Oracle Database 11g. Programowanie w języku PL/SQL

Autor: Michael McLaughlin
Tłumaczenie: Tomasz Walczak
ISBN: 978-83-246-1938-2
Tytuł oryginału: Oracle Database 11g PL/SQL Programming
Format: 168x237, stron: 904

Poznaj tajniki programowania w języku PL/SQL

- Jak pisać kod w języku PL/SQL?
- Jak zarządzać błędami?
- Jak tworzyć funkcje i procedury?

PL/SQL to wbudowany język proceduralny baz danych Oracle. Jest on rozszerzeniem języka SQL i umożliwia tworzenie takich konstrukcji, jak pętle, instrukcje warunkowe, zmienne i wyzwalacze. Dzięki temu można zautomatyzować wiele czynności administracyjnych oraz rejestrować zmiany danych lub nadzorować ich modyfikacje. Język ten pozwala więc na budowanie dynamicznych i stabilnych aplikacji, opartych na bazach danych typu klient-serwer.

Książka „Oracle Database 11g. Programowanie w języku PL/SQL" zawiera omówienie wszystkich najnowszych funkcji i narzędzi tego języka programowania. Szczegółowe wyjaśnienia wzbogacone zostały o studia przypadków oraz przykładowy kod, który można wkleić do własnej aplikacji. Z tym podręcznikiem nauczysz się pobierać i modyfikować informacje w bazach danych, tworzyć wartościowe instrukcje w języku PL/SQL, skutecznie wykonywać zapytania i budować niezawodne zabezpieczenia. Dowiesz się także między innymi, jak stosować procedury, funkcje, pakiety, kolekcje i wyzwalacze oraz jak zoptymalizować wydajność aplikacji.

- Język PL/SQL – architektura i funkcje
- Struktury sterujące
- Instrukcje
- Zarządzanie błędami
- Programowanie w języku PL/SQL
- Kolekcje
- Obiekty i pakiety
- Wyzwalacze
- Komunikacja między sesjami
- Podprogramy zewnętrzne
- Typy obiektowe
- Biblioteka języka Java
- Rozwój aplikacji sieciowych

Twórz solidne aplikacje sieciowe w języku PL/SQL

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

Rozdział 11.
Dynamiczny SQL

Wbudowany dynamiczny język SQL (ang. *Native Dynamic SQL* — NDS), wprowadzony w Oracle 9*i* oraz usprawniony w wersjach 10*g* i 11*g*, udostępnia wszystkie funkcje pakietu DBMS_SQL oprócz jednej. NDS to technologia przyszłości i należy rozważyć jak najszybsze zastąpienie nią kodu opartego na pakiecie DBMS_SQL. Język NDS i pakiet DBMS_SQL służą do tworzenia oraz uruchamiania instrukcji w języku SQL w czasie wykonywania programu.

Ten rozdział zawiera trzy podrozdziały:

- Architektura dynamicznego SQL-a
- Wbudowany dynamiczny język SQL (NDS)
  - Instrukcje dynamiczne
  - Instrukcje dynamiczne z danymi wejściowymi
  - Instrukcje dynamiczne z danymi wejściowymi i wyjściowymi
  - Instrukcje dynamiczne o nieznanej liczbie danych wejściowych
- Pakiet DBMS_SQL
  - Instrukcje dynamiczne
  - Instrukcje dynamiczne o zmiennych wejściowych
  - Instrukcje dynamiczne o zmiennych wejściowych i wyjściowych

Dynamiczny SQL to technologia dająca wielkie możliwości, która pozwala tworzyć i wykonywać zapytania w trakcie wykonywania programu. Oznacza to, że instrukcje DDL i DML można modyfikować oraz dostosowywać do zmieniających się potrzeb.

Architektura instrukcji dynamicznych obowiązuje zarówno w języku NDS, jak i w pakiecie DBMS_SQL. Została opisana na początku rozdziału, a Czytelnik powinien zapoznać się z nią przed przejściem do podrozdziałów poświęconych NDS-owi i pakietowi DBMS_SQL. Najpierw opisano NDS, ponieważ programista może użyć go do obsługi wszystkich operacji z wyjątkiem tych, w których nie zna liczby i typów danych wartości wyjściowych. Do zarządzania takimi instrukcjami trzeba użyć pakietu DBMS_SQL. W ostatnim podrozdziale opisano także ten pakiet, ponieważ w aplikacjach często znajduje się dużo kodu przekazywanego i konserwowanego od wielu lat.

Architektura dynamicznego SQL-a

Dynamiczny SQL zapewnia elastyczność potrzebną przy rozwiązywaniu wielu problemów. Umożliwia pisanie funkcji w stylu lambda. Takie funkcje można deklarować tak samo, jak wszystkie inne, jednak mają one nieznane listy parametrów i typy zwracanych wartości. Dynamiczny SQL pozwala tworzyć takie funkcje w języku PL/SQL.

Choć można stosować oba podejścia, warto pamiętać, że w Oracle 11*g* znajduje się usprawniona wersja języka NDS, a pakiet `DBMS_SQL` jest udostępniany głównie z uwagi na zachowanie zgodności wstecz. Obie te techniki umożliwiają tworzenie programów dynamicznych. Należy wybrać to podejście, które najbardziej odpowiada przyszłym potrzebom programisty.

W obu technikach można stosować dwa podejścia — albo łączyć ze sobą łańcuchy znaków, albo zastosować miejsca na dane. Łączenie łańcuchów znaków grozi atakami przez wstrzyknięcie kodu SQL. W tej technice napastnicy wykorzystują mechanizmy związane z łańcuchami znaków w cudzysłowach. Użycie miejsc na dane zabezpiecza przed takimi atakami. Te miejsca na dane to zmienne powiązane. Działają jak parametry formalne instrukcji dynamicznych, nie są jednak tak uporządkowane jak sygnatury funkcji i procedur.

Język NDS i pakiet `DBMS_SQL` służą do tworzenia dynamicznych instrukcji w języku SQL. W czasie kompilacji kompilator nie sprawdza poprawności elementów instrukcji dynamicznych względem obiektów z bazy danych. Umożliwia to budowanie poleceń, które będą współdziałać z komponentami tworzonymi w przyszłości lub z wieloma różnymi obiektami. Operacje wykonywane przez instrukcje dynamiczne są związane ze sposobem ich wywoływania.

Proces uruchamiania instrukcji dynamicznej obejmuje cztery etapy. Najpierw instrukcja jest parsowana w czasie wykonywania programu. W drugim kroku w instrukcjach z miejscami na dane argumenty są odwzorowywane na parametry formalne. Na trzecim etapie program wykonuje instrukcję. W czwartym kroku instrukcja dynamiczna zwraca wartość do instrukcji wywołującej. W pakiecie `DBMS_SQL` proces ten jest nieco bardziej skomplikowany. Diagram z jego opisem można znaleźć w podręczniku *Oracle Database PL/SQL Packages and Types Reference*.

Wbudowany dynamiczny język SQL (NDS)

NDS to potężne, a zarazem proste narzędzie. Łatwo używać tego języka i wdrażać oparte na nim rozwiązania. Spełnia on większość potrzeb związanych z tworzeniem funkcji w stylu lambda. Ten podrozdział zawiera trzy punkty. Pierwszy opisuje instrukcje dynamiczne składające się ze scalanych ze sobą łańcuchów znaków. Drugi pokazuje, jak używać wejściowych zmiennych powiązanych. Trzeci uczy, jak zwracać dane z instrukcji NDS.

Instrukcje dynamiczne

W tym punkcie opisano uruchamianie instrukcji dynamicznych. W czasie definiowania programu są to statyczne powłoki, których można użyć do utworzenia instrukcji w czasie wykonywania programu. Instrukcje tego rodzaju odpowiadają metodzie 1. pakietu `DBMS_SQL` (listę tych metod zawiera tabela 11.1).

Instrukcje DDL są umieszczane w dynamicznym kodzie SQL w celu uniknięcia błędów w czasie kompilacji. Dotyczy to na przykład poleceń, które program ma uruchomić, jeśli dany obiekt istnieje. Bez dynamicznych instrukcji SQL jednostka programu nie zadziała z uwagi na brak potrzebnych obiektów w bazie danych.

Przyczyny tworzenia dynamicznych instrukcji DML są inne. Zazwyczaj ich stosowanie związane jest ze sprawdzaniem informacji w bieżącej sesji przed uruchomieniem instrukcji. Na przykład programista może wczytać wartość `CLIENT_INFO` sesji, aby sprawdzić dane uwierzytelniające, role i uprawnienia użytkownika aplikacji.

W kolejnych podpunktach opisano dynamiczne instrukcje DDL i DML.

Dynamiczne instrukcje DDL

W niezależnych skryptach przed przeprowadzeniem pewnych operacji na obiekcie często trzeba sprawdzić, czy znajduje się on w bazie danych. Próba uruchomienia instrukcji `DROP` na nieistniejącej tabeli lub sekwencji spowoduje błąd.

Poniższy blok anonimowy warunkowo usuwa sekwencję. Program ten używa pętli `FOR` do sprawdzenia, czy sekwencja istnieje, a następnie tworzy i uruchamia dynamiczną instrukcję DDL.

Przed przetestowaniem kodu należy włączyć zmienną `SERVEROUTPUT` środowiska SQL*Plus, aby móc zobaczyć potwierdzenie wykonania zadania. Blok ten można z powodzeniem uruchomić kilkakrotnie niezależnie od tego, czy sekwencja `sample_sequence` istnieje, czy nie. Ten przykładowy program tworzy sekwencję, sprawdza jej dostępność w widoku `user_sequences`, a następnie uruchamia blok anonimowy. Po wykonaniu tych operacji kieruje zapytanie do wspomnianego widoku, aby potwierdzić, że sekwencja została usunięta.

```
-- Ten kod znajduje się w pliku create_nds1.sql dostępnym na witrynie wydawnictwa.
BEGIN
  -- Użycie pętli do sprawdzenia, czy trzeba usunąć sekwencję.
  FOR i IN (SELECT null
            FROM   user_objects
            WHERE  object_name = 'SAMPLE_SEQUENCE') LOOP
    EXECUTE IMMEDIATE 'DROP SEQUENCE sample_sequence';
    dbms_output.put_line('Usunięto [sample_sequence].');
  END LOOP;
END;
/
```

NDS jest prosty i bezpośredni. Wystarczy wywołać zapytanie, aby sprawdzić, czy tabela jest dostępna. Jeśli tak, można ją usunąć. Instrukcja `EXECUTE IMMEDIATE` uruchamia polecenie.

Dynamiczne instrukcje DML

Dynamiczne instrukcje DML to często łańcuchy znaków łączone w czasie wykonywania programu. Elementy łańcucha można przekazywać jako parametry funkcji i procedur. Problem z łączeniem łańcuchów znaków podawanych jako dane wejściowe polega na tym, że grozi to **atakami przez wstrzyknięcie kodu SQL**. Pakiet `DBMS_ASSERT` pozwala skontrolować parametry wejściowe pod kątem niebezpiecznego kodu.

Poniższa procedura umożliwia dynamiczne utworzenie instrukcji INSERT działającej na tabeli item:

```
-- Ten kod znajduje się w pliku create_nds2.sql dostępnym na witrynie wydawnictwa.
CREATE OR REPLACE PROCEDURE insert_item
( table_name    VARCHAR2
, asin          VARCHAR2
, item_type     VARCHAR2
, item_title    VARCHAR2
, item_subtitle VARCHAR2 := ''
, rating        VARCHAR2
, agency        VARCHAR2
, release_date  VARCHAR2 ) IS
  stmt VARCHAR2(2000);
BEGIN
  stmt := 'INSERT INTO '||dbms_assert.simple_sql_name(table_name)||' VALUES '
        || '( item_s1.nextval '
        || ','||dbms_assert.enquote_literal('ASIN'||CHR(58)||asin)
        || ',(SELECT   common_lookup_id '
        || '  FROM     common_lookup '
        || '  WHERE    common_lookup_type = '
        ||      dbms_assert.enquote_literal(item_type)||')'
        || ','||dbms_assert.enquote_literal(item_title)
        || ','||dbms_assert.enquote_literal(item_subtitle)
        || ', empty_clob() '
        || ', NULL '
        || ','||dbms_assert.enquote_literal(rating)
        || ','||dbms_assert.enquote_literal(agency)
        || ','||dbms_assert.enquote_literal(release_date)
        || ', 3, SYSDATE, 3, SYSDATE)';
       dbms_output.put_line(stmt);
  EXECUTE IMMEDIATE stmt;
END insert_item;
/
```

Nazwę tabeli item można zapisać na stałe w łańcuchu znaków, jednak tu jest parametrem, co pozwala pokazać działanie funkcji QUALIFIED_SQL_NAME. Porównuje ona łańcuchy znaków z wartościami z przestrzeni nazw schematu. Funkcja ta zgłasza błąd ORA-44004, jeśli argument jest nieprawidłowy. Funkcja ENQUOTE_LITERAL dodaje cudzysłowy wokół literałów znakowych w instrukcjach SQL. Jest to lepsze rozwiązanie niż stosowane dawniej poprzedzanie apostrofów innymi apostrofami, co wymagało użycia składni '''łańcuch znaków''' w celu utworzenia literału znakowego 'łańcuch znaków'.

