POLITECHNIKA WARSZAWSKA
Wydział Elektroniki i Technik Informacyjnych

PRACA DYPLOMOWA MAGISTERSKA

Marta Bańkowska
Nr albumu: 161484

Analiza porównawcza języków procedur składowanych PL/SQL, SQL PL, PL/pgSQL i T-SQL

Praca wykonana pod kierunkiem
doc. dra inż. Tomasza Traczyka

Wrzesień 2009

Streszczenie

W pracy porównano składnię języków procedur składowanych w czterech systemach zarządzania bazami danych: Oracle, IBM DB2, MS SQL Server i PostgreSQL. Analizą objęto podstawowe i zaawansowane właściwości języków. Wskazano podobieństwa i różnice oraz poparto je licznymi przykładami. Dokonano również oceny przydatności i wygody użycia poszczególnych języków.

Tytuł pracy w języku angielskim

Comparative analysis of stored procedure languages PL/SQL, SQL PL, PL/pgSQL and T-SQL

Abstract

The thesis contains comparison of stored procedure languages in four RDBMS'es (Relational Database Management Systems): Oracle, IBM DB2, MS SQL Server and PostgreSQL. The analysis includes basic and advanced characteristics of the languages. Similarities and differences have been indicated and supported by numerous examples. There has also been added an evaluation of usefulness and ease of use of respective languages.

Spis treści

Rozdział 1

Wprowadzenie

1.1 Cel pracy

Głównym celem pracy jest porównanie składni języków procedur składowanych w czterech systemach zarządzania bazami danych: Oracle, IBM DB2, MS SQL Server i PostgreSQL.

Dodatkowym zadaniem było uczestnictwo w projekcie *Pi of the Sky*. Dokładniej: wykonanie migracji części kodów procedur składowanych z systemu opartego na PostgreSQL do nowego, opartego na IBM DB2.

1.2 Proceduralny SQL – co to takiego?

Proceduralny SQL to język powstały przez rozszerzenie języka SQL o konstrukcje programistyczne takie jak deklaracje zmiennych czy instrukcje sterowania. Umożliwia zakodowanie reguł biznesowych po stronie serwera w postaci podprogramów, które można wielokrotnie wywoływać z poziomu różnych aplikacji. W przeciwieństwie do języka SQL pozwala zdefiniować nie tylko to, co ma być wykonane ale również sposób, w jaki ma być wykonane.

Język SQL (*Structured Query Language*) to język służący do komunikacji z bazą danych. Zawiera instrukcje do tworzenia i manipulacji obiektami bazy oraz instrukcje do umieszczania i manipulacji danymi w bazie. Jest językiem deklaratywnym – programista określa jakie warunki ma spełniać wynik, a nie w jaki sposób ten wynik osiągnąć.
Z punktu widzenia programisty jest jednak językiem niepełnym. Nie pozwala na tworzenie pętli, obsługę błędów oraz wszystko to, do czego przyzwyczajony jest użytkownik języków proceduralnych. Z tego powodu o SQL często mówi się, że jest raczej sposobem dostępu do bazy danych niż językiem programowania.

Konstrukcje proceduralne, istniejące w typowych językach programowania, często okazują się niezwykle przydatne (a czasem wręcz niezbędne)

podczas kodowania logiki biznesowej. Proceduralny SQL powstał właśnie jako połączenie możliwości oferowanych przez SQL z konstrukcjami proceduralnymi. Obok całego dobrodziejstwa SQL, proceduralny SQL oferuje możliwość tworzenia bardziej złożonego kodu oraz możliwość sterowania wykonaniem poszczególnych jego części. W praktyce przekłada się to (między innymi) na:

- korzystanie ze zmiennych, parametrów, tablic, kursorów, rekordów, wyrażeń warunkowych,
- stosowanie pętli i możliwość dostępu do kolejnych, pojedynczych rekordów, jeden po drugim,
- możliwość korzystania z mechanizmów obsługi błędów,
- możliwość przechowywania i wykonywania kodu proceduralnego po stronie serwera.

Ponadto proceduralny SQL „integruje" się z pewną grupą innych języków programowania. Owa integracja oznacza możliwość napisania funkcji w innym języku (np. C, C++, JAVA), a następnie powiązania jej z funkcją lub procedurą proceduralnego SQL bądź udostępnienia w taki sposób, jakby był to podprogram napisany w proceduralnym języku SQL. Następnie program taki wywołuje się tak jak podprogram napisany w proceduralnym SQL.

Wykorzystując możliwości proceduralnego języka SQL programiści mogą sporą część logiki biznesowej przenieść z aplikacji na stronę serwera. Procedury składowane na serwerze wykonywane są (dzięki mechanizmom optymalizacji i brakowi transmisji danych) szybciej, a kod aplikacji staje się bardziej przejrzysty i łatwiejszy do konserwacji. Działają również wbudowane w DBMS mechanizmy bezpieczeństwa.
Ponadto oprogramowanie serwera bazy danych ma tę zaletę, że w przypadku zmiany logiki biznesowej, zmiany w kodzie dokonywane są centralnie. Pomaga to zminimalizować liczbę modyfikacji wszystkich aplikacji działających w przedsiębiorstwie[1].

1.3 *Pi of the Sky* i jego miejsce w pracy

Pi of the Sky[2] to eksperyment astrofizyków zajmujących się poszukiwaniem błysków gamma. W ramach projektu przechowywane są ogromne ilości danych. Są to przede wszystkim informacje pochodzące z nocnych pomiarów. Zespół kamer obserwuje niebo i wykonuje zdjęcia. Zdjęcia trafiają do

[1] Nie każda logika daje się zaimplementować po stronie serwera, lecz warto oprogramowywać tę, którą się da.

[2] Oficjalna strona projektu: http://grb.fuw.edu.pl/

bazy danych, gdzie ich zawartość jest wstępnie interpretowana i przedstawiana w postaci liczb. Liczby są przechowywne w odpowiednich tabelach. W ten sposób powstają miliony surowych pomiarów, które podlegają dalszemu przetwarzaniu i analizie.

Obecnie na potrzeby eksperymentu *Pi of the Sky* wykorzystywany jest system zarządzania bazami danych PostgreSQL. Ma on jednak pewne ograniczenia, w tym również wydajnościowe, które stanowią barierę dla projektu. Z tego powodu podjęto decyzję o migracji systemu na bazę IBM DB2. Nowa platforma ma zapewnić dużo większą wydajność i skalowalność. Dotychczasowe zasoby zgromadzone w bazach PostgreSQL muszą zostać przetransportowane do nowej bazy.

Migracja z PostgreSQL na IBM DB2 obejmuje przeniesienie struktur danych a także logiki zaimplementowanej po stronie serwera, oraz migrację danych. Trudność tego procesu polega na tym, że obecnie istnieje wiele baz postgresowych. Mają one niejednolitą strukturę i są wykorzystywane wedle bieżących potrzeb. Migracja jest dobrą okazją do ujednolicenia struktur i stworzenia centralnej bazy danych. Szczególnie pierwsze z tych zagadnień ma niezwykłą wagę: dużo łatwiej jest bowiem analizować dane w pełni usystematyzowane i jednolite w ramach całego projektu.

Oto planowany przebieg migracji:

- instalacja i konfiguracja serwera IBM DB2,
- analiza i ujednolicenie dotychczasowych struktur danych,
- przygotowanie struktur danych, które będą implementowane pod DB2,
- przeniesienie warstwy oprogramowania serwera (funkcje, procedury, wyzwalacze),
- przeniesienie danych,
- testy, testy, testy...

Zadaniem autorki było ogólne uczestnictwo w całości procesu i szczególne uczestnictwo w przenoszeniu warstwy oprogramowania serwera. Wymagało to przepisania kodu zbioru podprogramów składowanych (procedur, funkcji, wyzwalaczy) oraz zoptymalizowania ich działania pod kątem serwera IBM DB2 V9.5.

Współpracownikiem autorki ze strony *Pi of the Sky* był pan Marcin Sokołowski. W dodatku A znajduje się lista procedur i funkcji, które zostały wytypowane przez niego do migracji. Wybór procedur był zdeterminowany stopniem ich ważności w projekcie. Załączony dokument ma charakter mało formalny, lecz zdecydowano się zamieścić go w tej formie, gdyż dobrze przedstawia to, co było do zrobienia oraz problemy jakie mogły wyniknąć w trakcie realizacji.

Zdaniem autorki, podczas migrowania procedur większym problemem mogło być zrozumienie logiki projektu (jest on już dość duży) niż samo przepisanie ich na język serwera DB2. Po wstępnej analizie, jako drugą trudność zauważono to, że wielokroć procedury są zagnieżdżone w sobie nawzajem, tworząc długie łańcuchy wywołań. Zatem przedstawiona w załączniku lista stała się niepełna, a liczba funkcji do przepisania mocno przybliżona. Niemniej również, po wstępnej analizie i zapoznaniu się z projektem, zadanie zapowiadało się bardzo ciekawie.

1.4 Przewodnik po następnych rozdziałach

Poniżej przedstawiono krótką charakterystykę treści zawartych w kolejnych rozdziałach pracy.

W rozdziale drugim, zatytułowanym *Proceduralny SQL – podstawy i nie tylko*, omówiono podstawowe właściwości języków PL/SQL, SQL PL, T-SQL i PL/pgSQL. Poruszono zagadnienia związane z typami danych, używaniem zmiennych i sterowaniem wykonaniem programu. Porównano sposoby pobierania i przetwarzania danych w poszczególnych językach. Ostatnie części rozdziału zastały poświęcone nieco bardziej zaawansowanym tematom – omówiono tu możliwość i sposoby wykorzystania tablic oraz typów tablicowych a także porównano sposoby przechwytywania i obsługiwania błędów.

Rozdział trzeci *Proceduralny SQL – zaawansowane możliwości* obejmuje zagadnienia związane z tworzeniem i uruchamianiem programów. Pokazano tu jakie rodzaje podprogramów można tworzyć w każdym z omawianych języków proceduralnych. Zaprezentowano metody tworzenia funkcji i procedur składowanych. Omówiono specjalny rodzaj procedur składowanych jakim są wyzwalcze. Pokazano również sposoby wywoływania i odbierania wartości z podprogramów. Na zakończenie rozdziału przedstawiono czym jest i jak stosować w programach dynamiczny język SQL.

Rozdział czwarty *Podsumowanie* w swej pierwszej części przedstawia wnioski płynące z omówionych w poprzednich rozdziałach zagadnień. Zestawiono tu najistotniejsze podobieństwa i różnice występujące pomiędzy językami PL/SQL, PL/pgSQL, SQL PL i T-SQL. Druga część rozdziału stanowi podsumowanie pracy wykonanej w ramach projektu *Pi of the Sky*.

Rozdział 2

Proceduralny SQL – podstawy i nie tylko

2.1 Serwery i dialekty

Na rynku obecnych jest wielu producentów, oferujących systemy zarządzania bazami danych. Wsród nich znajdują się: Oracle Corporation, International Business Machines Corporation (IBM), Microsoft Corporation oraz PostgreSQL Global Development Group. Pierwsi trzej proponują systemy komercyjne, ostatni system typu *open-source*. Odpowiednio w kolejności ich produkty to: Oracle Database, IBM DB2 Universal Database, Microsoft SQL Server i PostgreSQL. Na potrzeby niniejszego opracowania będą wykorzystywane następujące wersje wyżej wymienionych systemów: Oracle Database 11*g*, IBM DB2 V9.5, MS SQL Server 2005 oraz PostgreSQL w wersji 8.2.5 lub wyższej.

Każdy z czterech omawianych systemów posiada wbudowany język typu proceduralnego SQL. Niestety nie jest on ustandaryzowany i każdy z producentów implementuje go na swój sposób. Tak oto mamy do czynienia z kilkoma różnymi realizacjami proceduralnego SQL-a. Podstawą wszystkich jest standard SQL, dzięki czemu języki te mają ze sobą wiele wspólnego, niemniej jednak są też i poważne różnice.

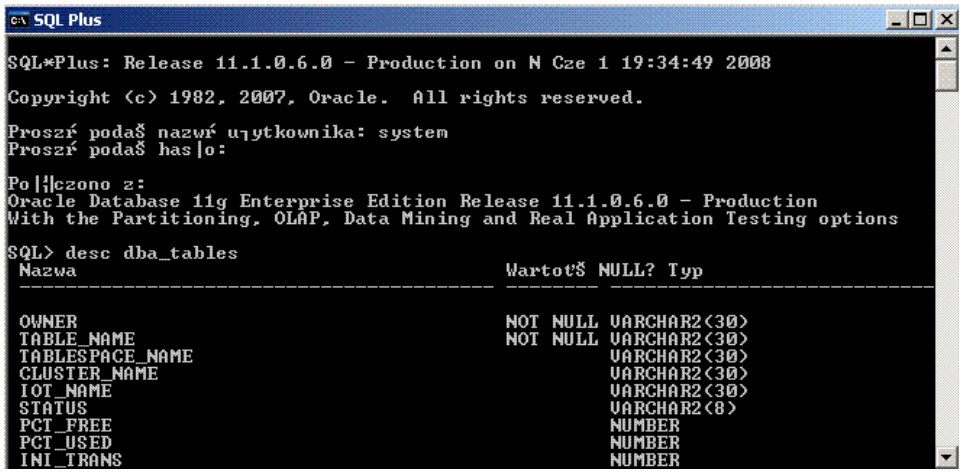
W kolejnych podrozdziałach zawarto porównanie czterech realizacji proceduralnego SQL:

– PL/SQL (Oracle),

– SQL PL (IBM),

– T-SQL (pełna nazwa: Transact-SQL, MS SQL Server),

– PL/PgSQL (PostgreSQL).



