitycznego pierwszego dnia każdego miesiąca o godzinie 06:16, czy też uruchamiać konkretną procedurę co dwie godziny.

Podstawowymi pojęciami używanymi w tej tematyce są: *schedule*, *program* i *job*.

Schedule (harmonogram) - terminarz wykonywania jobów. Można zrobić harmonogram który określa realizację pewnej czynności z określoną częstotliwością i skojarzyć go z kilkoma jobami. Dzięki temu kilka jobów uruchamianych jest zgodnie z jednym harmonogramem i w przypadku konieczności jego zmiany robi się to w jednym miejscu. Odpowiada na pytanie*: Kiedy to ma się wykonać?*

Program – w nim można zdefiniować działania przy pomocy bloku anonimowego, procedury PL/SQL czy też pliku zewnętrznego z poziomu systemu operacyjnego. Można go skojarzyć z wieloma jobami i również w przypadku jego zmiany robi się to w jednym miejscu. Odpowiada na pytanie: *Co ma się wykonać?*

Job - łączy w sobie definicje harmonogramu i programu czyli odpowiada na pytanie: *Jaki program i zgodnie z jakim harmonogramem ma się wykonać?* Jako jedyny z tych obiektów ma charakter dynamiczny, to znaczy tylko uruchomienie joba zrealizuje zadanie. Harmonogramy i programy stanowią statyczne definicje.

Do zarządzania zadaniami (tworzenie zadań, uruchamianie ich oraz kasowanie) służy specjalny pakiet programowy Oracle o nazwie *dbms_scheduler*.

Poniżej zostaną przedstawione przykładowe sposoby definiowania elementów umożliwiających wykorzystanie jobów w realizacji zadań w bazie danych.

1. Tworzenie harmonogramów przy użyciu dbms_scheduler.create_schedule

```
begin
   -- codziennie od Monday do Friday o godzinie 22:00

   dbms_scheduler.create_schedule
         (schedule_name => 'INTERVAL_DAILY_2200',
          start_date=> trunc(sysdate)+18/24,       -- start dzisiaj o 18:00
          repeat_interval=>'freq=DAILY; byday=MON,TUE,WED,THU,FRI; byhour=22',
          comments=>'Uruchamiane (Mon-Fri) o 22:00'
          );
end;
```

```
begin
   -- codziennie co godzinę

   dbms_scheduler.create_schedule

         (schedule_name  => 'INTERVAL_EVERY_HOUR',
          start_date    => trunc ( sysdate ) + 18/24,          -- uaktywnienie nastąpi o godzinie 18:00
          repeat_interval => 'freq=HOURLY; interval=1',
          comments     => 'Uruchamiane codziennie co godzinę'
    );

   -- codziennie co 5 minut

   dbms_scheduler.create_schedule

          (schedule_name  => 'INTERVAL_EVERY_5_MINUTES',
          start_date    => trunc ( sysdate +1 ) + 20/24/60,   -- uaktywnienie nastąpi o godzinie 00:20
                                                               -- następnego dnia
          repeat_interval => 'freq=MINUTELY; interval=5',
          comments     => 'Uruchamiane codziennie co 5 minut'
    );

   -- codziennie co minutę przez 30 dni

   dbms_scheduler.create_schedule

         (schedule_name  => 'INTERVAL_EVERY_MINUTE',
         start_date    => trunc ( sysdate +1 ) + 20//24/60,
         end_date      => trunc ( sysdate ) + 30,
         repeat_interval => 'freq=MINUTELY; interval=1',
          comments     => 'Uruchamiane codziennie przez 30 dni co 1 minutę'
     );
       -- w każdą niedzielę o godzinie 18:00

   dbms_scheduler.create_schedule

         (schedule_name => 'INTERVAL_EVERY_SUN_1800',
          start_date=> trunc ( sysdate ) + 18/24,
         repeat_interval=> 'freq=DAILY; byday=SUN; byhour=18;',
         comments=>'Uruchamiane w niedzielę o godzinie 18'
    );
end;
```

2. Tworzenie programów przy użyciu dbms_scheduler.create_program

```
begin
    -- wywołanie bloku anonimowego

   dbms_scheduler.create_program
         (program_name => 'PROG_INIT_DRAWING_TABLE',
         program_action =>
                  ' BEGIN
                        delete from DRAWING_TABLE;
                    END; ' ,
          program_type => 'plsql_block',
          comments => 'Inicjowanie tabeli DRAWING_TABLE',
     );
   end;
```

```
begin
   -- wywołanie wbudowanej procedury

   dbms_scheduler.create_program
         (program_name=> 'PROG_DRAWING_ONE_VALUE',
          program_type=> 'stored_procedure',
          program_action=> 'pr_generation_drawing_value',
         comments=>'Losowanie pojedynczej wartości'

   );
end;

-----------------------------------

begin
   -- wywołanie pakietowej procedury

   dbms_scheduler.create_program
          (program_name=> 'PROG_PERCENT_DRAWING_TABLE',
           program_type=> ' stored_procedure ',
          program_action=> 'pkg_drawing.pr_count_percent';
          comments=>'Obliczanie procentów w DRAWING_TABLE'

   );
end;
```