Aby przetestować utworzoną wcześniej procedurę, można użyć poniższego bloku anonimowego:

```
-- Ten kod znajduje się w pliku create_nds2.sql dostępnym na witrynie wydawnictwa.
BEGIN
  insert_item (table_name => 'ITEM'
              ,asin => 'B00005O3VC'
              ,item_type => 'DVD_FULL_SCREEN'
              ,item_title => 'Monty Python and the Holy Grail'
              ,item_subtitle => 'Special Edition'
              ,rating => 'PG'
              ,agency => 'MPAA'
              ,release_date => '23-OCT-2001');
END;
/
```

Kod ten doda nowy element do tabeli item.

Ataki przez wstrzyknięcie kodu SQL

Ataki przez wstrzyknięcie kodu SQL to próby sfałszowania danych przy użyciu nieparzystych cudzysłowów w instrukcjach języka SQL. Dynamiczny kod SQL to miejsce, w którym crackerzy mogą próbować wykorzystać luki w programie.

Baza danych Oracle udostępnia pakiet DBMS_ASSERT, który pomaga zapobiegać takim atakom. Poniżej opisano funkcje tego pakietu:

- Funkcja ENQUOTE_LITERAL przyjmuje łańcuch znaków i dodaje apostrofy na jego początku oraz końcu.
- Funkcja ENQUOTE_NAME przyjmuje łańcuch znaków i przekształca go na same duże litery przed dodaniem cudzysłowów na jego początku i końcu. Ustawienie opcjonalnego parametru logicznego na false spowoduje, że funkcja nie zmieni wielkości znaków.
- Funkcja NOOP przyjmuje łańcuch znaków i zwraca tę samą wartość bez sprawdzania jej poprawności. Dostępne są przeciążone wersje tej funkcji obsługujące typy VARCHAR2 i CLOB.
- Funkcja QUALIFIED_SQL_NAME sprawdza, czy wejściowy łańcuch znaków reprezentuje poprawną nazwę obiektu schematu. Funkcja ta umożliwia walidację funkcji, procedur, pakietów i obiektów zdefiniowanych przez użytkownika. Argumenty można sprawdzać w wersjach pisanych przy użyciu małych liter, dużych liter i znaków o różnej wielkości.
- Funkcja SCHEMA_NAME sprawdza, czy wejściowy łańcuch znaków reprezentuje poprawną nazwę schematu. Aby funkcja działała prawidłowo, argument musi składać się wyłącznie z dużych liter. Dlatego przy przekazywaniu argumentu należy umieścić go w funkcji UPPER (jej opis zawiera dodatek J).
- Funkcja SIMPLE_SQL_NAME sprawdza, czy wejściowy łańcuch znaków reprezentuje poprawną nazwę obiektu schematu. Funkcja ta umożliwia walidację funkcji, procedur, pakietów i obiektów zdefiniowanych przez użytkownika.
- Funkcja SQL_OBJECT_NAME sprawdza, czy wejściowy łańcuch znaków reprezentuje poprawną nazwę obiektu schematu. Funkcja ta umożliwia walidację funkcji, procedur i pakietów. Kiedy powstawała ta książka, przy próbie sprawdzenia typu obiektowego zdefiniowanego przez użytkownika funkcja zwracała błąd ORA-44002.

Więcej informacji o pakiecie DBMS_ASSERT zawiera podręcznik *Oracle Database PL/SQL Packages and Types Reference*. Jeśli programista używa zmiennych powiązanych, zamiast łączyć łańcuchy znaków, kod NDS jest odporny na ataki przez wstrzyknięcie kodu SQL.

Instrukcje dynamiczne z danymi wejściowymi

Instrukcje dynamiczne z danymi wejściowymi wykraczają o krok poza łączenie łańcuchów znaków. Ta technika umożliwia pisanie instrukcji z miejscami na dane. Miejsca na dane działają jak parametry formalne, jednak znajdują się pomiędzy fragmentami instrukcji języka SQL. Argumenty można przekazać do instrukcji przez umieszczenie ich w klauzuli USING. Domyślnie do zwracania wartości służy klauzula RETURNING INTO.

Miejsca na dane są określane na podstawie pozycji w instrukcji języka SQL lub wywołania w języku PL/SQL. W klauzuli USING trzeba podać argumenty odpowiadające wszystkim miejscom na dane. Klauzula ta przyjmuje listę parametrów rozdzielonych przecinkami. Jeśli

programista nie zmieni ustawień, parametry będą miały tryb IN (będą **przekazywane przez wartość**). Można zmienić ten domyślny tryb działania i użyć trybu IN OUT lub IN.

Przy wykonywaniu instrukcji SQL należy używać parametrów w trybie IN. Tryb IN OUT lub OUT wymaga umieszczenia instrukcji SQL w bloku anonimowym lub użycia funkcji albo procedury języka PL/SQL. Dokumentacja bazy danych Oracle 11*g* zawiera następujące zalecenia dotyczące miejsc na dane:

- Jeśli dynamiczna instrukcja SELECT języka SQL zwraca co najwyżej jeden wiersz, należy zwracać wartość przy użyciu klauzuli INTO. Wymaga to wykonania jednej z dwóch operacji: (a) otwarcia instrukcji jako kursora referencyjnego lub (b) umieszczenia instrukcji SQL w bloku anonimowym. W tym pierwszym przypadku w klauzuli USING nie można używać parametrów w trybie IN OUT lub OUT, natomiast w drugim rozwiązaniu jest to wymagane.
- Jeśli dynamiczna instrukcja SELECT języka SQL zwraca więcej niż jeden wiersz, należy zwracać wartości przy użyciu klauzuli BULK COLLECT INTO. Klauzula ta — podobnie jak klauzula INTO — wymaga zastosowania jednego z dwóch podejść: (a) otwarcia instrukcji jako kursora referencyjnego lub (b) umieszczenia instrukcji SQL w bloku anonimowym. W tym pierwszym przypadku w klauzuli USING nie można używać parametrów w trybie IN OUT lub OUT, natomiast w drugim rozwiązaniu jest to wymagane.
- Jeśli dynamiczna instrukcja SQL to polecenie DML zawierające tylko wejściowe miejsca na dane, należy umieścić je w klauzuli USING.
- Jeśli dynamiczna instrukcja SQL to polecenie DML zawierające klauzulę RETURNING INTO, należy przekazywać zmienne wejściowe w klauzuli USING, a wartości wyjściowe — w klauzuli RETURNING INTO języka NDS.
- Jeśli dynamiczna instrukcja SQL to blok anonimowy języka PL/SQL lub instrukcja CALL, w klauzuli USING należy umieścić zarówno parametry wejściowe, jak i wyjściowe. Wszystkie parametry w tej klauzuli mają domyślnie tryb IN, dlatego programista musi czasem zmienić to ustawienie i nadać odpowiednim z nich tryb IN OUT lub OUT.

Przykładowy kod zaprezentowany w tym punkcie przedstawia wszystkie techniki związane z instrukcjami SQL i wywoływaniem bloków anonimowych języka PL/SQL. Zgodnie z praktyczną regułą należy unikać umieszczania instrukcji NDS w blokach anonimowych, ponieważ klauzula RETURNING INTO jest prostsza i bardziej wydajna.

Poniższy kod to nowa wersja procedury insert_item przedstawionej w poprzednim punkcie. To rozwiązanie opiera się na zmiennych powiązanych:

```
-- Ten kod znajduje się w pliku create_nds3.sql dostępnym na witrynie wydawnictwa.
CREATE OR REPLACE PROCEDURE insert_item
( asin          VARCHAR2
, item_type     VARCHAR2
, item_title    VARCHAR2
, item_subtitle VARCHAR2 := ''
, rating        VARCHAR2
, agency        VARCHAR2
, release_date  DATE ) IS
  stmt VARCHAR2(2000);
```

```
BEGIN
  stmt := 'INSERT INTO item VALUES '
       || '( item_s1.nextval '
       || ','''ASIN'''||CHR(58)||:asin '
       || ',(SELECT   common_lookup_id '
       || '  FROM     common_lookup '
       || '  WHERE    common_lookup_type = :item_type)'
       || ', :item_title '
       || ', :item_subtitle '
       || ', empty_clob() '
       || ', NULL '
       || ', :rating '
       || ', :agency '
       || ', :release_date '
       || ', 3, SYSDATE, 3, SYSDATE)';
EXECUTE IMMEDIATE stmt
  USING asin, item_type, item_title, item_subtitle, rating, agency, release_date;
END insert_item;
/
```

W tym kodzie można zauważyć kilka zmian. Najważniejsza z nich to usunięcie wszystkich wywołań opartych na pakiecie `DBMS_ASSERT`. Zmienne powiązane dziedziczą typ po argumentach przekazanych przy użyciu klauzuli `USING`. Dlatego wokół zmiennych, które w normalnych warunkach byłyby literałami znakowymi, nie trzeba dodawać cudzysłowów. Następna zmiana to usunięcie operacji podstawiania nazwy tabeli. Próba wykonania tej czynności spowoduje w czasie wykonywania programu błąd `ORA-00903`. Ostatnia modyfikacja to typ danych parametru `release_date`, który obecnie ma typ `DATE`.

Instrukcja `EXECUTE IMMEDIATE` używa wszystkich zmiennych przekazanych jako argumenty za pomocą klauzuli `USING` jak zmiennych w trybie `IN`. Tryb `ten` jest tu domyślny, podobnie jak w funkcjach i procedurach. Jeśli instrukcja ma zwracać zmienne do zasięgu programu lokalnego, trzeba zmienić ich tryb na `OUT`.

Przekazanie mniejszej liczby argumentów, niż jest miejsc na dane, prowadzi do błędu `ORA-01008`. Informuje on o tym, że nie wszystkie zmienne są powiązane. Klauzula `USING` zastępuje starsze procedury `BIND_VALUE` i `BIND_ARRAY` pakietu `DBMS_SQL`.

Poniższy blok anonimowy pozwala przetestować nową wersję procedury `insert_item`:

```
-- Ten kod znajduje się w pliku create_nds3.sql dostępnym na witrynie wydawnictwa.
BEGIN
  insert_item (asin => 'B00005O3VC'
               ,item_type => 'DVD_FULL_SCREEN'
               ,item_title => 'Monty Python and the Holy Grail'
               ,item_subtitle => 'Special Edition'
               ,rating => 'PG'
               ,agency => 'MPAA'
               ,release_date => '23-OCT-2001');
END;
/
```

Zmienne powiązane to zwykle lepsze rozwiązanie niż scalanie łańcuchów znaków, obie metody mają jednak określone zastosowania. Zaletą zmiennych powiązanych jest to, że zabezpieczają programistę przed atakami przez wstrzyknięcie kodu SQL.

Instrukcje dynamiczne z danymi wejściowymi i wyjściowymi

W NDS możliwość powiązania danych wejściowych jest bardzo cenna. Wartość mechanizmu pobierania zmiennych wyjściowych wynika z **prostoty** tego rozwiązania. To bardzo korzystna zmiana w porównaniu z rozwlekłymi rozwiązaniami opartymi na pakiecie DBMS_SQL, które opisano w punkcie „Instrukcje dynamiczne ze zmiennymi wejściowymi i wyjściowymi" w dalszej części rozdziału.

```
-- Ten kod znajduje się w pliku create_nds4.sql dostępnym na witrynie wydawnictwa.
DECLARE
  -- Jawna definicja struktury rekordowej.
  TYPE title_record IS RECORD
  ( item_title     VARCHAR2(60)
  , item_subtitle  VARCHAR2(60));
  -- Definicje zmiennych na potrzeby instrukcji dynamicznej.
  title_cursor  SYS_REFCURSOR;
  title_row     TITLE_RECORD;
  stmt          VARCHAR2(2000);
BEGIN
  -- Przygotowanie instrukcji.
  stmt := 'SELECT  item_title, item_subtitle '
       || 'FROM    item '
       || 'WHERE   SUBSTR(item_title,1,12) = :input';
  -- Otwarcie i wczytanie kursora dynamicznego oraz zamknięcie go.
  OPEN title_cursor FOR stmt USING 'Harry Potter';
  LOOP
    FETCH title_cursor INTO title_row;
    EXIT WHEN title_cursor%NOTFOUND;
    dbms_output.put_line(
      '['||title_row.item_title||']['||title_row.item_subtitle||']');
  END LOOP;
  CLOSE title_cursor;
END;
/
```

Ta instrukcja NDS jest dynamiczna i przyjmuje jedną wejściową zmienną powiązaną. Klauzula USING w instrukcji OPEN FOR określa kryteria filtrowania. W tym kontekście klauzula USING działa w trybie IN. Próba użycia trybu OUT spowoduje zgłoszenie przez parser wyjątku PLS-00254.

Dane wyjściowe z zapytania można zwrócić w podobny sposób, jak w dowolnej innej instrukcji z użyciem kursora referencyjnego. W rozdziale 6. znajduje się ramka opisująca systemowe kursory referencyjne.