Rysunek 2.1: Konsola Oracle

2.2 W czym pisać, czyli narzędzia programistyczne

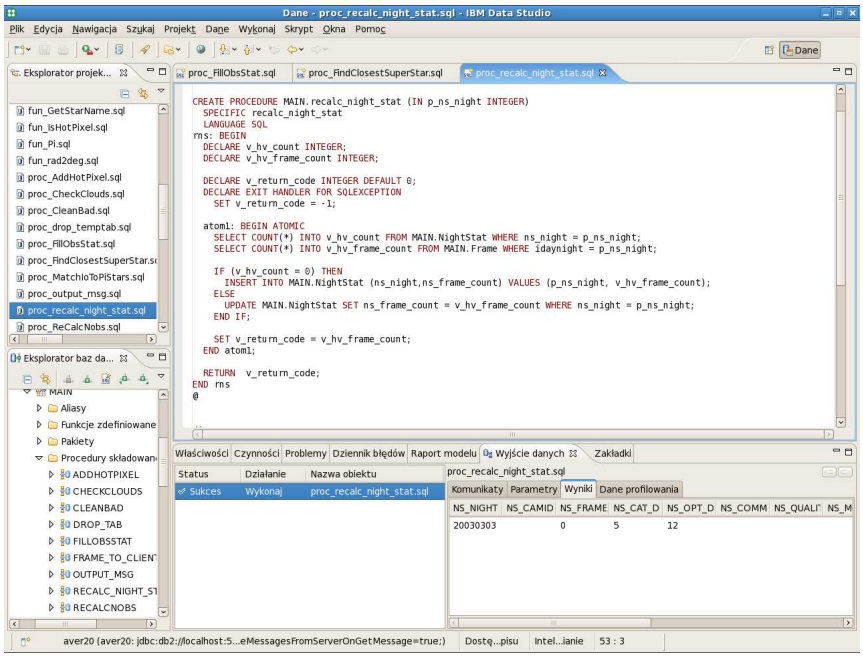
Zarówno Oracle, jak i DB2, MS SQL Server oraz PostgreSQL są wyposażone w rodzime narzędzia programistyczne. Nie oznacza to jednak, że użytkownik jest zobligowany do ich użytkowania. Kod procedur można tworzyć w dowolnym edytorze tekstowym. Na rynku jest dostępnych wiele edytorów, przystosowanych do pracy z różnymi systemami baz danych. Wybór narzędzia zależy głównie od upodobań programisty. Rodzime narzędzia omawianych tu czterech producentów to:

- Oracle: SQL*Plus i SQLDeveloper,
- IBM: Procesor CLP (DB2) i IBM Data Studio,
- MS SQL Server: SQLCMD (wcześniej OSQL) oraz SQL Server Management Studio (SSMS),
- PostgreSQL: psql i pgAdmin.

SQL*Plus, procesor wiersza komend CLP, SQLCMD i psql to narzędzia wiersza poleceń. Pozostałe to środowiska graficzne. Oto dwa przykładowe widoki: konsola (Oracle) na Rysunku 2.1 i edytor graficzny (IBM) na Rysunku 2.2.

2.3 Podstawowe zagadnienia proceduralnego SQL

Pracując z proceduralnym językiem SQL mamy do czynienia z typami danych, zmiennymi, blokami i instrukcjami sterującymi. Pojawiają się instruk-



Rysunek 2.2: IBM Data Studio

cje pobierania i przetwarzania danych, i związane z nimi transakcje. Posługujemy się rekordami i kursorami. Możemy również korzystać z udogodnień w postaci wbudowanych funkcji serwera. A oto jak te sprawy wyglądają w omawianych czterech językach.

2.3.1 Typy danych, zmienne, bloki i instrukcje sterujące

Bloki

Podstawową jednostką programu napisanego w proceduralnym języku SQL jest *blok*. Wewnątrz bloku zapisujemy zestaw instrukcji. Instrukcje mogą manipulować danymi, wyświetlać informacje na ekranie, wykonywać operacje na plikach czy wywoływać inne programy. Zazwyczaj blok obejmuje pewien logiczny fragment całego programu. Bloki mogą być anonimowe i nazwane. Mogą też być wzajemnie zagnieżdżane. Przykładem nazwanego bloku jest procedura. Składnia najprostszego bloku we wszystkich omawianych językach jest następująca:

```
BEGIN
   instrukcja[;]
   ...
END
```

Nawiasy kwadratowe oznaczają opcjonalność elementu. W języku T-SQL średnik kończący instrukcję można pominąć, w pozostałych jest wymagany i jego brak spowoduje błąd. W rzeczywistości jednak programy są bardziej skomplikowane i wówczas struktura bloku nieco różni się w każdym z języków. W Oracle'owym PL/SQL blok składa się z czterech części, z których dwie są opcjonalne:

```
[<<etykieta>>]
 DECLARE /opcjonalna/
   lista zmiennych i kursorów
 BEGIN
   przetwarzane instrukcje
 EXCEPTION /opcjonalna/
   obsługa wyjątków
 END [etykieta];
```

Sekcja BEGIN-END (podobnie jak w innych językach) musi zawierać przynajmniej jedną instrukcję. Podobnie wygląda struktura bloku w PL/pgSQL:

```
[<<etykieta>>]
 DECLARE /opcjonalna/
   sekcja deklaracji zmiennych
 BEGIN
   przetwarzane instrukcje
 END [etykieta]
```

Język PL/pgSQL również wyszczególnia część `EXCEPTION`. Definiuje ją jednak jako klauzulę bloku BEGIN-END. W praktyce, po uwzględnieniu klauzuli `EXCEPTION`, blok w języku PL/pgSQL przyjmuje taką postać jak przedstawiony wcześniej blok w języku PL/SQL.
Struktury bloków w SQL PL i T-SQL nie wyszczególniają osobno sekcji deklaracji zmiennych. Deklaracje zmiennych są tu zawarte wewnątrz bloków BEGIN-END.
SQL PL:

```
[etykieta:]
 BEGIN
    DECLARE

    inne instrukcje
 END [etykieta]
```

T-SQL:

```
[etykieta:]
 BEGIN
   instrukcje, w tym DECLARE
 END
```

Pisząc w SQL PL należy pamiętać o umieszczeniu wszystkich deklaracji na początku bloku BEGIN – END. Ponadto kolejność deklaracji zmiennych w SQL PL jest określona i zależna od typu deklarowanej zmiennej. Używając T-SQL mamy natomiast pełną dowolność. Zmienną możemy zadeklarować w każdym miejscu bloku. Ma to swoich zwolenników – nie trzeba się zastanawiać nad strukturą programu i można zadeklarować zmienną wówczas, gdy zaistnieje potrzeba. Zdaniem autorki jest to jednak zaleta pozorna: tak napisany kod jest nieczytelny i trudny w zrozumieniu oraz konserwacji. W T-SQL, jako jedynym, nie można powtórzyć nazwy etykiety na końcu etykietowanego bloku. Zapis `END etykieta` spowoduje błąd.

Podsumowując, Oracle zaproponował najbardziej usystematyzowaną ze struktur bloku programu. Choć na pierwszy rzut oka struktura ta może się wydawać nieco sztywna w stosunku do innych, to jednak okazuje się najwygodniejsza podczas pracy. Jej zalety stają się widoczne w momencie pracy z dużym, skomplikowanym programem. Stałe miejsce danego typu instrukcji w kodzie pozwala szybciej zrozumieć logikę zdefiniowaną w programie. Łatwiej jest odnaleźć poszczególne elementy oraz śledzić przebieg programu. Ma to duże znaczenie zarówno w pracy z własnym, jak i napisanym przez inną osobę, kodem.

Zmienne

Jedną z podstawowych instrukcji jest deklaracja zmiennej. Zmienna jest obszarem pamięci i służy do przechowywania danych wewnątrz bloków proceduralnego SQL. Poprzez zmienne przekazywane są informacje pomiędzy bazą danych a proceduralnym SQL.

W językach PL/SQL i PL/pgSQL składnia deklaracji zmiennej wygląda następująco:

```
DECLARE nazwa_zmiennej [ CONSTANT ] typ_danych
                  [ NOT NULL ] [ { DEFAULT | := } wartość ];
```

Nieco inaczej w SQL PL:

```
DECLARE nazwa_zmiennej typ_danych [ DEFAULT wartość ];
```

oraz w T-SQL:

```
DECLARE @nazwa_zmiennej [ AS ] typ_danych [ ; ]
```

Nazwy zmiennych są nadawane przez użytkownika i we wszystkich systemach muszą być zgodne z zasadami tworzenia identyfikatorów (ograniczenia co do długości i znaków specjalnych) w tych systemach. Cechą charakterystyczną Transact-SQL jest obecność znaku `@` na początku nazwy zmiennej. T-SQL jako jedyny nie pozwala na nadanie zmiennej wartości domyślnej. Ponadto, w PL/SQL i PL/pgSQL możemy zadeklarować stałą, używając słowa CONSTANT. Stałej takiej nie można później zmienić.

Typy danych

Deklarując zmienną musimy określić jakiego typu dane będzie przechowywała. Każdy z producentów implementuje własny system typów danych. W tym miejscu należy zaznaczyć, że zajmujemy się typami danych proceduralnego języka SQL, a nie typami z baz danych. Choć w większości przypadków typy z bazy danych i typy z proceduralnego SQL się pokrywają, to jednak zdarzają się odstępstwa.

W języku PL/SQL można wyodrębnić cztery kategorie typów predefiniowanych:

- typy skalarne,
- typy złożone,
- typy referencyjne
- oraz duże obiekty (LOB).

Typy skalarne przechowują pojedyncze wartości. Wyróżnia się kilka podkategorii typów skalarnych:

- liczby,
- znaki i łańcuchy znaków,
- wartości logiczne,
- datę, czas i przedziały czasowe.

Pierwsza z podkategorii, to typy przechowujące wartości liczbowe, przeznaczone do wykonywania obliczeń. Do kategorii tej należą cztery typy: `PLS_INTEGER` (lub `BINARY_INTEGER`), `BINARY_FLOAT`, `BINARY_DOUBLE` oraz `NUMBER`. Typy `PLS_INTEGER` (`BINARY_INTEGER`) i `NUMBER` mają ponadto swoje podtypy:

- Typ `PLS_INTEGER` (`BINARY_INTEGER`):
  - `NATURAL`,
  - `NATURALN`,
  - `POSITIVE`,
  - `POSITIVEN`,
  - `SIGNTYPE`
  - oraz `SIMPLE_INTEGER`;
- Typ `NUMBER`:

- `DEC, DECIMAL` lub `NUMERIC`,
- `DOUBLE PRECISION` lub `FLOAT`,
- `INT, INTEGER` lub `SMALLINT`
- oraz `REAL`.

Typy `BINARY_FLOAT` i `BINARY_DOUBLE` to typy pojedynczej i podwójnej precyzji. Obliczenia na tych typach nie wywołują błędów. Oracle udostępnia zestaw predefiniowanych stałych, których należy użyć (w przypadku tych typów) w celu sprawdzenia uzyskiwanych wartości (np.: `BINARY_FLOAT_NAN`, `BINARY_DOUBLE_INFINITY`).

Do kolejnej kategorii, kategorii typów znakowych, Oracle zalicza następujące typy:

- `CHAR`,
- `VARCHAR2`,
- `RAW`,
- `NCHAR`,
- `NVARCHAR2`,
- `LONG`,
- `LONG RAW`,
- `ROWID`
- i `UROWID`.

Typy `LONG, LONG RAW` i `ROWID` są przestarzałe i utrzymywane jedynie w celu wstecznej kompatybilności. Typy `LONG, LONG RAW` należy obecnie zastąpić typami odpowiednio `CLOB` lub `NCLOB` oraz `BLOB` lub `BFILE`. Typ `ROWID` zastępowany jest przez `UROWID`. Typ `RAW` przechowuje dane w postaci ciągów binarnych lub bajtów. Pseudokolumny `ROWID` i `UROWID` przechowują dane w postaci binarnej. Żeby pobrać dane z pseudokolumny `ROWID` do zmiennej typu znakowego (`VARCHAR2`, `CHAR`) należy zastosować funkcję `ROWIDTOCHAR`. Dla `UROWID` stosowanie specjalnej funkcji nie jest konieczne.

W kategorii wartości logicznych natomiast Oracle dysponuje tylko jednym typem. Jest to typ `BOOLEAN`. Wartości jakie można przypisać do zmiennej tego typu to: `TRUE, FALSE` oraz `NULL`.

Ostatnią predefiniowaną kategorią typów skalarnych są typy daty i czasu. Są to:

- `DATE`,
- `TIMESTAMP`,

- `TIMESTAMP WITH TIME ZONE`,
- `TIMESTAMP WITH LOCAL TIME ZONE`.

Oraz typy reprezentujące interwały czasowe:

- `INTERVAL YEAR TO MONTH`,
- `INTERVAL DAY TO SECOND`.

Do typów dużych obiektów (`LOB`) zaliczane są: `BFILE, BLOB, CLOB` oraz `NCLOB`. Oracle wyodrębnia jako osobną kategorię typy `XML`, do której zalicza się typ `XMLType` oraz rodzina typów `URI`.

Poza typami sklarnymi, można również posługiwać się typami złożonymi oraz referencjami. Typy złożone obejmują rekordy i kolekcje oraz typy obiektowe. Do typów referencyjnych zaliczają się natomiast dwa typy: `REF CURSOR` i `REF`.

Na podstawie typów predefiniowanych można tworzyć podtypy. Podtyp dziedziczy wszystkie właściwości z typu macierzystego. Różnicą jest ograniczony zbiór wartości, które może przyjmować, np.:

```
DECLARE
  SUBTYPE Liczby IS NUMBER(6,2);
```

Język PL/pgSQL wspiera wszystkie typy oferowane przez system zarządzania bazami danych PostgreSQL. Dokumentacja PostgreSQL wprowadza następującą klasyfikację typów:

- typy numeryczne,
- typ monetarny (`money`),
- typy znakowe,
- typ binarny (`bytea`),
- typy daty i czasu,
- typ logiczny (`boolean`),
- typy wyliczeniowe (`ENUM`),
- typy geometryczne,
- typy sieciowe,
- ciągi bitowe (`bit(n)` i `bit varying`),
- typy przeszukiwania pełnotekstowego,

- typ UUID (Universally Unique Identifier),
- typ XML,
- tablice,
- typy złożone,
- identyfikatory obiektów
- oraz pseudo-typy.