3. Tworzenie zadania (job) przez połączenie harmonogramu z programem przy użyciu dbms_scheduler.create_job:

```
begin

   -- połączenie harmonogramu z programem

         dbms_scheduler.create_job
                (job_name => 'JOB_DRAWING_ONE_VALUE',
                 program_name=> 'PROG_DRAWING_ONE_VALUE',
                 schedule_name=>'INTERVAL_EVERY_MINUTE',
                 comments=>'Losowanie jednej wartości co 1 minutę');


        dbms_scheduler.create_job
                (job_name => 'JOB_PERCENT_DRAWING_TABLE',
                 program_name=> 'PROG_PERCENT_DRAWING_TABLE',
                 schedule_name=>'INTERVAL_EVERY_5_MINUTES',
                 comments=>'Obliczanie procentów w DRAWING_TABLE');

end;
```

4. Tworzenie zadania (job) bez definicji harmonogramu i programu przy użyciu dbms_scheduler.create_job:

Jedną z właściwości pakietów programowych jest możliwość przeciążania procedur w nich zawartych. Procedura pakietowa *create_job* jest taką właśnie procedurą. Dzięki temu można definiować zadania na różne sposoby.

Na przykład zawrzeć w jego definicji nazwę programu oraz szczegółowe parametry harmonogramu:

```
.....

dbms_scheduler.create_job
        (job_name => 'JOB_DRAWING_ONE_VALUE',
         program_name=> 'PROG_DRAWING_ONE_VALUE',
         start_date    => trunc ( sysdate ) + 18/24,
         end_date      => trunc ( sysdate ) + 30,
         repeat_interval => 'freq=MINUTELY; interval=1',
          comments=>'Losowanie jednej wartości co 1 minutę przez 30 dni');

.........
```

lub w ogóle nie definiować nazw programu i harmonogramu:

```
.....

dbms_scheduler.create_job
        (job_name => 'JOB_PERCENT_DRAWING_TABLE',

        program_type=> 'stored_procedure',
        program_action=> 'pkg_drawing.pr_count_percent';

        start_date    => trunc ( sysdate ) + 18/24,
        repeat_interval => 'freq=MINUTELY; interval=5'

        comments=>'Obliczanie procentów w DRAWING_TABLE');

        .....
```

Procedura *create_job* zawiera w sobie wszystkie możliwe argumenty formalne występujące w procedurach *create_program* i *create_schedule*.

5. Uruchamianie zadań (jobów):

```
begin
    dbms_scheduler.run_job ( 'JOB_DRAWING_ONE_VALUE' );
    dbms_scheduler.run_job ( 'JOB_PERCENT_DRAWING_TABLE' );
end;
```

6. Restart zadania:

W przypadku konieczności wznowienia działania zadania na skutek, na przykład, zmiany jego parametrów należy zadanie deaktywować i ponownie aktywować:

```
begin
    dbms_scheduler.disable ( 'JOB_DRAWING_ONE_VALUE' );
    dbms_scheduler.enable  ( 'JOB_DRAWING_ONE_VALUE' );

    dbms_scheduler.disable ( 'JOB_PERCENT_DRAWING_TABLE' );
    dbms_scheduler.enable  ( 'JOB_PERCENT_DRAWING_TABLE' );
end;
```

7. Metadane związane z zadaniami, programami i harmonogramami

Definicje wszystkich obiektów związanych z zadaniami można analizować poprzez wyświetlanie ich metadanych poniższymi zdaniami SQL:

```
select * from user_scheduler_jobs;
select * from user_scheduler_programs;
select * from user_scheduler_schedules;
```

8. Usuwanie definicji zadań, programów i harmonogramów

```
begin
    dbms_scheduler.drop_job ( 'JOB_DRAWING_ONE_VALUE' );
    dbms_scheduler.drop_job ( 'JOB_PERCENT_DRAWING_TABLE' );

    dbms_scheduler.drop_program ( 'PROG_DRAWING_ONE_VALUE' );
    dbms_scheduler.drop_program ( 'PROG_INIT_DRAWING_TABLE' );
    dbms_scheduler.drop_program ( 'PROG_PERCENT_DRAWING_TABLE' );

    dbms_scheduler.drop_schedule ( 'INTERVAL_DAILY_2200' );
    dbms_scheduler.drop_schedule ( 'INTERVAL_EVERY_5_MINUTES' );

end;
```

Uwagi:

1. Istnieje możliwość zmiany zdefiniowanych uprzednio parametrów zadań, programów i harmonogramów przy pomocy procedury pakietowej *dbms_scheduler.set_attribute*, na przykład:

```
dbms_scheduler.set_attribute
  (
    name => 'INTERVAL_EVERY_MINUTE',
    attribute => 'start_date',
    value => to_date ( '23.12.2020 12:30' , 'dd.mm.yyyy hh24:mi' )
  );
```

   lub

```
dbms_scheduler.set_attribute
  (
    name => 'INTERVAL_EVERY_MINUTE',
    attribute => 'repeat_interval',
    value => 'freq=MINUTELY;interval=2'
  );
```

2. Istnieje również możliwość przekazywania, poprzez programy i zadania, argumentów do procedur poprzez te obiekty wykonywanych. Zagadnienie to nie będzie omawiane w tym materiale.