NDS obsługuje też operacje masowe. W czasie analizy przykładów ilustrujących takie operacje można zajrzeć do punktu „Instrukcje masowe" z rozdziału 4. i przypomnieć sobie potrzebne informacje. W następnym fragmencie wystarczy wywołać instrukcję FETCH BULK COLLECT INTO:

```
-- Ten kod znajduje się w pliku create_nds5.sql dostępnym na witrynie wydawnictwa.
DECLARE
  -- Jawna definicja struktury rekordowej.
  TYPE title_record IS RECORD
  ( item_title     VARCHAR2(60)
```

```
  , item_subtitle  VARCHAR2(60));
  TYPE title_collection IS TABLE OF TITLE_RECORD;
  -- Definicje zmiennych na potrzeby instrukcji dynamicznej.
  title_cursor  SYS_REFCURSOR;
  titles        TITLE_COLLECTION;
  stmt          VARCHAR2(2000);
BEGIN
  -- Przygotowanie instrukcji.
  stmt := 'SELECT  item_title, item_subtitle '
       || 'FROM    item '
       || 'WHERE   SUBSTR(item_title,1,12) = :input';
  -- Otwarcie i wczytanie kursora dynamicznego oraz zamknięcie go.
  OPEN title_cursor FOR stmt USING 'Harry Potter';
  FETCH title_cursor BULK COLLECT INTO titles;
  FOR i IN 1..titles.COUNT LOOP
    dbms_output.put_line(
      '['||titles(i).item_title||']['||titles(i).item_subtitle||']');
  END LOOP;
  CLOSE title_cursor;
END;
/
```

Instrukcja `FETCH BULK COLLECT INTO` przenosi cały zbiór wynikowy kursora do zmiennej typu kolekcji. W szerszym zasięgu programu można zwrócić ten zbiór do innego bloku języka PL/SQL lub użyć funkcji potokowej (zobacz rozdział 6.). Aby przypomnieć sobie, jak używać wstawiania masowego do przetwarzania pobranej kolekcji, warto zajrzeć do punktu „Instrukcje FORALL" w rozdziale 4.

Ostatnie omawiane zagadnienie dotyczy używania języka NDS do obsługi zmiennych wejściowych i wyjściowych. W tym celu należy zadeklarować w klauzuli `USING` argumenty w trybie `OUT`. To podejście wymaga wykonania dwóch zadań. Trzeba umieścić instrukcję języka SQL w anonimowym bloku języka PL/SQL oraz zwrócić zmienną przy użyciu klauzuli `RETURNING INTO` w instrukcji dynamicznej.

Dwa następne skrypty wymagają dodania nowego wiersza do tabeli `item`. Poniższy blok anonimowy używa procedury `insert_item` ze skryptu *create_nds3.sql*:

```
-- Ten kod znajduje się w pliku create_nds5.sql dostępnym na witrynie wydawnictwa.
BEGIN
  insert_item (asin => 'B000G6BLWE'
              ,item_type => 'DVD_FULL_SCREEN'
              ,item_title => 'Young Frankenstein'
              ,rating => 'PG'
              ,agency => 'MPAA'
              ,release_date => '05-SEP-2006');
END;
/
```

Następny przykład ilustruje wczytywanie i zapisywanie danych przy użyciu dynamicznej instrukcji języka SQL oraz lokalizatora `CLOB`. Jest to rozwiązanie zalecane w dokumentacji bazy danych Oracle 11*g*. Ma ono kilka zalet. Po pierwsze, wszystkie wejściowe zmienne powiązane są przekazywane w klauzuli `USING`, a wszystkie wyjściowe zmienne tego rodzaju instrukcja zwraca w klauzuli `RETURNING INTO`. Po drugie, instrukcja nie wymaga utworzenia zewnętrznego bloku anonimowego języka PL/SQL.

Skrypt z zalecanym rozwiązaniem wygląda następująco:

```
-- Ten kod znajduje się w pliku create_nds6.sql dostępnym na witrynie wydawnictwa.
DECLARE
  -- Jawna definicja struktury rekordowej.
  target  CLOB;
  source  VARCHAR2(2000) := 'Klasyk w reżyserii Mela Brooksa!';
  movie   VARCHAR2(60) := 'Young Frankenstein';
  stmt    VARCHAR2(2000);
BEGIN
  -- Przygotowanie instrukcji.
  stmt := 'UPDATE  item '
       || 'SET     item_desc = empty_clob() '
       || 'WHERE   item_id = '
       || '          (SELECT item_id '
       || '           FROM   item '
       || '           WHERE  item_title = :input) '
       || 'RETURNING item_desc INTO :descriptor';
  EXECUTE IMMEDIATE stmt USING movie RETURNING INTO target;
  dbms_lob.writeappend(target,LENGTH(source),source);
  COMMIT;
END;
/
```

Program przypisuje do miejsca na dane `:input` jeden argument określony w klauzuli `USING`. Klauzula `RETURNING INTO` zwraca miejsce na dane `:descriptor` i zapisuje je w zmiennej lokalnej `target`. Jak wyjaśniono to w rozdziale 8., lokalizator typu LOB to specjalny wskaźnik na obszar roboczy. Lokalizator umożliwia wczytywanie i zapisywanie danych w zmiennych typu CLOB oraz działa jak zmienna w trybie `IN OUT`. Jest to bardzo proste i bezpośrednie podejście w porównaniu z pozostałymi możliwościami. Inna technika polega na zastąpieniu klauzuli `RETURNING INTO` parametrem w trybie `IN OUT` w klauzuli `USING`, co wymaga umieszczenia instrukcji języka SQL w bloku anonimowym języka PL/SQL.

Do zarządzania podaną instrukcją `UPDATE` można też zastosować niezależną procedurę:

```
-- Ten kod znajduje się w pliku create_nds7.sql dostępnym na witrynie wydawnictwa.
CREATE OR REPLACE PROCEDURE get_clob
( item_title_in  VARCHAR2, item_desc_out IN OUT CLOB ) IS
BEGIN
  UPDATE  item
  SET     item_desc = empty_clob()
  WHERE   item_id =
           (SELECT item_id
            FROM   item
            WHERE  item_title = item_title_in)
  RETURNING item_desc INTO item_desc_out;
END get_clob;
/
```

Po utworzeniu tej procedury można ją wywołać przy użyciu instrukcji NDS. To rozwiązanie **jest bardziej zbliżone do wywoływania kodu przy użyciu interfejsu OCI niż do instrukcji NDS**. Umożliwia dynamiczne przekazywanie parametrów wywołań i filtrowanie ich za pomocą kodu proceduralnego.

Poniższy program wywołuje utworzoną wcześniej procedurę składowaną i zapisuje nowy łańcuch znaków w kolumnie typu CLOB. Kod wywołania znajduje się w bloku anonimowym, co jest niezbędne, jeśli programista chce użyć miejsc na dane w trybie IN OUT lub OUT.

```
-- Ten kod znajduje się w pliku create_nds7.sql dostępnym na witrynie wydawnictwa.
DECLARE
  -- Jawna definicja struktury rekordowej.
  target  CLOB;
  source  VARCHAR2(2000) := 'Klasyczny film w reżyserii Mela Brooksa!';
  movie   VARCHAR2(60) := 'Young Frankenstein';
  stmt    VARCHAR2(2000);
BEGIN
  -- Przygotowanie instrukcji.
  stmt := 'BEGIN '
       || '  get_clob(:input,:output); '
       || 'END;';
  EXECUTE IMMEDIATE stmt USING movie, IN OUT target;
  dbms_lob.writeappend(target,LENGTH(source),source);
  COMMIT;
END;
/
```

Klauzula USING wiąże zmienną lokalną movie z miejscem na dane :input, a zmienną target — z miejscem na dane :output. Wywołanie niezależnej procedury powoduje zwrócenie lokalizatora typu CLOB. Ten lokalizator jest pierwszym argumentem procedury DBMS_LOB.WRITEAPPEND, która zapisuje zawartość zmiennej lokalnej source w kolumnie typu CLOB **dzięki miejscu na dane**.

Nie można zastąpić zmiennej w trybie IN OUT klauzulą RETURNING INTO. Próba wykonania tej operacji spowoduje błąd ORA-06547, informujący, że klauzuli tej można używać tylko w instrukcjach INSERT, UPDATE i DELETE.


Uwaga

Jeśli tabela zawiera więcej niż jeden wiersz spełniający podane kryteria, wystąpi błąd. Aby przetestować program, należy usunąć zbędne kopie danych.

Aby sprawdzić, czy program zapisał informacje, należy uruchomić poniższe zapytanie:

```
SELECT item_desc FROM item WHERE item_title = 'Young Frankenstein';
```

Program wyświetli następujące dane wyjściowe:

```
ITEM_DESC
--------------------
Klasyczny film w reżyserii Mela Brooksa!
```

Instrukcje dynamiczne o nieznanej liczbie danych wejściowych

W tym punkcie opisano, jak tworzyć instrukcje o nieznanej liczbie miejsc na dane. Przedstawiono tu tak zwaną 4. metodę pakietu DBMS_SQL, która umożliwia powiązanie zmiennej liczby wejściowych miejsc na dane.

Następny program pokazuje, jak utworzyć instrukcję o nieznanej liczbie danych wejściowych zwracającą określoną listę kolumn. Także przy zmiennej liście danych wyjściowych należy użyć 4. metody pakietu DBMS_SQL.

```
-- Ten kod znajduje się w pliku create_nds8.sql dostępnym na witrynie wydawnictwa.
DECLARE
  -- Jawna deklaracja struktury rekordowej i tabeli takich struktur.
  TYPE title_record IS RECORD
  ( item_title     VARCHAR2(60)
  , item_subtitle  VARCHAR2(60));
  TYPE title_table IS TABLE OF title_record;
  -- Deklaracje zmiennych na potrzeby instrukcji dynamicznej.
  title_cursor  SYS_REFCURSOR;
  title_rows    TITLE_TABLE;
  -- Deklaracje zmiennych na potrzeby pakietu DBMS_SQL.
  c             INTEGER := dbms_sql.open_cursor;
  fdbk          INTEGER;
  -- Deklaracje zmiennych lokalnych.
  counter       NUMBER := 1;
  column_names  DBMS_SQL.VARCHAR2_TABLE;
  item_ids      DBMS_SQL.NUMBER_TABLE;
  stmt          VARCHAR2(2000);
  substmt       VARCHAR2(2000) := '';
BEGIN
  -- Wyszukiwanie wierszy spełniających określone kryteria.
  FOR i IN (SELECT 'item_ids' AS column_names
            ,       item_id
            FROM    item
            WHERE   REGEXP_LIKE(item_title,'^Harry Potter')) LOOP
    column_names(counter) := counter;
    item_ids(counter) := i.item_id;
    counter := counter + 1;
  END LOOP;
  -- Dynamiczne tworzenie podzapytania.
  IF item_ids.COUNT = 1 THEN
    substmt := 'WHERE item_id IN (:item_ids)';
  ELSE
    substmt := 'WHERE item_id IN (';
    FOR i IN 1..item_ids.COUNT LOOP
      IF i = 1 THEN
        substmt := substmt ||':'||i;
      ELSE
        substmt := substmt ||',:'||i;
      END IF;
    END LOOP;
    substmt := substmt || ')';
  END IF;
  -- Przygotowanie instrukcji.
  stmt := 'SELECT  item_title, item_subtitle '
       || 'FROM    item '
       ||  substmt;
  -- Przetwarzanie instrukcji przy użyciu pakietu DBMS_SQL.
  dbms_sql.parse(c,stmt,dbms_sql.native);
  -- Wiązanie zmiennych określających nazwę i wartość.
  FOR i IN 1..item_ids.COUNT LOOP
    dbms_sql.bind_variable(c,column_names(i),item_ids(i));
  END LOOP;
  -- Wykonanie instrukcji przy użyciu pakietu DBMS_SQL.
  fdbk := dbms_sql.execute(c);
```

```
  -- Przekształcanie kursora na potrzeby instrukcji NDS.
  title_cursor := dbms_sql.to_refcursor(c);
  -- Otwarcie i wczytanie kursora dynamicznego oraz zamknięcie go.
  FETCH title_cursor BULK COLLECT INTO title_rows;
  FOR i IN 1..title_rows.COUNT LOOP
    dbms_output.put_line(
      '['||title_rows(i).item_title||']['||title_rows(i).item_subtitle||']');
  END LOOP;
  -- Zamknięcie systemowego kursora referencyjnego.
  CLOSE title_cursor;
END;
/
```

Ten program dynamicznie tworzy instrukcję `SELECT` języka SQL. Gotowe zapytanie wygląda następująco:

```
SELECT item_title, item_subtitle FROM item
WHERE item_id IN (:1,:2,:3,:4,:5,:6,:7,:8,:9,:10,:11,:12,:13,:14)
```

Pętla wiąże listę liczbowych miejsc na dane z wartościami z tablicy asocjacyjnej `item_ids`. Wywołanie funkcji `DBMS_SQL.TO_REFCURSOR` przekształca kursor z pakietu `DBMS_SQL` na zwykły systemowy kursor referencyjny o słabej kontroli typu, a także zamyka wspomniany kursor z pakietu. Jeśli programista spróbuje zamknąć kursor z pakietu `DBMS_SQL` po jego przekształceniu, wywoła błąd `ORA-29471`. Komunikat powiązany z tym błędem informuje, że program nie może uzyskać dostępu do kursora, ponieważ zasoby nie należą już do pakietu.

Po utworzeniu systemowego kursora referencyjnego wystarczy użyć standardowych metod języka NDS do masowego pobrania zbioru rekordów. Można też przekształcić kursor z powrotem — z wersji używanej w instrukcji NDS na odmianę z pakietu `DBMS_SQL`. Służy do tego funkcja `TO_CURSOR_NUMBER`.

W tym podrozdziale opisano stosowanie języka NDS. Warto zapamiętać dwie rzeczy: kod w języku NDS jest łatwy zarówno w implementacji, jak i w użyciu. W następnym podrozdziale znalazł się opis starszego i bardziej skomplikowanego pakietu `DBMS_SQL`.

Pakiet DBMS_SQL

Pakiet `DBMS_SQL` po raz pierwszy pojawił się w bazie danych Oracle 7. Umożliwił przechowywanie w bazie danych obiektów z kodem służących do dynamicznego tworzenia instrukcji w języku SQL. Była to nowatorska technika, która rozwiązywała problem sprawdzania zależności w kodzie PL/SQL. Przed udostępnieniem tego pakietu programista nie mógł zapisać instrukcji w języku SQL, dopóki nie utworzył tabeli o odpowiedniej definicji.