Do typów numerycznych zaliczają się:

- `smallint`,
- `integer`,
- `integer`,
- `decimal` lub `numeric`,
- `real`,
- `double precision`,
- `serial`
- oraz `bigserial`.

Typy znakowe to:

- `character` lub `char`,
- `character varying` lub `varchar`
- oraz `text`.

Natomiast do obsługi dat i czasu służą typy:

- `date`,
- `time [ without time zone ]`,
- `time with time zone`,
- `timestamp [ without time zone ]`
- oraz `timestamp with time zone`.

Podobnie jak w PL/SQL istnieje również typ `interval`. Pewną osobliwością są dostępne w PostgreSQL typy geometryczne (np.: `point`, `line` lub `path`), oraz typy adresów sieciowych (`cidr`, `inet` i `macaddr`). W kategorii pseudotypów mieszczą się natomiast między innymi `record` oraz `trigger`.

Jak widać, między typami oferowanymi przez Oracle i PostgreSQL można zauważyć wiele typów sobie odpowiadających, szczególnie w kategoriach typów znakowych, liczbowych oraz daty i czasu. PostgreSQL jako jedyny z omawianych tu systemów zarządzania bazami danych wprowadza taką różnorodność typów, jak specjalne typy do przechowywania adresów sieciowych czy typy geometryczne.

W tym momencie warto jeszcze wspomnieć kilka słów o dostępnym w omówionych powyżej językach PL/SQL i PL/pgSQL złożonym typie danych `RECORD`. Rekordy umożliwiają powiązanie kilku zmiennych w jeden byt. W PL/SQL jest to typ definiowany przez użytkownika. Deklaruje się go następująco:

```
DECLARE
  TYPE typ_rekordu IS RECORD (
    pole_1 typ_1 [NOT NULL] [:=wyrażenie_1],
    ...
    pole_n typ_n [NOT NULL] [:=wyrażenie_n]);
```

Oto przykład deklaracji rekordu t_Uczen i dwóch zmiennych rekordowych: z_Uczen1 i z_Uczen2 oraz przypisania wartości do pierwszej z nich:

```
DECLARE
  TYPE t_Uczen IS RECORD (
    imie VARCHAR(50),
    nazwisko VARCHAR(50),
    rok_ur NUMBER(4,0));
  z_Uczen1 t_uczen;
  z_Uczen2 t_uczen;
BEGIN
  z_Uczen1.imie := 'Magdalena';
  z_Uczen1.nazwisko := 'Chrząszcz';
  z_Uczen1.rok_ur := 1998;
END;
/
```

Rekord może być również przypisany instrukcją `SELECT`. Należy jedynie pamiętać, by pola w rekordzie odpowiadały kolumnom z instrukcji `SELECT`. Użycie omówionego w dalszej części pracy operatora `%ROWTYPE` jest niczym innym jak niejawną deklaracją rekordu.
W PL/pgSQL rekord nie jest prawdziwym typem danych, a zmienna zadeklarowana jako `RECORD` nie posiada struktury. Można do niej przypisać

wiersze z dowolnej tabeli. Podobnie jak dla PL/SQL, operator `%ROWTYPE` definiuje tutaj odmianę zmiennej rekordowej – zmienną o odpowiadającej danej tabeli strukturze. Oto przykład deklaracji zmiennej rekordowej bez struktury w PL/pgSQL:

```
pewna_zmienna RECORD;
```

Rekordy przydają się w pracy z kursorami[1].

Wracając do klasyfikacji typów, niezwykle prosto, w porównaniu z klasyfikacjami Oracle i PostgreSQL, wygląda klasyfikacja typów proponowana przez IBM. Główne kategorie to:

- typy numeryczne,
- typy znakowe,
- typy daty i czasu
- oraz typ referencyjny (`DATALINK`).

Do typów numerycznych zaliczają się:

- `SMALLINT`,
- `INTEGER`,
- `BIGINT`,
- `DECIMAL`,
- `REAL`
- i `DOUBLE`.

Typy ciągów znaków obejmują:

- `CHAR`,
- `VARCHAR`,
- `CLOB`,
- `GRAPHIC`,
- `VARGRAPHIC`,
- `DBCLOB`
- oraz `BLOB`;

[1] Kursory zostaną omówione w dalszej części pracy.

gdzie `BLOB`, `CLOB` i `DBCLOB` to typy reprezentujące duże obiekty. Można również do nich zaliczyć typ `XML`. Typ `BLOB` jest typem binarnym.
Ostatnią kategorią są typy obsługi daty i czasu. Zaliczamy tu:

- `DATE`,
- `TIME`
- oraz `TIMESTAMP`.

Typ `DATALINK` przechowuje zaś logiczną referencję do pliku zewnętrznego, zlokalizowanego poza bazą danych.

Ostatnią klasyfikacją jest zaszeregowanie typów według Microsoft. Tutaj podstawowe kategorie typów to:

- dokładne numeryczne,
- przybliżone numeryczne,
- daty i czasu,
- ciągi znakowe,
- ciagi znakowe Unicode,
- ciagi binarne
- oraz pozostałe typy.

Do typów dokładnych numerycznych zaliczają się następujące:

- `bigint`,
- `int`,
- `smallint`,
- `tinyint`,
- `decimal`,
- `numeric`,
- `money`,
- `smallmoney`,
- a także `bit`.

Przybliżone typy numeryczne to: `float` i `real`.
Kategorie typów znakówych i znakowych Unicode obejmują:

- `char` i `nchar`,
- `varchar` i `nvarchar`
- oraz `text` i `ntext`,

zaś typy binarne to:

- `binary`,
- `varbinary`
- i `image`.

Datę i czas obsługują dwa typy: `datetime` i `smalldatetime`. Typ `timestamp` wyróżniany jest w kategorii typów pozostałych, podobnie jak typy `cursor`, `sqlvariant`, `table`, `uniqueidentifier` oraz tym `xml`. Ponadto Microsoft wyodrębnia typy przechowujące duże wartości:

```
varchar(max), nvarchar(max) oraz varbinary(max)
```

oraz typy dużych obiektów:

```
text, ntext, image, varchar(max),
nvarchar(max), varbinary(max) oraz xml.
```

Typy `text`, `ntext` i `image` są obecnie przestarzałe. W ich miejsce należy stosować typy `varchar(max)`, `nvarchar(max)` oraz `varbinary(max)`.

Przyglądając się wszystkim przedstawionym tu klasyfikacjom typów, można wyciągnąć wniosek, że z tych czterech dialektów PL/SQL oferuje najbardziej usystematyzowny zestaw typów danych. Ma również najbogatszą propozycję typów liczbowych. Równocześnie jednak zawiera najwięcej odstępstw od standardu ANSI SQL. PL/pgSQL udostępnia natomiast sporo typów niestandardowych i niespotykanych w innych językach, jak rodzina typów geometrycznych czy rodzina typów sieciowych. Na koniec należy dodać, że poza typami wbudowanymi użytkownik ma również możliwość tworzenia własnych typów danych.

Zmienne, wartości i zasięg zmiennych

Przypisanie wartości do zadeklarowanej zmiennej odbywa się w ciele bloku. W tym celu należy użyć następującej instrukcji:

```
PL/SQL i PL/pgSQL:      zmienna := wartość;
SQL PL:             SET  zmienna  = wartość;
T-SQL :             SET @zmienna  = wartość;
```

Wartość do zmiennej można podstawić również instrukcją `SELECT INTO`, która zostanie omówiona później (T-SQL jej nie wspiera, pozostałe języki tak). W SQL PL możemy również wykorzystać składnię:

```
VALUES CURRENT DATE INTO zmienna;
```

do podstawienia wartości zwróconej przez funkcję lub zmiennej specjalnej (tutaj daty) do naszej zmiennej. PL/pgSQL pozwala natomiast na przemianowanie zmiennej poleceniem:

```
RENAME stara_nazwa_zmiennej TO nowa_nazwa_zmiennej;
```

Zmienna ma określony zasięg. Zadeklarowana w bloku ma zasięg tego bloku. Pisząc w PL/SQL i zagnieżdżając bloki możemy w bloku zagnieżdżonym zadeklarować zmienną o takiej samej nazwie co w bloku zewnętrznym. Przykładowo:

```
DECLARE
  zm_1 VARCHAR2(100);
  zm_2 VARCHAR2(100);
BEGIN
  zm_1 := 'zm_1 - blok zewnętrzny';
  zm_2 := 'zm_2 - blok zewnętrzny';
  DBMS_OUTPUT.PUT_LINE('zm_1: ' || zm_1);
  DBMS_OUTPUT.PUT_LINE('zm_2: ' || zm_2);

  DECLARE
    zm_1 CHAR(1);
    zm_2 CHAR(1);
  BEGIN
    zm_1 := 'A';
    zm_2 := 'B';
    DBMS_OUTPUT.PUT_LINE('zm_1: ' || zm_1);
    DBMS_OUTPUT.PUT_LINE('zm_2: ' || zm_2);
  END;

  DBMS_OUTPUT.PUT_LINE('zm_1: ' || zm_1);
  DBMS_OUTPUT.PUT_LINE('zm_2: ' || zm_2);
END;
```

Wówczas zmienna w bloku zagnieżdżonym przesłania zmienną z bloku zewnętrznego. Po wykonaniu powyższego kodu otrzymujemy:

```
zm_1: zm_1 - blok zewnętrzny
zm_2: zm_2 - blok zewnętrzny
zm_1: A
zm_2: B
zm_1: zm_1 - blok zewnętrzny
zm_2: zm_2 - blok zewnętrzny
```

Podobnie mamy w PL/pgSQL:

```
DECLARE
  zm_1 VARCHAR(100);
  zm_2 VARCHAR(100);
BEGIN
  zm_1 := ''zm_1 - blok zewnetrzny'';
  zm_2 := ''zm_2 - blok zewnetrzny'';

  RAISE NOTICE ''zm_1: %'', zm_1;
  RAISE NOTICE ''zm_2: %'', zm_2;

  DECLARE
    zm_1 CHAR(1);
    zm_2 CHAR(1);
  BEGIN
    zm_1 := ''A'';
    zm_2 := ''B'';
    RAISE NOTICE ''zm_1: %'', zm_1;
    RAISE NOTICE ''zm_2: %'', zm_2;
  END;

  RAISE NOTICE ''zm_1: %'', zm_1;
  RAISE NOTICE ''zm_2: %'', zm_2;
END;
```

oraz w SQL PL:

```
CREATE PROCEDURE przyklad_1 (OUT p_1 INT, p_2 VARCHAR(100))
  SPECIFIC przyklad_1
  LANGUAGE SQL
p1: BEGIN
  DECLARE zm_1 VARCHAR(100);
  DECLARE zm_2 VARCHAR(100);

  BEGIN
    SET zm_1 = 'zm_1 - blok zewnetrzny';
    SET zm_2 = 'zm_2 - blok zewnetrzny';

    BEGIN
      DECLARE zm_1 INT;
      DECLARE zm_2 INT;
    SET zm_1 = 3;
    SET zm_2 = 4;
    SET p_1 = zm_1 * zm_2;
    END;
```

```
      SET p_2 = zm_1 || ' ' ||zm_2;
    END;
  END p1
```

Z tej właściowości nie skorzystamy natomiast w T-SQL. Tu wykonanie bloku:

```
  BEGIN
    DECLARE @zm_1 INT;
    DECLARE @zm_2 VARCHAR(10);
    SET @zm_1 = 2;
    SELECT @zm_1;

    BEGIN
      DECLARE @zm_1 VARCHAR(10);
      SET @zm_1 = 'ala ma kota';
      SET @zm_2 = @zm_1;
    END
END
```

zakończy się błędem o zduplikowanej deklaracji zmiennej: „Msg 134, Level 15, State 1, Line 8 The variable name '@zm_1' has already been declared. Variable names must be unique within a query batch or stored procedure."

PL/SQL i PL/pgSQL udostępniają możliwość zadeklarowania zmiennej zgodnej z typem jednej konkretnej kolumny lub całym wierszem tabeli z bazy danych. Służą do tego operatory %TYPE i %ROWTYPE. Zakładając, że w bazie danych istnieje tabela UCZESTNICY, o czterech kolumnach: ID, IMIE, NAZWISKO i ROK_UR, możemy powołać do życia zmienne w następujący sposób:

```
DECLARE v_nazwisko UCZESTNICY.NAZWISKO%TYPE;
DECLARE v_uczestnik UCZESTNICY%ROWTYPE;
```

W przypadku bazy Oracle, jeżeli założymy, że typy danych przechowywanych w kolumnach tabeli to odpowiednio: NUMBER(3,0), VARCHAR2(50), VARCHAR2(50) i NUMBER(4,0), zmienna v_nazwisko będzie typu VARCHAR2(50), a zmienna v_uczestnik będzie miała postać rekordu o budowie:

```
ID          NUMBER(3,0)
IMIE        VARCHAR2(50)
NAZWISKO    VARCHAR2(50)
ROK_UR      NUMBER(4,0)
```

Taki sposób deklaracji zmiennej jest bardzo przydatny w wypadku zmiany typów danych kolum tabeli bazowej. Jeśli typ kolumny się zmieni, to zmienne zadeklarowane przy pomocy operatorów %TYPE i %ROWTYPE

automatycznie ulegną zmianie przy kolejnej kompilacji kodu. Programista nie musi ręcznie wprowadzać zmian w kodzie.
Niestety, zarówno w SQL PL jak i T-SQL tego udogodnienia nie znajdziemy.

Sterowanie wykonaniem programu

Kolejną ważną rzeczą jest możliwość sterowania wykonaniem programu. Programy napisane w proceduralnym języku SQL są wykonywane sekwencyjnie, od góry bloku kodu do jego dołu, o ile nie zostały zastosowane struktury sterujące. Struktury sterujące umożliwiają sprawdzanie warunków, wielokrotne wykonywanie kodu i przeskakiwanie do innych części bloków. Zgodnie z tym wyróżniamy trzy kategorie struktur sterujących:

- warunki – sterowanie wykonaniem kodu w zależności od spełnienia określonych warunków;
- pętle – wykonywanie kodu w pętli do momentu spełnienia danego warunku;
- nawigacja sekwencyjna – przenoszenie się do innych sekcji bloku.

Do pierwszej grupy struktur sterujących zaliczamy instrukcje `IF-THEN` oraz `CASE`.

Instrukcja `IF-THEN` działa następująco: `IF` warunek jest spełniony, `THEN` należy wykonać pewne operacje. Jeżeli warunek nie jest spełniony, operacje te należy pominąć i przejść do wykonywania pozostałej części programu. W PL/SQL mamy do dyspozycji trzy odmiany struktury `IF-THEN`. Jej najprostsza forma to:

```
IF warunek THEN
  operacje;
END IF;
```

gdzie *warunek* to porównanie dowolnego wyrażenia, zmiennej, stałej lub identyfikatora z innym wyrażeniem, zmienną, stałą lub identyfikatorem. Warunek musi przyjmować wartość `TRUE`, `FALSE` lub `NULL`. Jeżeli warunek zostanie spełniony (`TRUE`), *operacje* zostaną wykonane, jeżeli zaś nie zostanie spełniony (`FALSE` lub `NULL`), wówczas *operacje* nie zostaną wykonane. Drugi wariant struktury rozszerza jej możliwości o wykonanie operacji w przypadku, gdy zadany warunek nie jest spełniony:

```
IF warunek THEN
  operacje;
ELSE
  operacje;
END IF;
```

Trzecia odmiana tej struktury pozwala na połączenie wielu warunków `IF` w łańcuch:

```
IF warunek THEN
  operacje;
ELSIF warunek THEN
  operacje;
ELSIF warunek THEN
  operacje
[ELSE
  operacje]
END IF;
```

Oto przykład użycia struktury `IF-THEN` w języku PL/SQL:

```
CREATE OR REPLACE PROCEDURE przyklad_2
(p_miasto klienci.miasto%TYPE) AS
  v_liczba_klientow PLS_INTEGER;
BEGIN
  SELECT count(*) INTO v_liczba_klientow FROM klienci
    WHERE UPPER(miasto) LIKE UPPER(p_miasto);
  IF (v_liczba_klientow > 4) THEN
    DBMS_OUTPUT.PUT_LINE ('W mieście ' || p_miasto ||
      ' mamy wielu klientów, ' ||
     'a dokładnie jest ich ' ||
     TO_CHAR(v_liczba_klientow) ||'.');
  ELSIF (v_liczba_klientow BETWEEN 2 AND 4) THEN
    DBMS_OUTPUT.PUT_LINE ('W mieście ' || p_miasto ||
      ' mamy kilku klientów.');
  ELSIF (v_liczba_klientow = 1) THEN
    DBMS_OUTPUT.PUT_LINE ('W mieście ' || p_miasto ||
      ' mamy bardzo mało klientów.');
  ELSE
    DBMS_OUTPUT.PUT_LINE ('W mieście ' || p_miasto ||
      ' nie mamy klientów.');
  END IF;
END;
/
```

Język PL/pgSQL również oferuje te trzy warianty konstrukcji `IF-THEN`. Mają one taką samą składnię jak w PL/SQL. Jedyna różnica polega na tym, że w PL/pgSQL dostępne są dwie formy zapisu: `ELSIF` oraz `ELSEIF`, podczas gdy w PL/SQL istnieje tylko pierwsza z nich. Oto ten sam przykład do uruchomienia pod PostgreSQL:

```
CREATE FUNCTION przyklad_2
```

```
(p_miasto klienci.miasto%TYPE) RETURNS int AS'
DECLARE
  v_liczba_klientow INTEGER;
BEGIN
  SELECT count(*) INTO v_liczba_klientow FROM klienci
    WHERE UPPER(miasto) LIKE UPPER(p_miasto);
  IF (v_liczba_klientow > 4) THEN
    RAISE NOTICE ''W miescie % mamy wielu klientow,
      a dokladnie %.'', p_miasto, v_liczba_klientow;
  ELSEIF (v_liczba_klientow >= 2 AND
    v_liczba_klientow <= 4) THEN
    RAISE NOTICE ''W miescie % mamy kilku klientow.'',
      p_miasto;
  ELSIF (v_liczba_klientow = 1) THEN
    RAISE NOTICE ''W miescie % mamy bardzo mało
      klientów.'', p_miasto;
  ELSE
    RAISE NOTICE ''W miescie % nie mamy klientów.'',
      p_miasto;
  END IF;
  RETURN v_liczba_klientow;
END;
'LANGUAGE 'plpgsql';
```

Choć nie jest to eleganckie rozwiązanie, to dla przykładu użyto tu raz `ELSIF` a raz `ELSEIF`.
Również w SQL PL można skorzystać z wymienionych wcześniej trzech form `IF-THEN`. W wypadku wielu warunków stosuje się tu zapis `ELSEIF`. Przedstawia to poniższy przykład:

```
IF (p_1 = 1)  THEN
  SET p_2 = 'jeden';
ELSEIF (p_1 = 2) THEN
  SET p_2 = 'dwa';
ELSE
  SET p_2 = 'inne';
END IF;
```

Jak widać, składnia tej instrukcji w PL/SQL, PL/pgSQL i SQL PL jest niemalże identyczna. Tym bardziej staje się widoczna różnica w składni używanej w T-SQL. Tutaj należy skorzystać z następujących trzech sposobów zapisu:

```
IF warunek
  operacje
```

```
IF warunek
  operacje
ELSE
  operacje
```

```
IF warunek
  operacje
ELSE IF warunek
  operacje
ELSE
 operacje
```

Nie występuje tu słowo kluczowe `THEN` oraz kończący strukturę `END IF`. Ponadto należy pamiętać o zastosowaniu bloku `BEGIN-END` do wykonania więcej niż jednej operacji:

```
BEGIN
  DECLARE @v_zm INT;
  SET @v_zm = 7;

  IF (@v_zm = 2)
    BEGIN
      SELECT 'dwa';
      SELECT 'Tu nie ma END IF, za to
           w IF jest blok BEGIN-END';
    END
  ELSE IF (@v_zm = 3)
    SELECT 'trzy'
  ELSE
   SELECT 'inne'
END
```

Zdaniem autorki to najmniej „przyjazna" forma zapisu, o czym można się przekonać podczas próby napisania złożonego warunku. Oto jak nieczytelny zapis można uzyskać już w niewielkim programie:

```
BEGIN
  DECLARE @v_zm INT;
  SET @v_zm = 4;
  IF (@v_zm = 2)
    SELECT 'dwa';
  ELSE IF (@v_zm > 3)
```

```
    IF @v_zm < 6
      IF @v_zm = 4
        BEGIN
          SELECT 'równe'
          SELECT 'cztery';
        END
      ELSE
        SELECT 'nie równe cztery'
    ELSE
      SELECT 'większe lub równe 6';
  ELSE
   SELECT 'inne'
END
```

Jeżeli programista T-SQL nie zadba o właściwą estetykę kodu może po pewnym czasie spędzić sporo czasu nad rozszyfrowaniem własnych intencji. Wystarczy kilka zagnieżdżonych warunków, by poszukiwanie popełnionego błędu trwało dłużej niż napisanie kodu od początku. W przykładach pokazano również opcjonalność użycia kończącego instrukcję znaku „;".
Wszystkie języki pozwalają natomiast na zagnieżdżanie w strukturze warunkowej kolejnej takiej struktury.

Drugim sposobem na warunkowe wykonanie kodu jest instrukcja `CASE`. Znajdziemy ją w PL/SQL i SQL PL. PL/pgSQL do wersji 8.3 (czyli do chwili powstawania niniejszej pracy) oraz T-SQL jej nie implementują. Składnia instrukcji `CASE` w PL/SQL i SQL PL jest taka sama i dostępna w dwóch wariantach:

```
CASE warunek
  WHEN test_1 THEN operacje;
  ...
  WHEN test_n THEN operacje;
  [ELSE operacje;]
END CASE;


CASE
  WHEN warunek THEN operacje;
  ...
  WHEN warunek THEN operacje;
  [ELSE operacje;]
END CASE;
```

Instrukcje `IF-THEN` i `CASE` działają w ten sam sposób, różnią sie sposobem zapisu. Poniżej zamieszczono przykłady dla obu wariantów `CASE`. Pierwszy z nich pokazuje tę instrukcję w języku PL/SQL a drugi w SQL PL:

```
CASE v_kategoria
  WHEN 'Sensacja' THEN
    v_cena_z_rabatem := v_cena * 0.8;
  WHEN 'Fantastyka' THEN
    v_cena_z_rabatem := v_cena * 0.9;
  ELSE
    v_cena_z_rabatem := v_cena;
END CASE;

CASE
  WHEN (v_kwota > 1000) THEN
    SET v_rabat = 30;
  WHEN (v_kwota BETWEEN 500 AND 1000) THEN
    SET v_rabat = 20;
  WHEN ( v_kwota BETWEEN 200 AND 499.99) THEN
    SET v_rabat = 10;
  WHEN (v_miasto LIKE 'Warszawa') THEN
    SET v_rabat = 3;
  ELSE
    SET v_rabat = 2;
END CASE;
```

W drugim przykładzie zaprezentowano możliwość sprawdzania różnych warunków w obrębie jednej instrukcji `CASE`.
W językach PL/pgSQL i T-SQL, które nie implementują instrukcji `CASE`, w jej miejsce można zastosować wyrażenie `CASE`.

Wyrażenie `CASE` jest częścią języka SQL. Jest dostępne we wszystkich omawianych tu językach proceduralnych oraz w niektórych dialektach SQL. W języku PL/SQL stosuje się je następująco:

```
DECLARE
  znak CHAR(1) := 'A';
  napis VARCHAR2(10);
BEGIN
  napis :=
    CASE znak
      WHEN 'A' THEN 'Literka A'
    WHEN 'B' THEN 'Literka B'
      WHEN 'C' THEN 'Literka C'
      ELSE 'Inna literka'
  END;
  DBMS_OUTPUT.PUT_LINE(napis);
END;
/
```

Podobnie jak w instrukcji sterującej `CASE`, również w wyrażeniu `CASE` można zamiast porównania jednego warunku porównywać wiele warunków, przenosząc porównanie do klauzuli `WHEN`, czyli:

```
CASE
  WHEN znak = 'A' THEN ...
```

Składnia samego `CASE` jest taka sama we wszystkich czterech językach. Należy jednak pamiętać o użyciu odpowiedniej składni podczas deklarowania zmiennych i podstawiania do nich wartości. Na przykład w T-SQL o dodaniu „`@`" do nazwy zmiennej oraz użyciu „`SET @zmienna =` " zamiast „`:=`".

Drugą grupę struktur sterujących stanowią pętle. PL/SQL i PL/pgSQL oferują trzy konstrukcje pętli:

- podstawowową pętlę LOOP,
- pętlę WHILE
- oraz pętlę FOR.

W celu wyjścia z pętli można użyć instrukcji:

- EXIT
- lub EXIT WHEN,

zaś w celu wyskoczenia z bieżącej i przejścia do kolejnej iteracji:

- CONTINUE
- lub CONTINUE WHEN.

Wewnątrz pętli można również zastosować instrukcję `RETURN`, żeby zakończyć wykonywanie całego bloku kodu (nie tylko samej pętli). Instrukcja `RETURN` zostanie szerzej omówiona w podrozdziale omawiającym funkcje. Ponadto pętli można nadać etykietę, co pozwala odwoływać się do pętli tak, jakby miała nazwę. Pętli nie można nadać nazwy bezpośrednio. Etykietowanie pętli znajduje swoje zastowowanie w przypadku pętli zagnieżdżonych. Instrukcja `EXIT etykieta` pozwala wówczas opuścić pętlę wskazaną przez etykietę. Bez etykiety możliwe byłoby wyjście jedynie z pętli bieżącej.
Składnia pętli jest w obu językach (PL/SQL i PL/pgSQL) jednakowa. Najprostszą pętlę LOOP buduje się w następujący sposób:

```
[<<etykieta>>]
LOOP
  instrukcje;
END LOOP [etykieta];
```

Wewnątrz pętli trzeba umieścić instrukcję określającą warunek zakończenia wykonywania pętli. Można to zrobić za pomocą instrukcji `EXIT` lub `EXIT WHEN`. Brak takiego warunku spowoduje wykonywanie pętli w nieskończoność. Poniżej znajduje się przykład pętli LOOP z użyciem `EXIT` (napisany w języku PL/pgSQL) i z użyciem `EXIT WHEN` (napisany w PL/SQL).

```
DECLARE
  v_zm INTEGER := 1;
BEGIN
  LOOP
    RAISE NOTICE ''Przebieg: % '', v_zm;
    v_zm := v_zm + 1;
    IF v_zm >= 7 THEN
      EXIT;
    END IF;
  END LOOP;
  (...)
END;
```

```
DECLARE
  v_zm PLS_INTEGER := 1;
BEGIN
  LOOP
    DBMS_OUTPUT.PUT_LINE('Przebieg: ' || v_zm);
    v_zm := v_zm + 1;
    EXIT WHEN v_zm >= 7;
  END LOOP;
END;
```

W bloku napisanym w PL/pgSQL wystarczy zmienić nazwę użytego dla zmiennej typu danych i instrukcję wyprowadzającą dane na ekran, by móc uruchomić go pod Oracle i na odwrót. Składnia pętli jest identyczna. Za pomocą `CONTINUE WHEN` i `CONTINUE` można natomiast sterować wykonywaniem kolejnych iteracji pętli.
PL/SQL i `CONTINUE WHEN`:

```
DECLARE
  v_zm PLS_INTEGER := 1;
BEGIN
  LOOP
    DBMS_OUTPUT.PUT_LINE
      ('Zmienna v_zm = : ' || TO_CHAR(v_zm));
    v_zm := v_zm + 1;
    CONTINUE WHEN v_zm < 5;
```

```
    DBMS_OUTPUT.PUT_LINE
      ('Po CONTINUE: Zmienna v_zm = : ' || TO_CHAR(v_zm));
    EXIT WHEN v_zm = 7;
  END LOOP;

  DBMS_OUTPUT.PUT_LINE
    ('Po pętli: Zmienna v_zm = : ' || TO_CHAR(v_zm));
END;
```

PL/pgSQL i CONTINUE:

```
DECLARE
  v_zm INTEGER := 1;
BEGIN
  LOOP
    RAISE NOTICE ''Zmienna v_zm = %'', v_zm;
    v_zm := v_zm + 1;
    IF v_zm < 5 THEN
      CONTINUE;
    END IF;
    RAISE NOTICE ''Po CONTINUE: Zmienna v_zm = % '', v_zm;
    EXIT WHEN v_zm = 7;
  END LOOP;

  RAISE NOTICE ''Po petli: Zmienna v_zm = % '', v_zm;
  (...)
END;
```

W obu językach możliwe jest również użycie wspomnianej wcześniej instrukcji RETURN. Przykładowo, w PL/pgSQL może to wyglądać tak:

```
CREATE FUNCTION petla_return_1 () RETURNS int AS'
DECLARE
  v_zm INTEGER := 1;
BEGIN
  LOOP
    RAISE NOTICE ''To sie wyswietli.'';
    v_zm := v_zm + 1;
    RETURN 3;
    RAISE NOTICE ''A to juz nie.'';
    EXIT WHEN v_zm >= 7;
  END LOOP;
RETURN 1;
END;
'LANGUAGE 'plpgsql';
```

Funkcja zwróci wartość 3 i wypisze na ekranie komunikat "To sie wyswietli.". Wszelkie operacje znajdujące się po instrukcji `RETURN` zostaną pominięte.