W bazie danych Oracle 8*i* pakiet `DBMS_SQL` wzbogacono o obsługę kolekcji. Do wersji Oracle 9*i* pakiet ten wciąż był rozwijany. Jak opisano to w punkcie „Wbudowany dynamiczny SQL (NDS)” we wcześniejszej części rozdziału, od wersji 9*i* twórcy Oracle położyli większy nacisk na NDS.

Pakiet `DBMS_SQL` udostępnia kilka przeciążonych procedur. Jeśli programista uruchomi polecenie `describe` na tym pakiecie, zobaczy kopie wszystkich tych procedur związanych z różnymi

typami. Stałe, typy, funkcje i procedury tego pakietu zostały opisane w punkcie „Definicja pakietu DBMS_SQL".

W pakiecie DBMS_SQL dostępny jest jeden ważny mechanizm, którego nie obsługuje NDS. Przy użyciu tego pakietu można zarządzać dynamicznymi instrukcjami nawet wtedy, gdy liczba i typy danych kolumn są nieznane przed uruchomieniem programu. Jest to możliwe dzięki dwóm procedurom pakietu DBMS_SQL: DESCRIBE_COLUMNS i DESCRIBE_COLUMNS2.

Podobnie jak język NDS, pakiet DBMS_SQL obsługuje łączenie łańcuchów znaków i zmienne powiązane. Jeśli Czytelnik chce przypomnieć sobie informacje o zmiennych powiązanych, znajdzie je w rozdziale 2. Przy stosowaniu pakietu DBMS_SQL — inaczej niż przy używaniu języka NDS — niezbędne jest jawne przyznawanie uprawnień.

W Oracle są cztery rodzaje dynamicznych instrukcji języka SQL. Każdy z nich związany jest z określonymi funkcjami i procedurami, co opisuje tabela 11.1.

Tabela 11.1. *Metody działania pakietu DBMS_SQL*

Metoda	Opis	Funkcje i procedury
1	Metoda 1. obsługuje statyczne instrukcje DML i DDL. Instrukcje statyczne nie mają danych wejściowych ani wyjściowych. Metoda 1. nie współdziała z instrukcjami DQL.	EXECUTE OPEN_CURSOR PARSE
2	Metoda 2. obsługuje dynamiczne instrukcje DML obejmujących zmienne powiązane. Ta metoda wymaga, aby programista określił liczbę i typy danych zmiennych powiązanych w miejscu definicji instrukcji. Metoda 2. nie współdziała z instrukcjami DQL.	BIND_ARRAY BIND_VARIABLE EXECUTE OPEN_CURSOR PARSE
3	Metoda 3. obsługuje dynamiczne instrukcje DML obejmujących zmienne powiązane. To rozwiązanie współdziała też z klauzulą RETURNING INTO, która umożliwia pobieranie kolumn i lokalizatorów typów LOB w instrukcjach DML. Ta metoda wymaga, aby programista w miejscu definicji instrukcji określił liczbę i typy danych zmiennych powiązanych. Metoda 3. współdziała z instrukcjami DQL, jeśli programista może podać liczbę i typy danych kolumn w miejscu definicji instrukcji.	BIND_ARRAY BIND_VARIABLE COLUMN_VALUE DEFINE_COLUMN EXECUTE EXECUTE_AND_FETCH FETCH_ROWS OPEN_CURSOR PARSE VARIABLE_VALUE
4	Metoda 4. obsługuje dynamiczne instrukcje DML o zmiennych powiązanych. To rozwiązanie współdziała też z klauzulą RETURNING INTO, która umożliwia pobieranie kolumn i lokalizatorów typów LOB w instrukcjach DML. Ta metoda nie wymaga znajomości liczby i typów danych zmiennych powiązanych w momencie definiowania instrukcji. Metoda 4. współdziała z instrukcjami DQL nawet wtedy, gdy programista nie zna liczby i typów danych kolumn w miejscu definicji instrukcji.	BIND_ARRAY BIND_VARIABLE COLUMN_VALUE DEFINE_COLUMN DESCRIBE_COLUMNS DESCRIBE_COLUMNS2 DESCRIBE_COLUMNS3 EXECUTE EXECUTE_AND_FETCH FETCH_ROWS OPEN_CURSOR PARSE VARIABLE_VALUE

W czterech następnych punktach opisano pakiet DBMS_SQL. Pierwsze trzy ilustrują funkcje i zastosowania dynamicznego języka SQL używanego za pomocą tego pakietu. Punkt ostatni przedstawia stałe, typy, funkcje i procedury pakietu DBMS_SQL.

Instrukcje dynamiczne

W tym punkcie opisano uruchamianie instrukcji dynamicznych. W czasie definiowania programu są one statyczne, a następnie są tworzone w czasie wykonywania programu. Instrukcje tego rodzaju są zgodne z metodą 1. z tabeli 11.1.

Programiści piszą instrukcje DDL w dynamicznym języku SQL, aby uniknąć błędów kompilacji. Przykładem jest instrukcja, którą należy wykonać tylko wtedy, gdy dany obiekt istnieje. Bez dynamicznej instrukcji języka SQL próba uruchomienia programu może zakończyć się niepowodzeniem z powodu braku obiektu w bazie danych.

Stosowanie dynamicznych instrukcji DML ma inne uzasadnienie. Zazwyczaj program przed wykonaniem takich instrukcji musi sprawdzić określone dane w bieżącej sesji, na przykład wczytać wartość CLIENT_INFO, aby ustalić dane uwierzytelniające, role i uprawnienia użytkownika aplikacji.

Instrukcje GRANT i uprawnienia w pakiecie DBMS_SQL

Pakiet DBMS_SQL należy do schematu SYS. Czasem konieczne jest wcześniejsze przyznanie uprawnień do tego schematu użytkownikowi SYSTEM. Następnie można przyznać uprawnienia poszczególnym użytkownikom, zamiast nadawać je przez role. Zwykle użytkownicy potrzebują dostępu do pakietów DBMS_SQL i DBMS_SYS_SQL.

Aby przyznać potrzebne uprawnienia z konta SYS użytkownikowi SYSTEM, należy uruchomić dwie poniższe instrukcje:

```
GRANT EXECUTE ON dbms_sys_sql TO system WITH GRANT OPTION;
GRANT EXECUTE ON dbms_sql TO system WITH GRANT OPTION;
```

Po przyznaniu odpowiednich uprawnień użytkownikowi SYSTEM można nadać je użytkownikowi plsql, co umożliwi uruchomienie przykładowych programów. Należy przyznać poniższe uprawnienia z konta użytkownika SYSTEM:

```
GRANT EXECUTE ON dbms_sys_sql TO plsql;
GRANT EXECUTE ON dbms_sql TO plsql;
```

Teraz możliwe będzie uruchomienie przedstawionych dalej skryptów, pod warunkiem że Czytelnik zainstalował przykładowy kod wypożyczalni wideo opisany we wprowadzeniu.

W następnych podpunktach opisano odpowiednio dynamiczne instrukcje DDL i DML.

Dynamiczne instrukcje DDL

Niezależne skrypty często sprawdzają, czy określony obiekt jest dostępny w bazie danych. Dopiero potem program przystępuje do wykonywania operacji na tym obiekcie. Uruchomienie instrukcji DROP na nieistniejącej tabeli lub sekwencji to zły pomysł.

Poniższy blok anonimowy ilustruje, jak warunkowo usunąć sekwencję. Program używa pętli FOR do sprawdzenia, czy sekwencja istnieje, a następnie tworzy i uruchamia dynamiczną instrukcję DDL.

Aby zobaczyć komunikat, przed przetestowaniem poniższego kodu należy włączyć zmienną SERVEROUTPUT środowiska SQL*Plus:

```
-- Ten kod znajduje się w pliku create_dbms_sql1.sql dostępnym na witrynie wydawnictwa.
DECLARE
  -- Definicje zmiennych lokalnych na potrzeby pakietu DBMS_SQL.
  c         INTEGER := dbms_sql.open_cursor;
  fdbk      INTEGER;
  stmt      VARCHAR2(2000);

BEGIN
  -- Sprawdzanie w pętli, czy należy usunąć sekwencję.
  FOR i IN (SELECT null
            FROM   user_objects
            WHERE  object_name = 'SAMPLE_SEQUENCE') LOOP
    -- Otwarcie i wczytanie kursora dynamicznego oraz zamknięcie go.
    stmt := 'DROP SEQUENCE sample_sequence';
    dbms_sql.parse(c,stmt,dbms_sql.native);
    fdbk := dbms_sql.execute(c);
    dbms_sql.close_cursor(c);
    dbms_output.put_line('Usunięto sekwencję [SAMPLE_SEQUENCE].');
  END LOOP;
END;
/
```

W bloku deklaracji zdefiniowane są trzy zmienne na potrzeby instrukcji z użyciem pakiet DBMS_SQL. Jedna przechowuje numer kursora bazy danych. Nazwa tej zmiennej to c (od angielskiego słowa *cursor*), co wynika przede wszystkim z konwencji. Programista może użyć bardziej znaczącej nazwy, ale w przykładach użyto standardowego podejścia. Wywołanie funkcji DBMS_SQL.OPEN_CURSOR powoduje zdefiniowanie zmiennej c, ale już nie jej zadeklarowanie. Następna zmienna ma nazwę fdbk (to **następny skrót**, tym razem od słowa *feedback*, czyli informacje zwrotne). Ta zmienna służy do zapisania wartości zwróconej przez funkcję DBMS_SQL.EXECUTE. Trzecia nazwa jest prawie zrozumiała — stmt to skrót od angielskiego *statment*, czyli instrukcja.

Kod w sekcji wykonawczej przypisuje prawidłową instrukcję do zmiennej stmt. Następnie funkcja DBMS_SQL.PARSE łączy numer kursora i instrukcję oraz uruchamia tę ostatnią przy użyciu mechanizmów wykonawczych bazy danych.

Aby przetestować ten program, należy utworzyć sekwencję sample_sequence:

```
CREATE SEQUENCE sample_sequence;
```

Można się upewnić, że sekwencja jest dostępna i działa, kierując zapytanie do katalogu bazy danych lub zwiększając wartość sekwencji. Poniżej przedstawiono to drugie rozwiązanie:

```
SELECT sample_sequence.nextval FROM dual;
```

Po uruchomieniu warunkowej instrukcji usuwania powinien pojawić się poniższy komunikat:

```
Usunięto sekwencję [SAMPLE_SEQUENCE]
```

Czytelnik dowiedział się, jak zaimplementować dynamiczną instrukcję DDL przy użyciu pakietu DBMS_SQL. Porównując to rozwiązanie z techniką przedstawioną w punkcie „Wbudowany dynamiczny SQL (NDS)", można się przekonać, że wersja z użyciem pakietu DBMS_SQL wymaga więcej kodu przy niewielkich lub żadnych korzyściach.

Dynamiczne instrukcje DML

Dynamiczne instrukcje DML są zwykle tworzone jako łańcuchy znaków w czasie wykonywania programu. Często przed określeniem budowy instrukcji program sprawdza stan lub działanie pewnych elementów. W tym podpunkcie opisano 1. metodę, która umożliwia stosowanie wyłącznie łańcuchów znaków i ich kombinacji.

Przykładowy kod zawiera blok, który zmienia wartości kolumn za pomocą instrukcji INSERT. Przy uwierzytelnionych użytkownikach wpisuje dane jednego rodzaju, podczas gdy przy nieuwierzytelnionych — drugiego.

Program sprawdza wartość zmiennej CLIENT_INFO sesji, a następnie określa, które dane wstawić do kolumny LAST_UPDATED_BY tabeli. W rozdziale 10. znajduje się ramka „Odczyt i zapis metadanych sesji", w której opisano ustawianie i pobieranie wartości zmiennej CLIENT_INFO.

Program sprawdza, czy wspomniana wartość jest ustawiona. Jeśli nie, wstawia liczbę –1 w kolumnie LAST_UPDATED_BY. Informuje to o nieuprawnionym użytkowniku, a warunkowe wstawianie tych danych umożliwia sprawdzenie ręcznych wpisów SQL w produkcyjnej bazie danych. Z uwagi na kompletność program powinien aktualizować obie kolumny, CREATED_BY i LAST_UPDATED_BY, to zadanie wykonuje jednak następny przykładowy kod, w którym użyto zmiennych powiązanych.

```
-- Ten kod znajduje się w pliku create_dbms_sql2.sql dostępnym na witrynie wydawnictwa.
DECLARE
  -- Definicje lokalnych zmiennych na potrzeby pakietu DBMS_SQL.
  c      INTEGER := dbms_sql.open_cursor;
  fdbk   INTEGER;
  stmt1  VARCHAR2(2000);
  stmt2  VARCHAR2(20) := '-1,SYSDATE)';
  -- Zmienna na wartość V$SESSION.CLIENT_INFO.
  client VARCHAR2(64);
BEGIN
  stmt1 := 'INSERT INTO item VALUES '
        || '( item_s1.nextval '
        || ','''ASIN'||CHR(58)||' BOOOVBJEEG'''
        || ',(SELECT   common_lookup_id '
        || '  FROM     common_lookup '
        || '  WHERE    common_lookup_type = ''DVD_WIDE_SCREEN'') '
        || ','''Ratatouille'''
        || ','''''
        || ', empty_clob() '
        || ', NULL '
        || ','''G'''
        || ','''MPAA'''
        || ','''06-NOV-2007'''
        || ', 3, SYSDATE,';
  -- Pobieranie wartości CLIENT_INFO i warunkowe dołączanie tekstu do łańcucha znaków.
  dbms_application_info.read_client_info(client);
```

```
  IF client IS NOT NULL THEN
    stmt1 := stmt1 || client || ',SYSDATE)';
  ELSE
    stmt1 := stmt1 || stmt2;
  END IF;
  -- Tworzenie, przetwarzanie i uruchamianie instrukcji SQL oraz zamknięcie kursora.
  dbms_sql.parse(c,stmt1,dbms_sql.native);
  fdbk := dbms_sql.execute(c);
  dbms_sql.close_cursor(c);
  dbms_output.put_line('Wiersze wstawiono ['||fdbk||']');
END;
/
```

Jeśli użytkownik nie ustawi wartości kolumny `CLIENT_INFO`, skrypt powinien wstawić jeden wiersz i zapisać liczbę –1 w kolumnie `LAST_UPDATED_BY`. Jak można się przekonać na podstawie budowy użytego polecenia, zapisywanie instrukcji języka SQL w zmiennych jest żmudne i wymaga starannego stosowania apostrofów. Jeśli któryś z apostrofów nie będzie miał pary, program zgłosi błąd `ORA-01756`, informujący o tym, że ograniczone łańcuchy znaków nie zostały prawidłowo zakończone.