Jak wspomniano wcześniej, w przypadku pętli zagnieżdżonych użyteczne stają się etykiety. Etykieta nadana pętli pozwala wskazać, z której pętli należy wyskoczyć. Składnia etykiet jest również w obu językach taka sama. Sposób wykorzystywania etykiet pokazano na przykładzie języka PL/SQL:

```
DECLARE
  v_zm_1 PLS_INTEGER := 0;
  v_zm_2 PLS_INTEGER;
  v_zm_3 PLS_INTEGER := 0;
BEGIN
  <<petla_zew>>
  LOOP
    DBMS_OUTPUT.PUT_LINE('Petla_zew: '||TO_CHAR(v_zm_1));
    v_zm_1 := v_zm_1 + 1;
    v_zm_2 := 0;
    <<petla_wew>>
    LOOP
     v_zm_2 := v_zm_2 + 1;
     v_zm_3 := v_zm_3 + 2;
     DBMS_OUTPUT.PUT_LINE('Petla_wew: '||TO_CHAR(v_zm_2));
     DBMS_OUTPUT.PUT_LINE('Petla_wew: '||TO_CHAR(v_zm_3));
     EXIT petla_wew WHEN v_zm_2 > 2;
     EXIT petla_zew WHEN v_zm_3 > 5;
    END LOOP petla_wew;
  END LOOP petla_zew;
  DBMS_OUTPUT.PUT_LINE('.....................');
  DBMS_OUTPUT.PUT_LINE('Po petla_zew: '||TO_CHAR(v_zm_1));
  DBMS_OUTPUT.PUT_LINE('Po petla_wew: '||TO_CHAR(v_zm_2));
  DBMS_OUTPUT.PUT_LINE('Po petla_wew: '||TO_CHAR(v_zm_3));
END;
```

Po wykonaniu powyższego bloku otrzymujemy:

```
Petla_zew: 0
Petla_wew: 1
Petla_wew: 2
Petla_wew: 2
Petla_wew: 4
Petla_wew: 3
Petla_wew: 6
Petla_zew: 1
Petla_wew: 1
Petla_wew: 8
```

```
.....................
Po petla_zew: 2
Po petla_wew: 1
Po petla_wew: 8
```

Kolejną pętlą jest pętla `WHILE`. Zawarte w ciele tej pętli operacje są wykonywane dopóty, dopóki określony na początku pętli warunek jest prawdziwy. Warunek jest sprawdzany przed wykonaniem każdej iteracji pętli, co oznacza, że pętla może nie wykonać się ani razu. Liczba iteracji jest nieznana, a przerwanie wykonywania następuje w momencie, gdy warunek osiągnie wartość `FALSE` lub `NULL`. Składnia `WHILE LOOP` dla PL/SQL i PL/pgSQL wygląda następująco:

```
[<<etykieta>>]
WHILE warunek LOOP
   operacje;
END LOOP [etykieta];
```

Przykładowo, w języku PL/SQL:

```
WHILE (v_zm < 3) LOOP
  DBMS_OUTPUT.PUT_LINE ('Iteracja: ' || TO_CHAR(v_zm));
  v_zm := v_zm + 1;
END LOOP;
```

Ostatnią petlą jest pętla `FOR`. Pętla ta ma dwa rodzaje: może być iterowana po zbiorze liczb całkowitych lub po zbiorze wyników zapytania. Drugi typ zostanie omówiony później, wraz z kursorami. Teraz zostanie zaprezentowany wariant liczbowy. Pętla `FOR` wygląda w tym przypadku następująco:

```
[<<etykieta>>]
FOR licznik IN [REVERSE] zakres_dolny .. zakres_gorny LOOP
   operacje;
END LOOP [etykieta];
```

Licznik pętli nie musi być wcześniej zadeklarowany. Wewnątrz pętli można go odczytać, ale nie można zmodyfikować. Pierwsza iteracja pętli zaczyna się od licznika równego zakresowi dolnemu, a następnie w każdej iteracji licznik jest powiększany o 1, aż osiągnie zakres górny. Opcjonalne słowo `REVERSE` służy do odwrócenia kolejności odliczania. Pętla bez użycia słowa `REVERSE` wygląda w PL/SQL i PL/pgSQL identycznie[2]. Przykład w języku PL/pgSQL:

```
FOR i IN 1..4 LOOP
  RAISE NOTICE ''Iteracja: %'', i;
END LOOP;
```

[2]Identyczność dotyczy konstrukcji pętli, a nie instrukcji użytych wewnątrz niej, tu: RAISE NOTICE, którą w PL/SQL należy zastąpić instrukcją DBMS_OUTPUT.PUT_LINE.

W przypadku wykorzystania słowa `REVERSE` pojawia się natomiast różnica. Oto kod w języku PL/SQL:

```
BEGIN
  FOR i IN REVERSE 1..4 LOOP
   DBMS_OUTPUT.PUT_LINE ('Iteracja: ' || TO_CHAR(i));
  END LOOP;
END;
```

Rezultatem wykonania jest:

```
Iteracja: 4
Iteracja: 3
Iteracja: 2
Iteracja: 1
```

Żeby osiągnąć taki wynik w PL/pgSQL należy pętlę zapisać następująco:

```
FOR i IN REVERSE 4..1 LOOP
  RAISE NOTICE ''Iteracja: % '', i;
END LOOP;
```

Jak widać, w PL/pgSQL po słowie `REVERSE` należy podać zakres w odwróconej kolejności.
Wewnątrz pętli `WHILE` i `FOR` możemy równie posłużyć się intrukcjami `EXIT` i `EXIT WHEN`, `CONTINUE` i `CONTINUE WHEN` oraz `RETURN`.

Język SQL PL oferuje cztery rodzaje pętli. Podobnie jak PL/SQL oraz PL/pgSQL udostępnia pętle `LOOP`, `WHILE` i `FOR`. Czwarta konstrukcja to pętla `REPEAT`. W odróżnieniu od PL/SQL i PL/pgSQL, w SQL PL nie istnieje iterujący po liczbach całkowitych wariant pętli `FOR`. Tutaj `FOR LOOP` służy wyłącznie do operowania na zbiorze wyników zapytania, dlatego też pętla ta zostanie omówiona w kolejnym rozdziale wraz z przetwarzaniem danych. Pętla `LOOP` w języku SQL PL ma taką samą składnię jak w omówionych wcześniej językach. Należy jedynie pamiętać, by w przypadku używania etykiety użyć składni deklarowania etykiety odpowiedniej dla SQL PL, czyli `etykieta:` a nie `<<etykieta>>`, jak to ma miejsce w PL/SQL i PL/pgSQL. W celu wyskoczenia z pętli `LOOP` w SQL PL należy zastosować instrukcję `LEAVE etykieta`. Wynika z tego, że pętla musi mieć nadaną etykietę, żeby opuścić ją instrukcją `LEAVE`. Alternatywnie można także skorzystać z `GOTO`, która zostanie omówiona w dalszej części, lecz nie jest to zalecane. Oto przykład bardzo prostej pętli `LOOP` wraz z użyciem `LEAVE`:

```
l1: LOOP
      SET zm_1 = zm_1 + 1;
      IF zm_1 > 10 THEN
        LEAVE l1;
```

```
    END IF;
  END LOOP;
```

Możliwe jest również użycie `RETURN` wewnątrz pętli.
Odpowiednikiem `CONTINUE` z PL/SQL i PL/pgSQL jest tu natomiast instrukcja `ITERATE etykieta`. Składnia pętli `WHILE` w SQL PL wygląda zaś następująco:

```
[etykieta:]
WHILE warunek DO
  operacje;
END WHILE [etykieta];
```

W przeciwieństwie do pętli `WHILE`, która może nie wykonać się ani razu, oferowana przez SQL PL pętla `REPEAT` zawsze wykona się przynajmniej jeden raz. Zachowanie to wynika ze sprawdzania prawdziwości warunku po, a nie przed każdą iteracją. Oto konstrukcja tej pętli:

```
[etykieta:]
REPEAT
  operacje;
UNTIL warunek
END REPEAT [etykieta];
```

oraz przykłady prostych pętli `WHILE` i `REPEAT`:

```
WHILE (v_zm_2 < 7) DO
  SET p_3 = p_3 + v_zm_2;
  SET v_zm_2 = v_zm_2 + 1;
END WHILE;
```

```
REPEAT
  SET p_3 = p_3 + 1;
  SET v_zm_2 = v_zm_2 + 1;
UNTIL (v_zm_2 > 3)
END REPEAT;
```

Składnia pętli `WHILE` różni się od składni tej samej pętli w językach PL/SQL i PL/pgSQL. Słowo kluczowe `LOOP` jest tutaj zastąpione słowem `DO`, a kończące pętlę `END LOOP` słowami `END WHILE`.

Najuboższe możliwości w zakresie pętli oferuje T-SQL. Użytkownik ma do dyspozycji jedynie pętlę `WHILE` wraz z instrukcjami `BREAK` oraz `CONTINUE`. `BREAK` jest odpowiednikiem omawianych uprzednio `EXIT` i `LEAVE`, natomiast `CONTINUE` jest odpowiednikiem instrukcji `CONTINUE` i `ITERATE`. W SQL PL oraz T-SQL nie odnajdziemy odpowiedników `EXIT WHEN` i `CONTINUE WHEN`. Przykład pętli `WHILE` wraz z zastosowaniem instrukcji `BREAK` w języku T-SQL:

```
BEGIN
  DECLARE @licznik INT;
  SET @licznik = 1;
```

```
  WHILE (@licznik < 5)
    BEGIN
      PRINT 'Iteracja: ' + CONVERT(VARCHAR, @licznik);
      SET @licznik = @licznik  + 1;
      BREAK;
      PRINT 'A to się nie wyświetli ani razu.';
    END;
END;
```

Jak wynika z powyższego przykładu, pętla `WHILE` ma w języku T-SQL najprostszą ze wszystkich języków konstrukcję:

```
WHILE warunek
  operacje;
```

Ponownie widoczne jest charakterystyczne dla tego języka użycie minimalnej liczby słów kluczowych, w tym brak jawnego zamknięcia konstrukcji. W języku T-SQL, podobnie jak we wcześniej omówionych, można posłużyć się instrukcją `RETURN` w ciele pętli.

Trzecią grupę struktur sterujących stanowi nawigacja sekwencyjna. Są to instrukcje transferujące z jednego miejsca kodu do innego miejsca kodu. W PL/SQL taką rolę pełni instrukcja `GOTO etykieta`. Jej działanie obrazuje poniższy przykład:

```
DECLARE zmienne;
BEGIN
   operacje;
   (...)
   IF (warunek) THEN
     GOTO etykieta;
   END IF;
   operacje;
   (...)
   <<etykieta>>;
END;
```

Z używaniem instrukcji `GOTO` w języku PL/SQL wiążą się pewne ograniczenia: nie może ona odwoływać się do etykiety w bloku zagnieżdżonym, nie może być użyta do przeskoczenia z sekcji wyjątków do innej sekcji bloku, jeżeli `GOTO` znajduje się poza klauzulą `IF` to nie może wskazywać na etykietę wewnątrz tej struktury, a jeżeli `GOTO` znajduje się wewnętrz `IF` to nie może wskazywać na etykietę znajdującą się w innej takiej strukturze.

Jako drugą instrukcję sterowania sekwencyjnego Oracle wymienia instrukcję `NULL`. Służy ona do jawnego wskazania, że nie będzie wykonana

żadna operacja.
Język PL/pgSQL instrukcji `GOTO` nie oferuje, zaś instrukcję `NULL` tak.

Z `GOTO` możemy również skorzystać w SQL PL i T-SQL. W SQL PL nie można odwołać się za pomocą `GOTO` do etykiety znajdującej się w bloku zagieżdżonym, ale można wskazywać na etykiety znajdujące się wewnątrz struktury `IF` oraz odwoływać się instrukcją `GOTO` z jednej instrukcji `IF` do drugiej. W T-SQL możliwe jest natomiast zarówno wskazywanie na etykiety w blokach zagnieżdżonych, jak i na zawarte w strukturze `IF` oraz przeskakiwanie z jednego `IF` do innego. Poniżej znajduje się przykład wykorzystania etykiet i instrukcji `GOTO` w T-SQL:

```
BEGIN
  DECLARE @x INT;
  SET @x = 0;

  PRINT 'Napis pierwszy';
  GOTO etykieta1;
  PRINT 'Napis drugi';
  etykieta3:
  IF (@x=0)
    BEGIN
      PRINT '@x jest rowne 0';
      RETURN;
    END;
  RETURN;
  PRINT 'Napis trzeci';
  BEGIN
    etykieta1:
    PRINT 'Napis czwarty';
  END;
  GOTO etykieta2;
  IF (@x=100)
    BEGIN
      etykieta2:
      PRINT '@x wcale nie jest równe 100';
      GOTO etykieta3;
    END;
END;
```

Po jego wykonaniu otrzymujemy:

```
Napis pierwszy
Napis czwarty
@x wcale nie jest równe 100
@x jest rowne 0
```

Warto zwrócić tu uwagę na przeskok do `etykiety2`, znajdującej się wewnątrz instrukcji `IF`. Następuje pominięcie sprawdzenia warunku, który gdyby był sprawdzony, nie dopuściłby do wykonania instrukcji zawartych wewnątrz `IF`.