Dwukropki w dynamicznych instrukcjach języka SQL wskazują miejsca na dane. Kiedy instrukcja `DBMS_SQL.PARSE` przetwarza łańcuch znaków z instrukcją, oznacza miejsca na dane jako docelowe powiązane miejsca na wartości. Jeśli programista nie wywoła procedury `BIND_ARRAY` lub `BIND_VARIABLE` przed wykonaniem przetwarzanej instrukcji, program przestanie działać, ponieważ nie znajdzie zmiennej powiązanej. Procedura `BIND_VARIABLE` służy do wiązania zmiennych **skalarnych**, natomiast procedura `BIND_ARRAY` — **tabel zagnieżdżonych**.

Jeśli programista chce wstawić dwukropek jako tekst, powinien zamiast tego symbolu użyć wyrażenia `CHR(58)`, ponieważ parser nie zinterpretuje go jako zmiennej powiązanej. Choć przetworzony wyjściowy łańcuch znaków będzie zawierał dwukropek, proces parsowania nie powoduje jego wstawienia.

Składnia wszystkich poleceń z użyciem pakietu `DBMS_SQL` odpowiada składni przykładowego kodu z instrukcją DDL z poprzedniego podpunktu. Czytelnik nauczył się tworzyć i implementować dynamiczne instrukcje SQL przez budowanie oraz uruchamianie warunkowo łączonych łańcuchów znaków.

Instrukcje dynamiczne o zmiennych wejściowych

W poprzednim podpunkcie opisano, jak dynamicznie połączyć łańcuchy znaków w celu utworzenia instrukcji. Jest to niewygodne rozwiązanie i — jak można się domyślić — istnieje lepsza technika. Ten punkt opisuje 2. metodę opartą na pakiecie `DBMS_SQL`. To podejście umożliwia powiązanie zmiennych w instrukcjach.

Zwykle programista zna strukturę instrukcji DML w czasie pisania bloku języka PL/SQL. Dlatego może zapisać dynamiczną instrukcję w podobny sposób, jak funkcję, czyli z wartościami wejściowymi. Takie zmienne wejściowe to w tym przypadku miejsca na dane, a nie parametry formalne. W instrukcji działają one jak zmienne powiązane, a wielu programistów nazywa je w ten sposób.

Dużo łatwiej jest napisać instrukcję DDL lub DML przy użyciu miejsc na dane, niż łącząc łańcuchy znaków. Służy do tego metoda 2. pakietu DBMS_SQL opisana w tabeli 11.1. Tabela 11.2 zawiera listę wybranych błędów, które mogą pojawić się przy używaniu miejsc na dane i zmiennych powiązanych.

Tabela 11.2. *Błędy występujące przy stosowaniu pakietu DBMS_SQL*

Kod błędu	Opis i rozwiązanie
ORA-00928	Błąd ten wynika z umieszczenia miejsca na dane w sygnaturze przesłaniającej instrukcji INSERT. Sygnatura to lista parametrów formalnych między nazwą tabeli a klauzulą VALUES. Ogólny komunikat informujący o braku słowa kluczowego SELECT może być mylący.
ORA-06502	Ten błąd pojawia się wtedy, kiedy trzeba jawnie określić rozmiar zmiennych typów CHAR, RAW lub VARCHAR2. W wywołaniach procedur BIND_VARIABLE i BIND_VARIABLE_ROW należy wtedy podać rozmiar danych wyjściowych. Ogólny komunikat informujący o błędnej liczbie lub wartości w kodzie PL/SQL może być mylący.
ORA-01006	Błąd ORA-01006 wynika z umieszczenia miejsc na dane typu VARCHAR2 w cudzysłowach. Funkcja BIND_VARIABLE wiąże wartość i typ danych z instrukcją, dlatego cudzysłowy można pominąć. Ogólny komunikat informujący o tym, że zmienna powiązana nie istnieje, jest zupełnie niewłaściwy, jednak Czytelnik wie już, jak rozwiązać problem.
PLS-00049	Ten błąd to wynik zapisania w miejscu na dane wartości o nieoczekiwanym typie danych, którego nie można niejawnie przekształcić na typ docelowy. Trzeba się upewnić, że w przypisaniach bezpośrednio użyto odpowiednich typów danych. Jeśli programista nie polega na niejawnej konwersji, nigdy się nie rozczaruje. Komunikat informujący o nieprawidłowej zmiennej powiązanej nie jest jasny, ale uzasadniony, ponieważ program użył złego typu danych.

Warto zauważyć, że za pomocą pakietu DBMS_SQL można utworzyć blok języka PL/SQL. Jedyny warunek jest taki, aby zakończyć łańcuch znaków średnikiem. To odstępstwo od działania zwykłych instrukcji języka SQL. Owa różnica wynika z tego, że blok języka PL/SQL musi być zakończony zamykającym średnikiem. W instrukcjach języka SQL średnik to polecenie wykonania kodu. Przykład zastosowania tego podejścia przedstawiono w kolejnym punkcie — „Instrukcje dynamiczne o zmiennych wejściowych i wyjściowych".

Następny program zawiera nową wersję instrukcji INSERT z poprzedniego punktu. Tym razem w kodzie użyto zmiennych podstawianych. Ten blok to niezależna procedura, której można użyć do wstawienia nowych elementów do tabeli item.

Poniżej przedstawiono niezależną procedurę z miejscami na dane (zmiennymi powiązanymi) w trybie IN:

```
-- Ten kod znajduje się w pliku create_dbms_sql3.sql dostępnym na witrynie wydawnictwa.
CREATE OR REPLACE PROCEDURE insert_item
( asin     VARCHAR2
, title    VARCHAR2
, subtitle VARCHAR2 := NULL
, itype    VARCHAR2 := 'DVD_WIDE_SCREEN'
, rating   VARCHAR2
, agency   VARCHAR2
, release  DATE ) IS
  -- Definicje zmiennych lokalnych na potrzeby pakietu DBMS_SQL.
  c      INTEGER := dbms_sql.open_cursor;
  fdbk   INTEGER;
```

```
  stmt   VARCHAR2(2000);
  -- Zmienna na wartość parametru w trybie OUT.
  client VARCHAR2(64);
BEGIN
  stmt := 'INSERT INTO item VALUES '
        || '( item_s1.nextval '
        || ','''ASIN'''||CHR(58)|| :asin'
        || ',(SELECT   common_lookup_id '
        || '  FROM     common_lookup '
        || '  WHERE    common_lookup_type = :itype) '
        || ',:title'
        || ',:subtitle'
        || ', empty_clob() '
        || ', NULL '
        || ',:rating'
        || ',:agency'
        || ',:release'
        || ',:created_by,SYSDATE,:last_updated_by,SYSDATE)';
  -- Wywołanie i dynamiczne ustawienie wartości CLIENT_INFO sesji.
  dbms_application_info.read_client_info(client);
  IF client IS NOT NULL THEN
    client := TO_NUMBER(client);
  ELSE
    client := -1;
  END IF;
  -- Przetworzenie i wykonanie instrukcji.
  dbms_sql.parse(c,stmt,dbms_sql.native);
  dbms_sql.bind_variable(c,'asin',asin);
  dbms_sql.bind_variable(c,'itype',itype);
  dbms_sql.bind_variable(c,'title',title);
  dbms_sql.bind_variable(c,'subtitle',subtitle);
  dbms_sql.bind_variable(c,'rating',rating);
  dbms_sql.bind_variable(c,'agency',agency);
  dbms_sql.bind_variable(c,'release',release);
  dbms_sql.bind_variable(c,'created_by',client);
  dbms_sql.bind_variable(c,'last_updated_by',client);
  fdbk := dbms_sql.execute(c);
  dbms_sql.close_cursor(c);
  dbms_output.put_line('Wstawiono wiersze ['||fdbk||']');
END insert_item;
/
```

W kodzie miejsca na dane w dynamicznej instrukcji `INSERT` zostały wyróżnione pogrubieniem. Warto zauważyć, że nie ma wokół nich apostrofów. Jest tak, ponieważ wartość i typ danych są powiązane z instrukcją, dlatego ograniczniki stały się zbędne. Jeśli programista zapomni o tym i dołączy wewnętrzne apostrofy, w czasie wykonywania programu wystąpi błąd `ORA-01006`. Należy wtedy usunąć apostrofy lub umieścić instrukcję w bloku języka PL/SQL.

Wraz ze wzrostem liczby zmiennych powiązanych rośnie też liczba wywołań procedury `BIND_VARIABLE`. W tym punkcie opisano drugą metodę tworzenia dynamicznych instrukcji SQL, która umożliwia podstawianie zmiennych wejściowych.

Instrukcje dynamiczne o zmiennych wejściowych i wyjściowych

W tym punkcie opisano, jak tworzyć miejsca na dane wejściowe i wyjściowe w instrukcjach SQL. Omówiono tu trzecią metodę opartą na pakiecie DBMS_SQL. Umożliwia ona stosowanie zmiennych powiązanych w trybie IN i OUT w instrukcjach języka SQL.

Wskazówki diagnostyczne związane z instrukcjami SELECT i pakietem DBMS_SQL

Przy używaniu w procedurze DBMS_SQL.DEFINE_COLUMN skalarnych łańcuchów znaków o zmiennej długości trzeba podać fizyczny rozmiar łańcucha. Należy to zrobić także przy zwracaniu skalarnych wartości typu RAW. Jeśli programista o tym zapomni, pakiet DBMS_SQL wywoła błąd PLS-00307. Informuje on o tym, że wywołaniu odpowiada zbyt wiele deklaracji DEFINE_COLUMN. Ten błąd jest skomplikowany, ponieważ wiąże się z działaniem niejawnego rzutowania przy wywoływaniu wspomnianej procedury.

Aby ułatwić sobie pracę, programiści mogą po prostu podać czwarty parametr, który określa długość danych typu CHAR, RAW i VARCHAR2.

W trzeciej metodzie można używać dynamicznych instrukcji SELECT, jeśli w czasie kompilacji wiadomo, ile kolumn ma pobierać to polecenie. W tym punkcie opisano obsługę zbioru zwracanych wartości skalarnych i jednej skalarnej wartości wejściowej.

Przy zarządzaniu skalarnymi wartościami wyjściowymi zapytania są wykonywane **wiersz po wierszu**. Jeśli program zwraca w instrukcji SELECT wiele kolumn i zapisuje je w tablicy asocjacyjnej za pomocą przetwarzania **masowego**, stosowana jest technika **tablic równoległych**. Trzeba bardzo starannie zarządzać poruszaniem się po tych strukturach, aby zapewnić synchronizację wartości indeksów. Jeśli programista popełni błąd, program będzie pobierał wartości z różnych kolumn.

Opisana tu składnia jest jedną z najbardziej żmudnych przy stosowaniu pakietu DBMS_SQL i to zarówno przy jednej wartości oraz wykonywaniu instrukcji wiersz po wierszu, jak i przy stosowaniu instrukcji masowych. Warto zastanowić się nad użyciem w zamian prostszej klauzuli OPEN FOR z języka NDS.

Opisy przetwarzania **wiersz po wierszu** i **masowego** znalazły się w odrębnych podpunktach.

Przetwarzanie instrukcji wiersz po wierszu

Ten przykładowy program pokazuje, jak przetwarzać dane z jednego wiersza i wielu wierszy zwracane za pomocą dynamicznej instrukcji SELECT. Poniższe programy wymagają dostępu do tabeli item, którą można utworzyć przy użyciu skryptu *create_store.sql* opisanego we wprowadzeniu do książki.

Instrukcja zwracająca jeden wiersz wygląda następująco:

```
-- Ten kod znajduje się w pliku create_dbms_sql4.sql dostępnym na witrynie wydawnictwa.
DECLARE
  c                       INTEGER := dbms_sql.open_cursor;
  fdbk                    INTEGER;
```

```
  statement                VARCHAR2(2000);
  item_id                  NUMBER := 1081;
  item_title               VARCHAR2(60);
  item_subtitle            VARCHAR2(60);
BEGIN
  -- Tworzenie i przetwarzanie instrukcji SQL.
  statement := 'SELECT item_title, item_subtitle '
            || 'FROM item WHERE item_id = :item_id';
  dbms_sql.parse(c,statement,dbms_sql.native);
  -- Odwzorowywanie kolumn, wykonywanie instrukcji i kopiowanie wyników.
  dbms_sql.define_column(c,1,item_title,60);     -- Definicja zmiennej w trybie OUT.
  dbms_sql.define_column(c,2,item_subtitle,60); -- Definicja zmiennej w trybie OUT.
  dbms_sql.bind_variable(c,'item_id',item_id);  -- Zmienna powiązana o trybie IN.
  fdbk := dbms_sql.execute_and_fetch(c);
  dbms_sql.column_value(c,1,item_title);        -- Kopiowanie kolumny z zapytania do zmiennej.
  dbms_sql.column_value(c,2,item_subtitle);     -- Kopiowanie kolumny z zapytania do zmiennej.
  -- Wyświetlanie zwracanej wartości i zamykanie kursora.
  dbms_output.put_line('['||item_title||']['||NVL(item_subtitle,'None')||']');
  dbms_sql.close_cursor(c);
END;
/
```

To podejście umożliwia bezpośrednie użycie kolumn z instrukcji `SELECT`, ponieważ odpowiadają im zmienne w trybie `OUT`. Te kolumny trzeba zdefiniować przed wykonaniem instrukcji, a po pobraniu danych można skopiować je do zmiennych lokalnych. Program wskazuje kolumny przy użyciu pozycji, a zmienne lokalne — za pomocą nazw. Inaczej obsługiwane są zmienne w trybie `IN`, przy których trzeba użyć dwukropka, aby określić je jako zmienne podstawiane (**zmienne powiązane**).

To zapytanie powinno zwrócić następujące dane:

```
[We Were Soldiers][None]
```

Czytelnik wie już, jak zwrócić jeden wiersz, zwykle jednak trzeba pobrać więcej danych. Następny program wykonuje zapytanie wiersz po wierszu i wyświetla zwrócone dane wyjściowe:

```
-- Ten kod znajduje się w pliku dbms_sql5.sql dostępnym na witrynie wydawnictwa.
DECLARE
  c                        INTEGER := dbms_sql.open_cursor;
  fdbk                     INTEGER;
  statement                VARCHAR2(2000);
  item1                    NUMBER := 1003;
  item2                    NUMBER := 1013;
  item_title               VARCHAR2(60);
  item_subtitle            VARCHAR2(60);
BEGIN
  -- Tworzenie i przetwarzanie instrukcji SQL.
  statement := 'SELECT item_title, item_subtitle '
            || 'FROM item '
            || 'WHERE item_id BETWEEN :item1 AND :item2 '
            || 'AND item_type = 1014';
  dbms_sql.parse(c,statement,dbms_sql.native);
  -- Odwzorowywanie kolumn i wykonywanie instrukcji.
  dbms_sql.define_column(c,1,item_title,60);     -- Definicja zmiennej w trybie OUT.
  dbms_sql.define_column(c,2,item_subtitle,60); -- Definicja zmiennej w trybie OUT.
  dbms_sql.bind_variable(c,'item1',item1);       -- Zmienna powiązana o trybie IN.
  dbms_sql.bind_variable(c,'item2',item2);       -- Zmienna powiązana o trybie IN.
```

```
  fdbk := dbms_sql.execute(c);
  -- Wczytywanie wyników.
  LOOP
    EXIT WHEN dbms_sql.fetch_rows(c) = 0;        -- Brak dalszych wyników.
    -- Kopiowanie i wyświetlanie.
    dbms_sql.column_value(c,1,item_title);       -- Kopiowanie wartości kolumny z zapytania
                                                 -- do zmiennej.
    dbms_sql.column_value(c,2,item_subtitle);    -- Kopiowanie wartości kolumny z zapytania
                                                 -- do zmiennej.
    dbms_output.put_line('['||item_title||']['||NVL(item_subtitle,'None')||']');
  END LOOP;
  dbms_sql.close_cursor(c);
END;
/
```

Trzeba zdefiniować odwzorowanie każdej kolumny i powiązać odpowiednie zmienne. Aby można było przetworzyć dane, program w pętli kopiuje wartość kolumny każdego wiersza do zmiennej lokalnej.

Jeśli programista ustawił zmienną `SERVEROUTPUT` środowiska SQL*Plus, program powinien wyświetlić następujące dane:

```
[Casino Royale][None]
[Die Another Day][None]
[Die Another Day][2-Disc Ultimate Version]
[Golden Eye][Special Edition]
[Golden Eye][None]
[Tomorrow Never Dies][None]
[Tomorrow Never Dies][Special Edition]
[The World Is Not Enough][2-Disc Ultimate Edition]
[The World Is Not Enough][None]
```

Czytelnik dowiedział się, jak przetwarzać dane zwracane przez instrukcję `SELECT` w postaci **jednego wiersza** i **wielu wierszy**. W następnym podpunkcie opisano zarządzanie masowymi operacjami `SELECT`.

Przetwarzanie instrukcji masowych

Przetwarzanie masowe to często lepsze rozwiązanie niż wykonywanie instrukcji wiersz po wierszu. Do obsługi tej techniki należy użyć języka NDS, a nie pakietu `DBMS_SQL`. Klauzula `BULK COLLECT INTO` działa tylko w kontekście bloków języka PL/SQL. Proces wiązania masowego przy użyciu pakietu `DBMS_SQL` nie umożliwia obsługi instrukcji SQL w blokach anonimowych. Próba zastosowania tego nieobsługiwanego obejścia spowoduje błąd `PLS-00497`.

Definicja pakietu DBMS_SQL

Pakiet `DBMS_SQL` jest dostępny od wersji Oracle 7. Zmiany i poprawki sprawiły, że jest to stabilny i niezawodny komponent bazy danych. Mimo wprowadzenia wbudowanego dynamicznego języka SQL (języka NDS) w Oracle 9*i*, pakiet `DBMS_SQL` wciąż cieszy się popularnością. W Oracle 11*g* jedynym zadaniem, którego nie można wykonać za pomocą języka NDS, jest wykonywanie instrukcji używających nieznanego zbioru kolumn. Do zarządzania takimi instrukcjami służy pakiet `DBMS_SQL`.

W tym punkcie opisano stałe, zmienne, funkcje i procedury pakietu DBMS_SQL. Aby zapoznać się z definicją określonego komponentu, należy przejść do odpowiedniego podpunktu.

Stałe pakietu DBMS_SQL

Omawiany pakiet ma trzy stałe przeznaczone do obsługi procedury DBMS_SQL.PARSE. Od wersji Oracle 8 należy używać tylko stałej NATIVE. Opis tych stałych zawiera tabela 11.3.

Tabela 11.3. *Stałe pakietu DBMS_SQL*

Nazwa stałej	Opis	Wartość
NATIVE	Od wersji Oracle 8 należy używać tylko tej stałej. Ma ona typ INTEGER i określa język parsowania.	1
V6	Stałej tej nie należy używać.	0
V7	Tej stałej można używać tylko przy stosowaniu wersji Oracle 7, która nie jest już objęta pomocą techniczną ze strony firmy Oracle.	2

Typy danych pakietu DBMS_SQL

Pakiet DBMS_SQL obsługuje tablice asocjacyjne (wcześniej nazywane **tabelami języka PL/SQL**) indeksowane binarnymi liczbami całkowitymi i przechowujące wartości następujących typów danych: BFILE, BINARY_DOUBLE, BLOB, CLOB, DATE, INTERVAL_DAY_TO_SECOND, INTERVAL_DAY_TO_
↪MONTH, NUMBER, TIME, TIMESTAMP, TIMESTAMP_WITH_LTZ i UROWID. Nazwy typów danych opartych na tablicy asocjacyjnej są zgodne ze wzorcem <*typ_skalarny*>_TABLE. W podręczniku *Oracle Database PL/SQL Packages and Types References* typy te zostały opisane jako masowe typy SQL (ang. *Bulk SQL Types*).

Typ danych DBMS_SQL.VARCHAR2_TABLE został omówiony w tym samym podręczniku, co typ ogólny, i działa w podobny sposób jak masowe typy danych.

Pakiet DBMS_SQL obsługuje także trzy struktury rekordowe:

- Struktura desc_rec współdziała z procedurą DESCRIBE_COLUMNS, która używa tego typu do wyświetlania danych o kolumnach kursora otworzonego i przetwarzanego przez pakiet DBMS_SQL.

```
TYPE desc_rec IS RECORD ( col_type             BINARY_INTEGER := 0
,                         col_max_len          BINARY_INTEGER := 0
,                         col_name             VARCHAR2(32)   := ''
,                         col_name_len         BINARY_INTEGER := 0
,                         col_schema_name      VARCHAR2(32)   := ''
,                         col_schema_name_len  BINARY_INTEGER := 0
,                         col_precision        BINARY_INTEGER := 0
,                         col_scale            BINARY_INTEGER := 0
,                         col_charsetid        BINARY_INTEGER := 0
,                         col_charsetform      BINARY_INTEGER := 0
,                         col_null_ok          BOOLEAN        := TRUE);
```

- Struktura desc_rec2 współdziała z procedurą DESCRIBE_COLUMNS2, która używa tego typu do wyświetlania danych o kolumnach kursora otworzonego i przetwarzanego przez pakiet DBMS_SQL.

```
TYPE desc_rec IS RECORD ( col_type            BINARY_INTEGER  := 0
,                         col_max_len         BINARY_INTEGER  := 0
,                         col_name            VARCHAR2(32767) := ''
,                         col_name_len        BINARY_INTEGER  := 0
,                         col_schema_name     VARCHAR2(32)    := ''
,                         col_schema_name_len BINARY_INTEGER  := 0
,                         col_precision       BINARY_INTEGER  := 0
,                         col_scale           BINARY_INTEGER  := 0
,                         col_charsetid       BINARY_INTEGER  := 0
,                         col_charsetform     BINARY_INTEGER  := 0
,                         col_null_ok         BOOLEAN         := TRUE);
```

- Struktura `desc_rec3` współdziała z procedurą `DESCRIBE_COLUMNS3`, która używa tego typu do wyświetlania danych o kolumnach kursora otworzonego i przetwarzanego przez pakiet `DBMS_SQL`.

```
TYPE desc_rec IS RECORD ( col_type            BINARY_INTEGER  := 0
,                         col_max_len         BINARY_INTEGER  := 0
,                         col_name            VARCHAR2(32767) := ''
,                         col_name_len        BINARY_INTEGER  := 0
,                         col_schema_name     VARCHAR2(32)    := ''
,                         col_schema_name_len BINARY_INTEGER  := 0
,                         col_precision       BINARY_INTEGER  := 0
,                         col_scale           BINARY_INTEGER  := 0
,                         col_charsetid       BINARY_INTEGER  := 0
,                         col_charsetform     BINARY_INTEGER  := 0
,                         col_null_ok         BOOLEAN         := TRUE
,                         col_type_name       VARCHAR2(32)    := ''
,                         col_type_name_len   BINARY_INTEGER  := 0);
```

Ponadto istnieją tablice asocjacyjne dla każdego typu rekordowego. Te struktury rekordowe i tablice asocjacyjne są przydatne w 4. metodzie przetwarzania, która wymaga obsługi nieznanego w czasie kompilacji zbioru kolumn.

Funkcje i procedury pakietu DBMS_SQL

Funkcje i procedury pakietu `DBMS_SQL` pozostają takie same od wielu lat. Wciąż są powszechnie używane, choć prawie wszystkie operacje można wykonać także przy użyciu języka NDS. Taki stan rzeczy wynika po części z chęci zachowania zgodności wstecz i standardów pisania kodu, co nie sprzyja zmianom. Oracle 11*g* to następny krok w kierunku uznania w przyszłości pakietu `DBMS_SQL` za przestarzały.

Następne podpunkty powinny pomóc w szybkim zapoznaniu się z funkcjami i procedurami pakietu `DBMS_SQL` w celu konserwacji kodu lub zastąpienia tych elementów instrukcjami NDS. Jeśli programista natrafi na problemy związane z uprawnieniami, powinien zajrzeć do ramki „Instrukcje GRANT i uprawnienia w pakiecie DBMS_SQL” we wcześniejszej części rozdziału.

Procedura BIND_ARRAY

Procedura `BIND_ARRAY` służy do obsługi masowych operacji DML. Procedura ta wiąże kolekcję w postaci tabeli zagnieżdżonej z instrukcją języka SQL. Kolekcja może przechowywać dane typu z listy typów bazowych języka SQL. Procedura ta jest przeciążona i ma dwie sygnatury. Tryb wszystkich jej parametrów to `IN`.

Prototyp 1

```
bind_array( numer_kursora NUMBER
          , nazwa_kolumny VARCHAR2
          , kolekcja      <lista_typów_danych> )
```

Prototyp 2

```
bind_array( numer_kursora NUMBER
          , nazwa_kolumny VARCHAR2
          , kolekcja      <lista_typów_danych>
          , indeks1       NUMBER
          , indeks2       NUMBER )
```

Kolekcja to tablica asocjacyjna indeksowana numerami typu BINARY_INTEGER. Bazowym skalarnym typem danych może być: BFILE, BLOB, CLOB, DATE, NUMBER, ROWID, TIME, TIMESTAMP, TIME WITH TIME ZONE lub VARCHAR2. Dzięki bibliotekom OCI procedura ta może obsługiwać także tabele zagnieżdżone, tablice VARRAY i typy obiektowe zdefiniowane przez użytkownika.

Procedura BIND_VARIABLE

Procedura BIND_VARIABLE służy do obsługi operacji DML wykonywanych wiersz po wierszu. Ta procedura wiąże wartości różnych typów danych w instrukcjach języka SQL. Jest to procedura przeciążona o sygnaturze jednego rodzaju. Wszystkie jej parametry działają w trybie IN.