W odróżnieniu od PL/SQL i PL/pgSQL, w SQL PL i T-SQL nie istnieje instrukcja `NULL`, wskazująca, że nie są wykonywane żadne instrukcje.

Dokumentacja SQL PL, w ramach instrukcji sterowania sekwencyjnego, wymienia również omówione wcześniej instrukcje `LEAVE` i `ITERATE`.

Po przedstawieniu tych kilku zagadnień widoczne staje sie duże podobieństwo języków PL/SQL i PL/pgSQL. Pewne elementy wspólne można również odnaleźć w językach SQL PL i T-SQL, choć nie występują już one w takim stopniu jak w poprzedniej parze (PL/SQL i PL/pgSQL). Są też elementy wspólne dla wszystkich czterech omawianych tu języków.

2.3.2 Przetwarzanie danych

Z przetwarzaniem danych wiążą się takie zagadnienia jak pobieranie danych, przeprowadzanie operacji na danych (w tym operowanie na zbiorach danych) oraz transakcje. W celu ułatwienia przetwarzania danych systemy zarządzania bazami danych oraz proceduralne języki SQL oferują wiele funkcji wbudowanych, gotowych do wykorzystania przez programistę.

Podstawową instrukcją pobierania danych w języku PL/SQL jest instrukcja `SELECT INTO`, której składnia wygląda następująco:

```
SELECT lista_wyboru
  INTO {lista_zmiennych | rekord PL/SQL}
  FROM lista_tabel
  [WHERE klauzula_where]
  [GROUP BY klauzula_group_by]
  [HAVING klauzula_having]
  [ORDER BY lista_kolumn];
```

Pobiera ona wartość skalarną (lub wiele wartości) do zmiennej (lub wielu zmiennych). Liczba i typ pobieranych z bazy danych wartości musi odpowiadać liczbie i typom zmiennych, do których wartości są pobierane. Wygodne jest tutaj deklarowanie zmiennych przy użyciu `%TYPE` i `%ROWTYPE`, co zapewnia zgodność typów pobieranych danych z typami zmiennych. Wykorzystując `SELECT INTO` należy pamiętać, że powinna ona zwrócić tylko jeden wiersz danych. W przeciwnym wypadku zgłoszony zostanie wyjątek. Przykładowe wykorzystanie instrukcji `SELECT INTO` w PL/SQL[3]:

[3] Większość tabel wykorzystywanych w przykładach została stworzona jedynie do celów pokazania właściwości proceduralnego języka SQL. Nie są one częścią żadnej zorganizowanej bazy danych. Dlatego są to często pojedyncze, nieznormalizowane tabele.

```
DECLARE
  v_nr_katalogowy ksiazki.nr_katalogowy%TYPE;
  v_tytul ksiazki.tytul%TYPE;
  v_autor ksiazki.autor%TYPE;
BEGIN
  SELECT nr_katalogowy, tytul, autor
    INTO v_nr_katalogowy, v_tytul, v_autor
    FROM ksiazki WHERE UPPER(autor) LIKE '%TOMASZ%';
  DBMS_OUTPUT.PUT_LINE (TO_CHAR(v_nr_katalogowy) ||', '
     || v_tytul ||', ' || v_autor);
END;
/
```

W tym przypadku w bazie znajdował się tylko jeden autor o imieniu Tomasz i wykonanie zakończyło się sukcesem:

```
8735, W pułapce szaleńca, Tomasz Nawrocki
```

Jeżeli zmienimy warunek wyszukiwania tak, by spełniało go więcej wierszy, wykonanie instrukcji `SELECT INTO` zakończy się wyjątkiem `TOO_MANY_ROWS`:

```
DECLARE
  v_nr_katalogowy ksiazki.nr_katalogowy%TYPE;
  v_tytul ksiazki.tytul%TYPE;
  v_autor ksiazki.autor%TYPE;
BEGIN
  SELECT nr_katalogowy, tytul, autor
    INTO v_nr_katalogowy, v_tytul, v_autor
    FROM ksiazki WHERE UPPER(autor) LIKE '%TOM%';
  DBMS_OUTPUT.PUT_LINE (TO_CHAR(v_nr_katalogowy) ||', '
     || v_tytul ||', ' || v_autor);
END;
/

Błąd w linii 1:
ORA-01422: dokładne pobranie zwraca większą liczbę
  wierszy niż zamówiono
ORA-06512: przy linia 6
```

W języku PL/pgSQL również możemy skorzystać z instrukcji `SELECT INTO`. Różni się ona od swojego odpowiednika w PL/SQL tym, że nie zakończy się błędem, jeżeli zapytanie zwróci więcej niż jeden wiersz. W tym przypadku instrukcja weźmie pierwszy z listy wyników i podstawi go do zmiennej. Należy pamiętać o tym, że kolejność w jakiej zostaną zwrócone wiersze z bazy, będzie miała wpływ na to co znajdzie się w zmiennej.

Oto przykład na to, jak można uzyskać różne wyniki nie stosując bądź stosując klauzulę `ORDER BY`. Dysponujemy tabelą `event_type` o dwóch kolumnach: `event_type_val` i `event_type_desc`. Tabela zawiera trzynaście rekordów:

event_type_val	event_type_desc
0	Satellite
1	Flash or Satellite
2	Plane
3	Satellite or Plane
4	Flash!
5	Other
6	Meteor
7	Clouds
8	Hotpixel
9	Saturated star
10	Opened shutter
11	System error
101	Flash or Satellite Rejected

Tworzymy, a następnie uruchamiamy funkcję:

```
CREATE FUNCTION przetwarzanie () RETURNS int AS'
DECLARE
  v_opis event_type.event_type_desc%TYPE;
BEGIN
  SELECT event_type_desc INTO v_opis FROM event_type;
  RAISE NOTICE ''%'', v_opis;
RETURN 1;
END;
'LANGUAGE 'plpgsql';
```

Funkcja wypisze na ekran słowo 'Satellite'. Wystarczy dodać klauzulę `ORDER BY` do zapytania:

```
SELECT event_type_desc INTO v_opis
  FROM event_type ORDER BY event_type_desc;
```

by na ekranie pojawił się napis 'Clouds'. W obu przypadkach zapytanie zwróciło wiele wierszy, a do zmiennej `v_opis` został pobrany pierwszy z nich. Można wymusić, by instrukcja `SELECT INTO` w PL/pgSQL zachowywała się tak, jak w PL/SQL. Służy do tego opcjonalna klauzula `STRICT`. Instrukcja, w której użyto `STRICT` musi zwrócić nie więcej niż jeden wiersz. Jeżeli teraz zapytanie zapiszemy tak:

```
SELECT event_type_desc INTO STRICT v_opis
  FROM event_type ORDER BY event_type_desc;
```

otrzymamy komunikat o błędzie:

```
ERROR:  query returned more than one row.
```

Instrukcja `SELECT INTO` w języku SQL PL zachowuje się tak samo jak w języku PL/SQL – jeżeli zapytanie zwróci więcej niż jeden wiersz danych, to instrukcja zakończy się błędem:

```
Wynik selekcji skalarnej, instrukcja SELECT INTO lub
instrukcja VALUES INTO występuje w więcej niż jednym
wierszu.. SQLCODE=-811, SQLSTATE=21000
```

Stosując klauzulę `FETCH FIRST`, w której określa się liczbę pobieranych wierszy, można uzyskać zachowanie takie jak w języku PL/pgSQL. W klauzuli należy podać jeden wiersz. Zapytanie może wówczas zwrócić wiele wierszy, a pobrany zostanie tylko pierwszy.

```
SELECT event_type_desc INTO v_opis
  FROM MAIN.event_type ORDER BY event_type_desc
  FETCH FIRST 1 ROW ONLY;
```

Jedną instrukcją `SELECT INTO` w językach PL/SQL, PL/pgSQL oraz SQL PL można podstawić wartości do kilku zmiennych, na przykład w języku PL/SQL:

```
SELECT nr_katalogowy, tytul, autor
  INTO v_nr_katalogowy, v_tytul, v_autor ...
```

W języku T-SQL nie można zastosować instrukcji `SELECT INTO` w celu pobrania wartości do zmiennych. Wspiera on jedynie SQL-ową instrukcję `SELECT INTO` służącą do utworzenia i wypełnienia danymi nowej tabeli. W celu pobrania danych z bazy do zmiennej w T-SQL należy skorzystać albo z instrukcji `SET`:

```
DECLARE @tytul VARCHAR(50);
SET @tytul =
  (SELECT tytul FROM ksiazki WHERE nr_katalogowy = 1);
```

albo ze specjalnej składni instrukcji `SELECT`:

```
BEGIN
  DECLARE @tytul VARCHAR(50);
  DECLARE @nr_kat INT;

  SELECT @tytul = tytul, @nr_kat = nr_katalogowy
    FROM ksiazki WHERE nr_katalogowy = 1;
  SELECT @tytul;
  SELECT @nr_kat;
END;
```

W przypadku instrukcji `SET` zapytanie musi zwrócić maksymalnie jeden wiersz. Tą instrukcją można załadować dane tylko do jednej zmiennej. Natomiast w przypadku instrukcji `SELECT @zmienna` zapytanie może zwrócić wiele wierszy. Wówczas do zmiennej `@zmienna` zostanie podstawiona wartość z ostatniego pobranego wiersza. Instrukcją można pobrać dane do wielu zmiennych. W przedstawionym wyżej przykładzie, w wyniku wykonania instrukcji

```
SELECT @tytul = tytul, @nr_kat = nr_katalogowy
  FROM ksiazki WHERE nr_katalogowy = 1;
```

do zmiennych `@tytul` i `@nr_kat` zostaną podstawione wartości: `Tytul1` i `1`. Jeżeli usuniemy waunek `WHERE`:

```
SELECT @tytul = tytul, @nr_kat = nr_katalogowy
  FROM ksiazki;
```

to w zmiennych znajdą się wartości `Tytul5` i `5`, odpowiadające ostatniemu pobranemu z tabeli wierszowi.

Instrukcji `SELECT INTO` w PL/SQL, PL/pgSQL i SQL PL można również użyć w celu przypisania do zmiennych wartości niepochodzących z tabel bazy danych. Na przykład w PL/SQL:

```
SELECT 5, 'jakis tytul', 'jakis autor'
  INTO v_nr_katalogowy, v_tytul, v_autor FROM dual;
```

w PL/pgSQL:

```
SELECT ''napis'', 3 INTO v_opis, v_nr;
```

czy w SQL PL:

```
SElECT 7, 'opis' INTO v_nr, v_opis FROM sysibm.sysdummy1;
```

W T-SQL można natomiast napisać tak:

```
SELECT @tytul = 'Tytul ksiazki', @nr_kat = 7;
```

W powyższych przykładach widać również pewną właściwość: w PL/SQL i SQL PL, żeby pobrać do zmiennych dane niepochodzące z bazy należy skorzystać z tabeli–zaślepki (`dual` w Oracle i `sysibm.sysdummy1` w DB2). W przypadku PL/pgSQL i T-SQL wystarczyło pominąć klauzulę `FROM`.

W SQL PL, podobnie do T-SQL, możliwe jest również pobranie danych z bazy danych i przypisanie ich do zmiennej instrukcją `SET`:

```
SET (v_licznik, v_licznik_1)
  = (SELECT COUNT(*), 5 FROM MAIN.event_type);
```

W przeciwieństwie do T-SQL można w ten sposób podstawić wartości do kilku zmiennych jednocześnie. W PL/pgSQL można natomiast skorzystać z takiego zapisu (tylko dla jednej zmiennej):

```
v_licznik := (SELECT COUNT(*) FROM MAIN.event_type);
```

Poza instrukcjami pobierania danych, w kodzie proceduralnego SQL można również stosować instrukcje `INSERT`, `UPDATE` oraz `DELETE`. Ich użycie polega po prostu na wywołaniu danej instrukcji wewnątrz programu. W PL/SQL może to wyglądać tak:

```
DECLARE
  v_licznik PLS_INTEGER DEFAULT 0;
  v_nr_kat ksiazki.nr_katalogowy%TYPE;
  v_tytul ksiazki.tytul%TYPE;
BEGIN
  INSERT INTO ksiazki VALUES (4567, 2, 'Podróż pociągiem',
    'Halina Borkowska', 2007, 19.33)
    RETURNING nr_katalogowy INTO v_nr_kat;
  DBMS_OUTPUT.PUT_LINE ('Wstawiono książkę o numerze
    katalogowym: ' || TO_CHAR(v_nr_kat));
  UPDATE ksiazki
    SET tytul = 'Bardzo dluga podroz pociagiem'
    WHERE nr_katalogowy = 4567
    RETURNING tytul INTO v_tytul;
  DELETE FROM ksiazki
    WHERE UPPER(autor) LIKE UPPER('Halina Borkowska')
    RETURNING nr_katalogowy INTO v_nr_kat;
  COMMIT;
END;
/
```

Rezultat wykonania tego bloku:

```
Wstawiono książkę o numerze katalogowym: 4567
```

Pokazano tu również zastosowanie klauzuli `RETURNING` do zwracania informacji o zawartości wierszy, których dotyczyło działanie instrukcji DML, po zakończeniu działania tej instrukcji.
Podobny przykład w języku PL/pgSQL:

```
CREATE FUNCTION przetwarzanie_1 () RETURNS int AS'
DECLARE
  v_licznik INTEGER;
  v_nr event_type.event_type_val%TYPE;
  v_opis event_type.event_type_desc%TYPE;
```

```
BEGIN
  INSERT INTO event_type VALUES (999, ''Nowe zdarzenie'')
    RETURNING event_type_val INTO v_nr;
  RAISE NOTICE ''Wstawiono zdarzenie o numerze: %'', v_nr;
  UPDATE event_type
    SET event_type_desc = ''Zmodyfikowany opis''
    WHERE event_type_val = 999
    RETURNING event_type_val, event_type_desc
      INTO v_nr, v_opis;
  DELETE FROM event_type WHERE event_type_val = 999
    RETURNING event_type_desc INTO v_opis;
  RAISE NOTICE ''Usunieto zdarzenie o opisie: %'', v_opis;
RETURN 1;
END;
'LANGUAGE 'plpgsql';
```

Użycie podwojonych cudzysłowów w PL/pgSQL zostanie omówione w rozdziale o funkcjach.
W T-SQL przykładowy kod z zastosowaniem instrukcji DML może wyglądać tak:

```
BEGIN
  DECLARE @licznik INT;
  DECLARE @nr_kat INT;
  DECLARE @tab table(kol_1 VARCHAR(50), kol_2 VARCHAR(50));

  INSERT INTO ksiazki
    OUTPUT INSERTED.tytul, INSERTED.autor INTO @tab
    VALUES ('Tytul6', 'Autor4');
  SET @nr_kat = @@IDENTITY;
  SELECT @nr_kat;

  UPDATE ksiazki SET autor = 'Autor4_zmieniony'
    OUTPUT DELETED.autor, INSERTED.autor INTO @tab
    WHERE nr_katalogowy = @nr_kat;

  DELETE FROM ksiazki
    OUTPUT DELETED.tytul, DELETED.autor INTO @tab
    WHERE nr_katalogowy = @nr_kat;

  SELECT * FROM @tab;
END;
```

Tutaj do zwrócenia informacji o wierszu posłużyła klauzula `OUTPUT`[4].