Prototyp

```
bind_variable( numer_kursora    NUMBER
             , nazwa_kolumny    VARCHAR2
             , wartość_zmiennej <lista_typów_danych> )
```

Lista typów danych obejmuje następujące typy języka SQL: BFILE, BINARY_DOUBLE, BINARY_FLOAT, BLOB, CLOB, DATE, INTERVAL, YEAR TO MONTH, INTERVAL YEAR TO SECOND, NUMBER, REF OF STANDARD, ROWID, TIME, TIMESTAMP, TIMESTAMP WITH TIME ZONE, TIME WITH TIME ZONE i VARCHAR2.

Procedura BIND_VARIABLE_CHAR

Procedura BIND_VARIABLE_CHAR obsługuje operacje DML wykonywane wiersz po wierszu. Procedura ta wiąże zmienne typu CHAR w instrukcjach języka SQL. Jest to procedura przeciążona o dwóch sygnaturach. Wszystkie jej parametry mają tryb IN.

Prototyp 1

```
bind_variable_char( numer_kursora    NUMBER
                  , nazwa_kolumny    VARCHAR2
                  , wartość_zmiennej CHAR )
```

Prototyp 2

```
bind_variable_char( numer_kursora        NUMBER
                  , nazwa_kolumny        VARCHAR2
                  , wartość_zmiennej     CHAR
                  , rozmiar_zmiennej_wyj NUMBER )
```

Procedura BIND_VARIABLE_RAW

Procedura BIND_VARIABLE_RAW obsługuje operacje DML wykonywane wiersz po wierszu. Procedura ta wiąże zmienną typu RAW w instrukcjach języka SQL. Jest to procedura przeciążona o dwóch sygnaturach. Wszystkie jej parametry mają tryb IN.

Prototyp 1

```
bind_variable_raw( numer_kursora    NUMBER
                 , nazwa_kolumny    VARCHAR2
                 , wartość_zmiennej RAW )
```

Prototyp 2

```
bind_variable_raw( numer_kursora        NUMBER
                 , nazwa_kolumny        VARCHAR2
                 , wartość_zmiennej     RAW
                 , rozmiar_zmiennej_wyj NUMBER )
```

Procedura BIND_VARIABLE_ROWID

Procedura BIND_VARIABLE_ROWID obsługuje operacje DML wykonywane wiersz po wierszu. Procedura ta wiąże zmienną typu ROWID w instrukcjach języka SQL. Nie jest przeciążona i ma tylko jedną sygnaturę. Wszystkie jej parametry mają tryb IN.

Prototyp

```
bind_variable_rowid( numer_kursora    NUMBER
                   , nazwa_kolumny    VARCHAR2
                   , wartość_zmiennej ROWID )
```

Procedura CLOSE_CURSOR

Procedura CLOSE_CURSOR zamyka otwarty kursor pakietu DBMS_SQL. Nie jest przeciążona i ma tylko jedną sygnaturę. Numer kursora jest przekazywany przez referencję jako zmienna w trybie IN OUT.

Prototyp

```
close_cursor( numer_kursora NUMBER )
```

Procedura COLUMN_VALUE

Procedura COLUMN_VALUE obsługuje operacje DQL wykonywane masowo i wiersz po wierszu. Ta procedura wiąże dane wyjściowe instrukcji SELECT ze zmienną o trybie OUT. Może to być zmienna skalarna lub tabela zagnieżdżona przechowująca wartości skalarne. Nazwa kursora i pozycja to zmienne o trybie IN, natomiast wartość zmiennej lub kolekcji, błąd kolumny i długość to zmienne o trybie OUT. Procedura ta ma trzy przeciążone sygnatury.

Prototyp 1

```
column_value( numer_kursora    NUMBER
            , pozycja          NUMBER
            , wartość_zmiennej <lista_typów_danych> )
```

Prototyp 2

```
column_value( numer_kursora NUMBER
            , pozycja       NUMBER
            , kolekcja      <lista_typów_danych> )
```

Prototyp 3

```
column_value( numer_kursora  NUMBER
            , pozycja        NUMBER
            , kolekcja       <lista_typów_danych>
            , błąd_kolumny   NUMBER
            , długość        NUMBER )
```

Typem danych może być typ **skalarny** lub **tablica asocjacyjna**. Obsługiwane są tu następujące typy języka SQL: `BFILE`, `BLOB`, `CLOB`, `DATE`, `NUMBER`, `ROWID`, `TIME`, `TIMESTAMP`, `TIME WITH TIME ZONE` i `VARCHAR2`.

W prototypie o pięciu parametrach można używać **tablic asocjacyjnych** przechowujących wartości skalarne typu `DATE`, `NUMBER` i `VARCHAR2`. Dzięki bibliotekom OCI procedura ta obsługuje także tablice asocjacyjne, **tabele zagnieżdżone**, tablice `VARRAY` i typy obiektowe zdefiniowane przez użytkownika.

Procedura COLUMN_VALUE_CHAR

Procedura `COLUMN_VALUE_CHAR` obsługuje operacje DQL wykonywane wiersz po wierszu. Funkcja ta wiąże dane wyjściowe instrukcji `SELECT` wykonywanej na kolumnie typu `CHAR` ze zmienną o trybie `OUT`. Jest to procedura przeciążona o dwóch sygnaturach.

Prototyp 1

```
column_value( numer_kursora    NUMBER
            , pozycja          NUMBER
            , wartość_zmiennej CHAR )
```

Prototyp 2

```
column_value( numer_kursora    NUMBER
            , pozycja          NUMBER
            , wartość_zmiennej CHAR
            , błąd_kolumny     NUMBER
            , długość          NUMBER )
```

Procedura COLUMN_VALUE_LONG

Procedura `COLUMN_VALUE_LONG` obsługuje operacje DQL przeprowadzane wiersz po wierszu. Wiąże ona dane wyjściowe z instrukcji `SELECT` wykonywanej na kolumnie typu `LONG` ze zmienną o trybie `OUT`. Nie jest przeciążona i ma tylko jedną sygnaturę.

Prototyp

```
column_value( numer_kursora    NUMBER
            , pozycja          NUMBER
            , wartość_zmiennej LONG )
```

Procedura COLUMN_VALUE_RAW

Procedura COLUMN_VALUE_RAW obsługuje operacje DQL przeprowadzane wiersz po wierszu. Procedura ta wiąże dane wyjściowe z instrukcji SELECT wykonywanej na kolumnie typu RAW ze zmienną o trybie OUT. Jest to procedura przeciążona o dwóch sygnaturach.

Prototyp 1

```
column_value_raw( numer_kursora    NUMBER
                , pozycja          NUMBER
                , wartość_zmiennej RAW )
```

Prototyp 2

```
column_value_raw( numer_kursora    NUMBER
                , pozycja          NUMBER
                , wartość_zmiennej RAW
                , błąd_kolumny     NUMBER
                , długość          NUMBER )
```

Procedura COLUMN_VALUE_ROWID

Procedura COLUMN_VALUE_ROWID obsługuje operacje DQL przeprowadzane wiersz po wierszu. Procedura ta wiąże dane wyjściowe z instrukcji SELECT wykonywanej na kolumnie typu ROWID ze zmienną o trybie OUT. Jest to procedura przeciążona o dwóch sygnaturach.

Prototyp 1

```
column_value_rowid( numer_kursora    NUMBER
                  , pozycja          NUMBER
                  , wartość_zmiennej ROWID )
```

Prototyp 2

```
column_value_rowid( numer_kursora    NUMBER
                  , pozycja          NUMBER
                  , wartość_zmiennej ROWID
                  , błąd_kolumny     NUMBER
                  , długość          NUMBER )
```

Procedura DEFINE_ARRAY

Procedura DEFINE_ARRAY obsługuje masowe operacje DQL i odwzorowuje tabelę zagnieżdżoną na kolumny używane w instrukcji SELECT. Metody tej trzeba użyć przed wywołaniem procedury COLUMN_VALUE. Procedura DEFINE_ARRAY jest przeciążona i ma sygnaturę jednego rodzaju.

Prototyp

```
define_array( numer_kursora      NUMBER
            , pozycja            NUMBER
            , kolekcja           <lista_typów_danych>
            , liczba             NUMBER
            , dolne_ograniczenie NUMBER )
```

Parametr `liczba` określa maksymalną liczbę zwracanych elementów, a parametr `dolne_ograniczenie` wyznacza punkt początkowy, którym zwykle jest 1.

Typem danych może być **tablica asocjacyjna** przechowująca wartości jednego z typów języka SQL: `BFILE`, `BLOB`, `CLOB`, `DATE`, `NUMBER`, `ROWID`, `TIME`, `TIMESTAMP`, `TIME WITH TIME ZONE` lub `VARCHAR2`.

Procedura DEFINE_COLUMN

Procedura `DEFINE_COLUMN` obsługuje operacje DQL przeprowadzane wiersz po wierszu i definiuje kolumny używane w instrukcji `SELECT`. Metody tej trzeba użyć przed wywołaniem procedury `COLUMN_VALUE`. Procedura `DEFINE_COLUMN` jest przeciążona i ma sygnaturę jednego rodzaju.

Prototyp

```
define_column( numer_kursora    NUMBER
             , pozycja          NUMBER
             , wartość_zmiennej <lista_typów_danych> )
```

Typem danych może być jeden z **typów skalarnych** języka SQL: `BFILE`, `BLOB`, `CLOB`, `DATE`, `NUMBER`, `ROWID`, `TIME`, `TIMESTAMP`, `TIME WITH TIME ZONE` lub `VARCHAR2`.

Procedura DEFINE_COLUMN_CHAR

Procedura `DEFINE_COLUMN_CHAR` obsługuje operacje DQL przeprowadzane wiersz po wierszu i definiuje kolumny używane w instrukcji `SELECT`. Metody tej trzeba użyć przed wywołaniem procedury `COLUMN_VALUE`. Procedura `DEFINE_COLUMN_CHAR` nie jest przeciążona i ma jedną sygnaturę.

Prototyp

```
define_column_char( numer_kursora    NUMBER
                  , pozycja          NUMBER
                  , wartość_zmiennej CHAR )
```

Procedura DEFINE_COLUMN_LONG

Procedura `DEFINE_COLUMN_LONG` obsługuje operacje DQL przeprowadzane wiersz po wierszu i definiuje kolumny używane w instrukcji `SELECT`. Metody tej trzeba użyć przed wywołaniem procedury `COLUMN_VALUE`. Procedura `DEFINE_COLUMN_LONG` nie jest przeciążona i ma jedną sygnaturę.

Prototyp

```
define_column_long( numer_kursora    NUMBER
                  , pozycja          NUMBER
                  , wartość_zmiennej LONG )
```

Procedura DEFINE_COLUMN_RAW

Procedura `DEFINE_COLUMN_RAW` obsługuje operacje DQL przeprowadzane wiersz po wierszu i definiuje kolumny używane w instrukcji `SELECT`. Metody tej trzeba użyć przed wywołaniem procedury `COLUMN_VALUE`. Procedura `DEFINE_COLUMN_RAW` nie jest przeciążona i ma jedną sygnaturę.

Prototyp

```
define_column_raw( numer_kursora    NUMBER
                 , pozycja          NUMBER
                 , wartość_zmiennej RAW )
```

Procedura DEFINE_COLUMN_ROWID

Procedura DEFINE_COLUMN_ROWID obsługuje operacje DQL przeprowadzane wiersz po wierszu i definiuje kolumny używane w instrukcji SELECT. Metody tej trzeba użyć przed wywołaniem procedury COLUMN_VALUE. Procedura DEFINE_COLUMN_ROWID nie jest przeciążona i ma jedną sygnaturę.

Prototyp

```
define_column_rowid( numer_kursora    NUMBER
                   , pozycja          NUMBER
                   , wartość_zmiennej ROWID )
```

Procedura DESCRIBE_COLUMNS

Procedura DESCRIBE_COLUMNS obsługuje operacje DQL i DML przeprowadzane masowo oraz wiersz po wierszu. Procedura ta wyświetla opis kolumn kursora otworzonego i przetwarzanego przez pakiet DBMS_SQL. W Oracle 10*g* obsługuje kolumny o nazwach krótszych niż 30 znaków. W Oracle 11*g* można używać nazw 32-znakowych. Nie jest przeciążona i ma tylko jedną sygnaturę.

Prototyp

```
describe_columns( numer_kursora     NUMBER
                , liczba_kolumn     NUMBER
                , kolekcja_rekordów DBMS_SQL.DESC_TAB )
```

Typ danych DBMS_SQL.DESC_TAB to tablica asocjacyjna wartości typu rekordowego DBMS_SQL.DESC_REC. Rekordowy typ DESC_REC przechowuje metadane z informacjami na temat wartości kolumn. Te dane są podzbiorem informacji dostępnych w widoku user_tables.

Procedura DESCRIBE_COLUMNS2

Procedura DESCRIBE_COLUMNS2 obsługuje operacje DQL i DML przeprowadzane masowo oraz wiersz po wierszu. Procedura ta wyświetla opis kolumn kursora otworzonego i przetwarzanego przez pakiet DBMS_SQL. Od wersji Oracle 10*g* obsługuje kolumny o nazwach do 32767 znaków. Nie jest przeciążona i ma tylko jedną sygnaturę.

Prototyp

```
describe_columns2( numer_kursora     NUMBER
                 , liczba_kolumn     NUMBER
                 , kolekcja_rekordów DBMS_SQL.DESC_TAB2 )
```

Typ danych DBMS_SQL.DESC_TAB2 to tablica asocjacyjna wartości typu rekordowego DBMS_
↪SQL.DESC_REC2. Rekordowy typ DESC_REC2 przechowuje te same metadane z informacjami

na temat wartości kolumn, co typ DESC_REC, ale umożliwia stosowanie dłuższych nazw kolumn. Te dane są podzbiorem informacji dostępnych w widoku user_tables.

Procedura DESCRIBE_COLUMNS3

Procedura DESCRIBE_COLUMNS3 obsługuje operacje DQL i DML przeprowadzane masowo oraz wiersz po wierszu. Procedura ta wyświetla opis kolumn kursora otworzonego i przetwarzanego przez pakiet DBMS_SQL. Od wersji Oracle 10*g* obsługuje kolumny o nazwach do 32767 znaków. Nie jest przeciążona i ma tylko jedną sygnaturę.

Prototyp

```
describe_columns3( numer_kursora     NUMBER
                 , liczba_kolumn     NUMBER
                 , kolekcja_rekordów DBMS_SQL.DESC_TAB3 )
```

Typ danych DBMS_SQL.DESC_TAB3 to tablica asocjacyjna wartości typu rekordowego DBMS_SQL.DESC_REC3. Rekordowy typ DESC_REC3 przechowuje te same metadane z informacjami na temat wartości kolumn, co typ DESC_REC2, a ponadto nazwę typu danych i jej długość. Te dane są szerszym podzbiorem informacji dostępnych w widoku user_tables.

Funkcja EXECUTE

Funkcja EXECUTE uruchamia instrukcje powiązane z otwartym kursorem pakietu DBMS_SQL. Zwraca liczbę wierszy zmodyfikowanych przez instrukcję DML. Przy uruchamianiu instrukcji DDL należy zignorować tę wartość, ponieważ **z technicznego punktu widzenia jest ona niezdefiniowana**. Funkcja ta nie jest przeciążona i ma jedną sygnaturę. Tryb działania parametru funkcji to IN.

Prototyp

```
execute( numer_kursora NUMBER ) RETURN NUMBER
```

Funkcja EXECUTE_AND_FETCH

Funkcja EXECUTE_AND_FETCH uruchamia instrukcję powiązaną z otwartym kursorem DBMS_SQL i pobiera z niego wiersze. Jej działanie przypomina uruchomienie funkcji EXECUTE i FETCH_ROWS jedna po drugiej. Zwraca liczbę wierszy zmodyfikowanych przez instrukcję DML. Przy uruchamianiu instrukcji DDL należy zignorować tę wartość, ponieważ **z technicznego punktu widzenia jest ona niezdefiniowana**.

Opcjonalny parametr powodujący dokładne pobieranie ma wartość domyślną false, co umożliwia pobranie więcej niż jednego wiersza. Po zmianie tej wartości domyślnej można pobrać tylko jeden wiersz. W wersji Oracle 7 i nowszych nie można używać opcji dokładnego pobierania przy stosowaniu kolumn typu LONG.

Funkcja ta nie jest przeciążona i ma jedną sygnaturę. Tryb działania parametru to IN.

Prototyp

```
execute_and_fetch( nazwa_kursora       NUMBER
                 , dokładne_pobieranie BOOLEAN DEFAULT FALSE ) RETURN NUMBER
```

Funkcja FETCH_ROWS

Funkcja `FETCH_ROWS` pobiera wiersz lub ich zbiór z określonego kursora. Można ją uruchamiać do momentu wczytania wszystkich wierszy. Funkcja `COLUMN_VALUE` zapisuje pobrany wiersz do zmiennej lokalnej, która może mieć typ **skalarny** lub oparty na **tabeli zagnieżdżonej**. Numer kursora jest przekazywany w trybie `IN`.

Funkcja `FETCH_ROWS` zwraca liczbę pobranych wierszy lub wartość –1, która oznacza, że program wczytał już wszystkie wiersze.

Prototyp

```
fetch_rows( numer_kursora NUMBER ) RETURN NUMBER
```

Funkcja IS_OPEN

Funkcja `IS_OPEN` sprawdza, czy kursor jest otwarty. Jeśli tak, zwraca `true`, a jeśli kursor jest zamknięty, zwraca `false`. Funkcja ta nie jest przeciążona i ma jedną sygnaturę. Tryb działania parametru to `IN`.

Prototyp

```
execute( numer_kursora NUMBER ) RETURN BOOLEAN
```

Funkcja LAST_ERROR_POSITION

Funkcja `LAST_ERROR_POSITION` zwraca pozycję błędu w tekście instrukcji SQL. Wartość ta jest wyrażona w bajtach. W odróżnieniu od innych mechanizmów, które rozpoczynają liczenie od 1, tu początek łańcucha znaków to pozycja 0. Funkcję tę trzeba wywołać po procedurze `PARSE`, ale przed wywołaniem funkcji wykonawczych.

Prototyp

```
last_error_position RETURN NUMBER
```

Funkcja LAST_ROW_COUNT

Funkcja `LAST_ROW_COUNT` zwraca sumę wierszy pobranych przez zapytanie. Przed jej wywołaniem należy uruchomić funkcję `EXECUTE_AND_FETCH` lub `FETCH_ROWS`. Jeśli programista wywoła najpierw funkcję `EXECUTE`, a następnie `LAST_ROW_COUNT`, otrzyma liczbę 0.

Prototyp

```
last_row_count RETURN NUMBER
```

Funkcja LAST_ROW_ID

Funkcja LAST_ROW_ID zwraca wartość typu ROWID ostatniego wiersza pobranego przez zapytanie. Przed wywołaniem tej funkcji trzeba uruchomić funkcję EXECUTE_AND_FETCH lub FETCH_ROWS.

Prototyp

```
last_row_id RETURN ROWID
```

Funkcja LAST_SQL_FUNCTION_CODE

Funkcja LAST_SQL_FUNCTION_CODE zwraca kod funkcji SQL powiązanej z daną instrukcją. Opis tych kodów zawiera podręcznik *Oracle Call Interface Programmer's Guide*. Funkcję tę trzeba wywołać bezpośrednio po instrukcji SQL, ponieważ w przeciwnym razie zwróci niezdefiniowaną wartość.

Prototyp

```
last_sql_function_code RETURN INTEGER
```

Funkcja OPEN_CURSOR

Funkcja OPEN_CURSOR otwiera kursor w bazie danych i zwraca jego numer. Aby zamknąć kursor i zwolnić zasoby, trzeba wywołać funkcję CLOSE_CURSOR.

Prototyp

```
open_cursor RETURN INTEGER
```

Procedura PARSE

Procedura PARSE parsuje podany łańcuch znaków z instrukcją. Parsowanie wszystkich instrukcji następuje natychmiast. Następnie instrukcje DML są umieszczane w kolejce wywołań funkcji EXECUTE lub EXECUTE_AND_FETCH, podczas gdy instrukcje DDL program przetwarza bezpośrednio po ich udanym parsowaniu. Jest to procedura przeciążona o pięciu rodzajach sygnatur.

Prototyp 1

```
parse( numer_kursora NUMBER
     , instrukcja   {CLOB | VARCHAR2}
     , opcja_języka  NUMBER )
```

Prototyp 2

```
parse( numer_kursora NUMBER
     , instrukcja   {CLOB | VARCHAR2}
     , opcja_języka  NUMBER
     , wersja        VARCHAR2 )
```

Prototyp 3

```
parse( numer_kursora      NUMBER
     , instrukcja        {VARCHAR2S | VARCHAR2A}
     , opcja_języka       NUMBER
```

```
      , dolne_ograniczenie NUMBER
      , górne_ograniczenie NUMBER
      , opcja_języka       NUMBER )
```

Prototyp 4

```
parse( numer_kursora      NUMBER
     , instrukcja        {VARCHAR2S | VARCHAR2A}
     , opcja_języka       NUMBER
     , dolne_ograniczenie NUMBER
     , górne_ograniczenie NUMBER
     , opcja_języka       NUMBER
     , wersja             VARCHAR2 )
```

Prototyp 5

```
parse( numer_kursora         NUMBER
     , instrukcja           {CLOB | VARCHAR2 | VARCHAR2S | VARCHAR2A}
     , opcja_języka          NUMBER
     , wersja                VARCHAR2
     , wyzwalacz_uniwersalny VARCHAR2
     , uruchom_wyzwalacz     BOOLEAN )
```

Typ danych `VARCHAR2S` to kolekcja typu tabeli zagnieżdżonej przechowująca 256-znakowe łańcuchy. Typ `VARCHAR2A` to kolekcja typu tabeli zagnieżdżonej przechowująca łańcuchy o 32767 znakach.

Funkcja TO_CURSOR_NUMBER

Funkcja `TO_CURSOR_NUMBER` przekształca kursor języka NDS na kursor pakietu `DBMS_SQL`. Może to być przydatne, jeśli programista otworzy kursor powiązany z nieznanymi kolumnami i chce przetwarzać go za pomocą pakietu `DBMS_SQL`. Funkcja ta przyjmuje kursor referencyjny w trybie `IN` i zwraca ogólny kursor referencyjny.

Prototyp

```
to_cursor_number( kursor_referencyjny REF CURSOR ) RETURNS NUMBER
```

Funkcja TO_REFCURSOR

Funkcja `TO_REFCURSOR` przekształca numer kursora pakietu `DBMS_SQL` na kursor referencyjny języka NDS. Jest to przydatne, jeśli programista otworzy kursor za pomocą pakietu `DBMS_SQL` i chce przetwarzać go przy użyciu języka NDS. Funkcja ta przyjmuje jeden numer kursora w trybie `IN` i zwraca kursor referencyjny.

Prototyp

```
to_refcursor( numer_kursora NUMBER) RETURN REF CURSOR
```

Procedura VARIABLE_VALUE

Procedura `VARIABLE_VALUE` obsługuje masowe i przeprowadzane wiersz po wierszu operacje DML. Służy do przekazywania wyników o różnych typach danych przy użyciu klauzuli `RETURNING`

INTO. Procedura ta wiąże zmienne różnych typów danych w instrukcjach języka SQL. Jest to procedura przeciążona o jednym rodzaju sygnatury. Kursor i nazwa kolumny są przekazywane przez wartość w trybie IN. Wartość zmiennej jest zwracana, ponieważ program przekazuje ją w trybie OUT.

Prototyp

```
variable_value( numer_kursora    NUMBER
              , nazwa_kolumny    VARCHAR2
              , wartość_zmiennej <typ_danych> )
```

Lista dostępnych typów danych obejmuje typy **skalarne** i **tablice asocjacyjne** zmiennych skalarnych. Można użyć dowolnego z następujących skalarnych typów danych: BFILE, BINARY_
↪DOUBLE, BINARY_FLOAT, BLOB, CLOB, DATE, INTERVAL YEAR TO MONTH, INTERVAL YEAR TO SECOND, NUMBER, REF OF STANDARD, ROWID, TIME, TIMESTAMP, TIMESTAMP WITH TIME ZONE, TIME WITH TIME ZONE i VARCHAR2. Funkcja ta obsługuje przy użyciu bibliotek OCI także tablice asocjacyjne (tabele języka PL/SQL), tablice VARRAY i typy zdefiniowane przez użytkownika.

Procedura VARIABLE_VALUE_CHAR

Procedura VARIABLE_VALUE_CHAR obsługuje operacje DML przeprowadzane wiersz po wierszu. Służy do zwracania wyników typu CHAR przy użyciu klauzuli RETURNING INTO. Nie jest przeciążona. Kursor i nazwa kolumny są przekazywane przez wartość w trybie IN. Procedura zwraca wartość zmiennej, która jest przekazywana w trybie OUT.

Prototyp

```
variable_value_char( numer_kursora    NUMBER
                   , nazwa_kolumny    VARCHAR2
                   , wartość_zmiennej CHAR )
```

Procedura VARIABLE_VALUE_RAW

Procedura VARIABLE_VALUE_RAW obsługuje operacje DML przeprowadzane wiersz po wierszu. Służy do zwracania wyników typu RAW przy użyciu klauzuli RETURNING INTO. Nie jest przeciążona. Kursor i nazwa kolumny są przekazywane przez wartość w trybie IN. Procedura zwraca wartość zmiennej, która jest przekazywana w trybie OUT.

Prototyp

```
variable_value_raw( numer_kursora    NUMBER
                  , nazwa_kolumny    VARCHAR2
                  , wartość_zmiennej RAW )
```

Procedura VARIABLE_VALUE_ROWID

Procedura VARIABLE_VALUE_ROWID obsługuje operacje DML przeprowadzane wiersz po wierszu. Służy do zwracania wyników typu ROWID przy użyciu klauzuli RETURNING INTO. Nie jest przeciążona. Kursor i nazwa kolumny są przekazywane przez wartość w trybie IN. Procedura zwraca wartość zmiennej, która jest przekazywana w trybie OUT.

Prototyp

```
variable_value_rowid( numer_kursora    NUMBER
                    , nazwa_kolumny    VARCHAR2
                    , wartość_zmiennej ROWID )
```

W tym podrozdziale opisano funkcje i procedury pakietu DBMS_SQL. Przykłady zastosowania większości z nich znajdują się w kodzie ilustrującym korzystanie z tego pakietu.

Podsumowanie

W tym rozdziale omówiono stosowanie języka NDS i pakietu DBMS_SQL do tworzenia oraz uruchamiania dynamicznych instrukcji języka SQL. Czytelnik powinien umieć zastosować takie instrukcje w aplikacjach w języku PL/SQL.