[4]Należy pamiętać, że składnia instrukcji DML będących częścią języka SQL może być

Na koniec przykład w języku SQL PL:

```
IF (v_hv_count = 0) THEN
  INSERT INTO MAIN.NightStat (ns_night,ns_frame_count)
    VALUES (p_ns_night, v_hv_frame_count);
ELSE
  UPDATE MAIN.NightStat
    SET ns_frame_count = v_hv_frame_count
    WHERE ns_night = p_ns_night;
END IF;

DELETE FROM MAIN.event_type WHERE event_type_val = 999;
GET DIAGNOSTICS v_rows = ROW_COUNT;
```

W tym przypadku użyta została instrukcja `GET DIAGNOSTICS` w połączeniu z `ROW_COUNT` w celu zwrócenia liczby usuniętych wierszy (zmienna `v_rows` jest typu `INT`).

Podczas operowania na danych w bazie danych ważną rzeczą jest zachowanie spójności tych danych. Między innymi do tego celu służą transakcje. Transakcja obejmuje zbiór operacji, które albo w całości muszą zostać zatwierdzone, albo w całości wycofane. Transakcje nie są ograniczone do proceduralnego języka SQL. Są to dowolne zmiany wprowadzone w bazie danych za pomocą poleceń DML.

W Oracle nie ma żadnej specjalnej instrukcji rozpoczynającej transakcję. Nowa transakcja jest rozpoczynana w momencie wydania polecenia DML. Oracle blokuje modyfikowane obiekty do czasu zakończenia transakcji. W celu zakończenia transakcji należy ją zatwierdzić lub wycofać. Do chwili zakończenia transakcji zmiany, które zostały wykonane w transakcji, nie będą widoczne w innych sesjach. Transakcję zatwierdza się instrukcją `COMMIT [WORK]`, zaś wycofuje istrukcją `ROLLBACK [WORK]`. Instrukcja `ROLLBACK` spowoduje anulowanie całej transkcji. Można również wycofać transakcję częściowo. W tym celu należy posłużyć się poleceniem:

```
SAVEPOINT nazwa;
```

Tworzy ono punkt zapisu, do którego można następnie „odwinąć” transakcję poleceniem:

```
ROLLBACK [WORK] TO SAVEPIONT nazwa;
```

Oto przykład obrazujący użycie transakcji w PL/SQL:

```
DECLARE
  v_czy_kasowac BOOLEAN DEFAULT FALSE;
```

różna w poszczególnych systemach zarządzania bazami danych.

```
BEGIN
  INSERT INTO ksiazki VALUES (4567, 2,
    'Podróż pociągiem', 'Halina Borkowska', 2007, 19.33);
  SAVEPOINT S1;

  UPDATE ksiazki
    SET tytul = 'Bardzo dluga podroz pociagiem'
    WHERE nr_katalogowy = 4567;
  SAVEPOINT S2;

  DELETE FROM ksiazki WHERE nr_katalogowy = 4567;

  IF (v_czy_kasowac = FALSE) THEN
    ROLLBACK TO SAVEPOINT S2;
  ELSE
    COMMIT;
  END IF;
END;
/
```

W PL/SQL można ponadto zadeklarować transakcję jako transakcję autonomiczną. Transakcja taka jest rozpoczynana w transakcji nadrzędnej, ale działa bez względu na jej przebieg. Oznacza to, że wycofanie transakcji nadrzędnej nie ma wpływu na zawartą w niej transakcję autonomiczną i odwrotnie. Transakcje autonomiczne wymagają użycia specjalnej dyrektywy – `AUTONOMOUS_TRANSACTION` – w sekcji deklaracji danego bloku. W poniższym przykładzie transakcja autonomiczna zapisze książkę 'Czarodziej Oz' do tabeli, mimo, że transakcja nadrzędna jest zakończona instrukcją `ROLLBACK`:

```
CREATE OR REPLACE PROCEDURE wstaw_1 AS
PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  INSERT INTO ksiazki VALUES (4568, 3,
    'Czarodziej Oz', 'M. Rotz', 2003, 21.00);
  COMMIT;
END;
/

CREATE OR REPLACE PROCEDURE wstaw_2 AS
BEGIN
  INSERT INTO ksiazki VALUES (4569, 3,
    'Krasnoludy', 'W. Janicki', 2000, 20.00);
  wstaw_1;
  ROLLBACK;
EXCEPTION
```

```
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE (SQLERRM);
END;
/

exec wstaw_2;
```

Transakcje autonomiczne wykorzystuje się na przykład do rejestrowania zdarzeń. Pozwalają odnotować fakt zaistnienia próby wykonania danej operacji, bez względu na to, czy operacja ta się powiodła, czy też nie.

Transakcje należy kończyć jawnie. Brak `COMMIT` lub `ROLLBACK` w „zwykłej" transakcji może czasem „przejść niezauważony". W transakcji autonomicznej zawsze doprowadzi to do błędu (błąd pojawi się podczas wykonania, nie przy kompilacji kodu).

Polecenie `SET TRANSACTION` pozwala zaś ustawić właściwości transakcji. Do wyboru mamy poziomy: READ ONLY, READ WRITE, ISOLATION LEVEL READ COMMITED, ISOLATION LEVEL SERIALIZABLE oraz USE ROLLBACK SEGMENT. Na przykład w tak ustawionej transakcji:

```
SET TRANSACTION READ ONLY;
```

wszystkie pobierane dane będą pochodziły z jednej chwili czasowej. Inni użytkownicy mogą w tym czasie wykonywać modyfikacje na tabelach, z których pochodzą dane, modyfikacje te nie będą jednak widoczne w obrębie transakcji. W transakcji typu `READ ONLY` nie można przeprowadzać operacji `INSERT`, `UPDATE` ani `DELETE` (taka próba wywoła wyjątek ORA-01456). Polecenie `SET TRANSACTION` musi być pierwszą instrukcją w transakcji. Jeśli mimo to pojawia się błąd, oznacza to, że istnieje otwarta transakcja. Wówczas przed poleceniem `SET TRANSACTION` należy wydać instrukcję zakończenia transakcji.

W systemie zarządzania bazami danych PostgreSQL transakcje rozpoczyna się instrukcją `BEGIN [WORK | TRANSACTION]`, a kończy poleceniami `COMMIT [WORK | TRANSACTION]` lub `ROLLBACK [WORK | TRANSACTION]`. Wewnątrz transakcji można ustanowić punkt zapisu instrukcją `SAVEPOINT`. Można również ustawić tryb transakcji poleceniem `SET TRANSACTION`. Do wyboru są tryby: ISOLATION LEVEL SERIALIZABLE, ISOLATION LEVEL REPEATABLE READ, ISOLATION LEVEL READ COMMITTED, ISOLATION LEVEL READ UNCOMMITTED, READ WRITE oraz READ ONLY).

Jednak pisząc program w PL/pgSQL nie stosuje się instrukcji otwierających i zamykających transakcję. Pojawienie się tych instrukcji w kodzie programu spowoduje błąd. W języku PL/pgSQL funkcje[5] są wykonywane w ramach transakcji ustanowionej przez zewnętrzne zapytanie, nie mo-

[5] Tworzenie funkcji zostanie omówione w dalszej części pracy.

gą rozpocząć ani zakończyć transakcji, dopóki nie będzie istniał kontekst, w ramach którego mogą być wykonane. Możliwe jest natomiast wielokrotne użycie wewnątrz funkcji instrukcji `SET TRANSACTION`.

```
CREATE FUNCTION transakcje () RETURNS int AS'
BEGIN
  SET TRANSACTION READ WRITE;
  INSERT INTO event_type VALUES (999, ''Nowe zdarzenie'');

  SET TRANSACTION READ ONLY;
  UPDATE event_type
    SET event_type_desc = ''Zmodyfikowany opis''
    WHERE event_type_val = 999;
RETURN 1;
END;
'LANGUAGE 'plpgsql';
```

Wywołując funkcję *transakcje()* otrzymujemy:

```
postgres=# select transakcje();
ERROR:  transaction is read-only
CONTEXT:  SQL statement "UPDATE event_type
  SET event_type_desc = 'Zmodyfikowany opis'
  WHERE event_type_val = 999"
PL/pgSQL function "transakcje" line 7 at SQL statement
postgres=#
```

Wszystkie operacje zawarte w funkcji zostały anulowane. Wiersz (999, 'Nowe zdarzenie') nie został wstawiony do tabeli. Odpowiednik powyższego przykładu w języku PL/SQL przyjmuje następującą postać:

```
CREATE OR REPLACE PROCEDURE set_tran_1  AS
BEGIN
  SET TRANSACTION READ WRITE;
  INSERT INTO ksiazki VALUES (1, 2,
    'Podróż pociągiem', 'Halina Borkowska', 2007, 19.33);
  -- pierwszy COMMIT
  COMMIT;

  SET TRANSACTION READ ONLY;
  UPDATE ksiazki SET tytul = 'Bardzo dluga podroz
    pociagiem' WHERE nr_katalogowy = 1;
  COMMIT;
END;
/
```

W PL/SQL instrukcja `SET TRANSACTION` musi być być pierwszą instrukcją w transakcji, dlatego niezbędne jest zakończenie transakcji przed wydaniem drugiej instrukcji `SET TRANSACTION`. Zakomentowanie pierwszej instrukcji `COMMIT` nie wywoła błędu w czasie kompilacji kodu. Błąd pojawi się dopiero podczas próby wykonania procedury:

```
Błąd w linii 1:
ORA-01453: SET TRANSACTION musi być pierwszą
  instrukcją transakcji
ORA-06512: przy "SYSTEM.SET_TRAN_1", linia 8
ORA-06512: przy linia 1
```

W tym przypadku żadne dane nie zostaną zapisane do tabel bazy danych. Po odkomentowaniu pierwszej instrukcji `COMMIT`, skompilowaniu i ponownym uruchomieniu procedury również otrzymamy komunikat o błędzie:

```
Błąd w linii 1:
ORA-01456: nie można przeprowadzać operacji
  insert/delete/update w ramach
  transakcji READ ONLY
ORA-06512: przy "SYSTEM.SET_TRAN_1", linia 9
ORA-06512: przy linia 1
```

Tym razem jednak do tabeli `ksiazki` został zapisany nowy rekord danych (instrukcja `INSERT`). Natomiast modyfikacja rekordu nie została wykonana (instrukcja `UPDATE`).

Zarządzanie transakcjami w języku SQL PL jest bardziej urozmaicone niż w dotychczas omówionych. Rozpoczynając blok można zastosować słowo kluczowe `ATOMIC`, będzie on wówczas traktowany jako transakcja. W tak zadeklarowanym bloku nie należy jawnie kończyć transakcji. Blok zostanie albo w całości wykonany, albo wycofany.

```
BEGIN ATOMIC
  operacje;
  ...
END;
```

Bloków `ATOMIC` nie można zagnieżdżać w innych blokach tego typu. Transakcjami można natomiast zarządzać jawnie w blokach zadeklarowanych jako `NOT ATOMIC` (jest to domyślne działanie bloku). Można ustawić tak zwane zewnętrzne punkty kontrolne poleceniem `SAVEPOINT`, zwolnić je poleceniem `RELEASE` oraz wykonać `COMMIT` lub `ROLLBACK`:

```
SAVEPOINT nazwa [UNIQUE][ON ROLLBACK RETAIN LOCKS]
  ON ROLLBACK RETAIN CURSORS;
```

```
RELEASE [TO] SAVEPOINT nazwa;

ROLLBACK TO SAVEPOINT nazwa;
```

Opcja `UNIQUE` oznacza, że nazwa punktu nie może być ponownie użyta w transakcji. Opcja `ON ROLLBACK RETAIN LOCKS` spowoduje pozostawienie blokad. Po wydaniu instrukcji `RELEASE` dla danego punktu, nie będzie już możliwe odwinięcie transakcji do tego punktu. Po zwolnieniu punktu można go ponownie utworzyć. Oto przykład wykorzystania transakcji w SQL PL:

```
CREATE PROCEDURE TRANSAKCJE ()
  SPECIFIC TRANSAKCJE
  LANGUAGE SQL
P1: BEGIN
  INSERT INTO MAIN.EVENT_TYPE
    VALUES (999, 'Nowe zdarzenie');
  SAVEPOINT S1 ON ROLLBACK RETAIN CURSORS;

  UPDATE MAIN.EVENT_TYPE
    SET EVENT_TYPE_DESC = 'Zmodyfikowany opis'
    WHERE EVENT_TYPE_VAL = 999;
  SAVEPOINT S2 UNIQUE ON ROLLBACK RETAIN CURSORS;

  DELETE FROM MAIN.event_type WHERE event_type_val = 999;
  ROLLBACK TO SAVEPOINT S2;
  RELEASE SAVEPOINT S2;

  SAVEPOINT S2 UNIQUE ON ROLLBACK RETAIN CURSORS;
  INSERT INTO MAIN.EVENT_TYPE
    VALUES (998, 'Nowe zdarzenie 2');
  ROLLBACK TO SAVEPOINT S2;

  INSERT INTO MAIN.EVENT_TYPE
    VALUES (997, 'Nowe zdarzenie 3');
  COMMIT;
END P1
@
```

Poziomy izolacji (w SQL PL) można określić za pomocą specjalnej klauzuli poleceń DML:

```
INSERT/UPDATE/DELETE ..... WITH {RR|RS|CS|UR},
```

gdzie: `RR` to Repeatable Read, `RS` – Read Stability, `CS` – Cursor Stability a `UR` oznacza Uncommitted Read. Jeżeli w instrukcji, zarówno w statycznym jak i dynamicznym SQL[6], zostanie jawnie wyspecyfikowany po-

[6]Dynamiczny SQL zostanie omówiony w dalszej części pracy

ziom izolacji, wówczas podczas wykonywania zostanie on użyty. Jeżeli kaluzula `WITH` nie zostanie zastosowana, wówczas dla statycznego SQL zostanie przyjęty poziom izolacji z pakietu, z momentu, w którym pakiet był ładowany do bazy. Dla dynamicznego SQL można zastosować polecenie `SET CURRENT ISLATION {UR|CS|RR|RS|RESET}`. Ustawia ono wartość specjalnej zmiennej `CURRENT ISOLATION`. Jeżeli poziom izolacji nie zostanie w ten sposób ustawiony dla danej sesji, to (tak jak dla statycznego SQL) użyty będzie poziom wynikający z pakietu, z momentu kiedy pakiet był ładowany do bazy. Ponadto, dla danej sesji można ustawić opcje, z jakimi będą kompilowane procedury w tej sesji. Służy do tego procedura SET_ROUTINE_OPTS.

W Transact-SQL transakcję rozpoczyna się następująco:

```
BEGIN { TRAN | TRANSACTION }
    [ { nazwa | @zmienna_nazwy }
      [ WITH MARK [ 'opis' ] ][ ; ]
```

lub

```
BEGIN DISTRIBUTED { TRAN | TRANSACTION }
  [ transaction_name | @tran_name_variable ][ ; ]
```

Punkt zapisu tworzy się poleceniem:

```
SAVE { TRAN | TRANSACTION }
  { nazwa_punktu | @zmienna_nazwy_punktu }[ ; ]
```

Zapisać i wycofać traksakcje można poleceniami:

```
COMMIT [ WORK ][ ; ]

COMMIT { TRAN | TRANSACTION }
  [ nazwa | @zmienna_nazwy ] ][ ; ]

ROLLBACK [ WORK ][ ; ]

ROLLBACK { TRAN | TRANSACTION }
  [ nazwa | @zmienna_nazwy
  | nazwa_punktu | @zmienna_nazwy_punktu ][ ; ]
```

Tak wygląda to w praktyce:

```
BEGIN
  DECLARE @Tran VARCHAR(20);
  SET @Tran = 'Transakcja_1';
```

```
    SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

    BEGIN TRANSACTION @Tran
      UPDATE ksiazki SET autor = 'Artur Brokowski'
        WHERE UPPER(tytul) = 'KRZYWY KAPELUSZ';
      WAITFOR DELAY '00:00:05';
    ROLLBACK TRAN @Tran;
  END

  BEGIN
    DECLARE @Tran VARCHAR(20);
    DECLARE @Autor VARCHAR(50);
    SET @Tran = 'Transakcja_2';

    SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;

    BEGIN TRANSACTION @Tran
      SET @Autor = (SELECT autor FROM ksiazki
        WHERE UPPER(tytul) = 'KRZYWY KAPELUSZ');
      PRINT 'Autor: ' + @Autor;
      ...
    COMMIT TRAN @Tran
  END
```

Pierwsza z transakcji wykonuje modyfikacje danych w tabeli `ksiazki`. Transakcja ta zostaje po chwili wycofana. Uruchomiona równolegle z nią druga transakcja odczytuje niezatwierdzone dane:

```
Autor: Artur Brokowski
```

Jak pokazano w przykładzie, tryb wykonania transakcji można ustawić poleceniem:

```
SET TRANSACTION ISOLATION LEVEL { READ UNCOMMITTED |
  READ COMMITTED | REPEATABLE READ | SNAPSHOT |
  SERIALIZABLE}[ ; ]
```

Kursory

Kursory udostępniają podzbiór danych zdefiniowany przez zapytanie.

W języku PL/SQL rozróżniamy dwa typy kursorów: jawne i niejawne. Kursory niejawne są powoływane do życia dla instrukcji DML czy `SELECT...INTO` i są automatycznie obsługiwane przez mechanizm PL/SQL. Kursor jawny definiuje się poleceniem:

```
CURSOR nazwa_kursora [lista_paramerów]
  [RETURN zwracany_typ]
  IS zapytanie
  [FOR UPDATE [OF (lista_kolumn) [NOWAIT]]];
```

Dane pobierane są do pamięci w momencie otwarcia kursora i przechowywane w niej do czasu jego zamknięcia. Zadeklarowanie kursora z użyciem klauzuli `FOR UPDATE` spowoduje zablokowanie rekordów po otwarciu kursora. Rekordy będą dostępne w innych sesjach tylko do odczytu.
Kursory w PL/SQL należy deklarować w sekcji deklaracji bloku. Otwierać można je w sekcji wykonawczej lub sekcji wyjątków. Do otwarcia kursora służy polecenie:

```
OPEN nazwa_kursora [(wartości_parametrów)];
```

Po otwarciu kursora można pobierać jego kolejne wiersze instrukcją:

```
FETCH nazwa_kursora INTO nazwy_zmiennych | rekord_PL/SQL;
```

Na koniec kursor należy zamknąć za pomocą:

```
CLOSE nazwa_kursora;
```

Instrukcja `FETCH` pobiera rekordy pojedynczo. Możliwe jest jednoczesne pobranie wszystkich rekordów kursora do jednej lub więcej kolekcji za pomocą klauzuli `BULK COLLECT`. Ponadto w Oracle mamy do dyspozycji sześć atrybutów kursorów. Dostarczają one informacji o stanie kursora. Są to:

- `%ISOPEN` – Przyjmuje wartość `TRUE`, jeśli kursor jest otwarty. W przeciwnym razie wynosi `FALSE`;
- `%FOUND` – Pozwala sprawdzić, czy instrukcja `%FETCH` pobrała rekord. Jeśli pobrała, argument ma wartość `TRUE`. W przeciwnym razie `FALSE`;
- `%NOTFOUND` – Działa odwrotnie do `%FOUND`. Przyjmuje watość `TRUE` jeśli `%FETCH` nie pobrała rekordu;
- `%ROWCOUNT` – Pozwala w dowolnym momencie sprawdzić liczbę wierszy pobranych do tej pory z kursora;
- `%BULK_EXCEPTIONS` – Udostępnia informacje o wyjątkach zaistniałych podczas operacji masowego pobierania;
- `%BULK_ROWCOUNT` – Zwraca informacje o liczbie wierszy przy pobieraniu masowym.

Po zbiorze rekordów kursora najwygodniej poruszać się za pomocą pętli. W pętlach `LOOP` i `WHILE` kursor trzeba jawnie otworzyć, pobrać rekordy, a nastepnie go zamknąć. Można również skorzystać z pętli `FOR`. W tym

przypadku jest to specjalny, przeznaczony do pracy z kursorami, wariant tej pętli. Umożliwia on nawigację po kursorze bez jawnego wykonywania operacji otwierania kursora, pobierania wierszy oraz zamykania kursora. Nie trzeba też jawnie deklarować zmiennej używanej wewnątrz pętli, ani wykonywać sprawdzenia przy użyciu `%FOUND`. Oto przykłady kursora z pętlą `WHILE` i `FOR`:

```
DECLARE
  z_Ksiazka ksiazki%ROWTYPE;
  CURSOR k_Ksiazki IS
    SELECT * FROM ksiazki;
BEGIN
  OPEN k_Ksiazki;
  FETCH k_Ksiazki INTO z_Ksiazka;
  WHILE k_Ksiazki%FOUND LOOP
    DBMS_OUTPUT.PUT_LINE(z_Ksiazka.tytul ||
      ' , ' || z_Ksiazka.autor);
    FETCH k_Ksiazki INTO z_Ksiazka;
  END LOOP;
  CLOSE k_Ksiazki;
END;
/


DECLARE
  CURSOR k_Ksiazki IS
    SELECT * FROM ksiazki;
BEGIN
  FOR z_Ksiazka IN k_Ksiazki LOOP
    DBMS_OUTPUT.PUT_LINE (z_Ksiazka.tytul ||
      ', ' || z_Ksiazka.autor);
  END LOOP;
END;
/
```

W pętli `FOR` można również zastosować kursor niejawny, dzięki czemu zapis staje się jeszcze krótszy. Traci się jednak możliwość odwoływania do kursora poprzez nazwę:

```
BEGIN
  FOR z_Ksiazka IN (SELECT tytul FROM ksiazki) LOOP
    DBMS_OUTPUT.PUT_LINE (z_Ksiazka.tytul);
  END LOOP;
END;
/
```

Deklarując kursor można uczynić go bardziej elastycznym za pomocą parametrów. Jest to tak zwany kursor sparametryzowany, a zadeklarowane parametry wykorzystywane są w klauzuli `WHERE` zapytania kursora. Parametrom można przypisać wartości domyślne. Deklaracja i otwarcie kursora sparametryzowanego mogą wyglądać następująco:

```
DECLARE
  z_NrKat ksiazki.nr_katalogowy%TYPE;
  z_Tytul  ksiazki.tytul%TYPE;
  z_Autor  ksiazki.autor%TYPE;
  CURSOR k_Ksiazki(p_Autor ksiazki.autor%TYPE) IS
    SELECT nr_katalogowy, tytul, autor
      FROM ksiazki
      WHERE UPPER(autor) LIKE UPPER(p_Autor);
BEGIN
  OPEN k_Ksiazki ('%Tom%');
  FETCH k_Ksiazki INTO z_NrKat, z_Tytul, z_Autor;
  WHILE k_Ksiazki%FOUND LOOP
    DBMS_OUTPUT.PUT_LINE (z_NrKat ||
      ' - ' || z_Tytul || ', ' || z_Autor);
    FETCH k_Ksiazki INTO z_NrKat, z_Tytul, z_Autor;
  END LOOP;
  CLOSE k_Ksiazki;
END;
/
```

Pokazano tu również użycie zwykłych zmiennych (a nie typu `RECORD`) do obsługi kursora.

Zadeklarowanie kursora jako `FOR UPDATE` pozwala zastosować klauzulę `WHERE CURRENT OF` w instrukcjach `UPDATE` i `DELETE`. Oto przykład takiego kursora, wraz z użyciem instrukcji zakończenia transakcji:

```
DECLARE
  z_Ksiazka ksiazki%ROWTYPE;
  CURSOR k_Ksiazki(p_Autor ksiazki.autor%TYPE) IS
    SELECT *
      FROM ksiazki
      WHERE UPPER(autor) LIKE UPPER(p_Autor)
      FOR UPDATE OF tytul;
BEGIN
  OPEN k_Ksiazki ('%Tom%');
  FETCH k_Ksiazki INTO z_Ksiazka;
  WHILE k_Ksiazki%FOUND LOOP
    UPDATE ksiazki SET tytul = tytul || ' - wycofany'
      WHERE CURRENT OF k_Ksiazki;
```

```
    FETCH k_Ksiazki INTO z_Ksiazka;
  END LOOP;
  CLOSE k_Ksiazki;
  --ROLLBACK;
  COMMIT;
END;
/
```

Oprócz kursorów można również posługiwać się tak zwanymi zmiennymi kursora. Są to wskaźniki do obszaru kursora. Można je powiązać z wieloma zapytaniami w obrębie jednego bloku PL/SQL (najpierw z jednym zapytaniem, a następnie z innym). Zmienna kursora może być ograniczona (zwracać konkretny typ) lub nieograniczona (w deklaracji nie podaje się jakiego typu będą zwracane wyniki). Żeby posłużyć się zmienną kursora należy najpierw zadeklarować typ odwołania do kursora (czyli REF CURSOR) a następnie zmienną tego typu. Składnia typu wygląda następująco:

```
TYPE nazwa_typu IS REF CURSOR RETURN typ_rekordu;
```

Oto przykładowa deklaracja zmiennej ograniczonej:

```
TYPE t_Ksiazki IS REF CURSOR RETURN ksiazki%ROWTYPE;
z_Ksiazka t_Ksiazki;
```

oraz przykład wykorzystania nieograniczonej zmiennej kursora:

```
CREATE OR REPLACE PROCEDURE WyswietlGrupy
  (p_NazwaGrupy IN grupy_wiekowe.nazwa_grupy%TYPE) AS
  TYPE t_KursorGrup IS REF CURSOR;
  z_Grupy t_KursorGrup;
  z_GrupaWiekowa grupy_wiekowe%ROWTYPE;
   ...
BEGIN
  IF p_NazwaGrupy IS NULL THEN
    OPEN z_Grupy FOR
      SELECT * FROM grupy_wiekowe;
    ...
  ELSE
    OPEN z_Grupy FOR
      SELECT * FROM grupy_wiekowe
      WHERE nazwa_grupy LIKE '%'||UPPER(p_NazwaGrupy)||'%';
    ...
  END IF;
  FETCH z_Grupy INTO z_GrupaWiekowa;
  ...
END;
/
```

Zmienne kursora można również przekazywać jako parametry w podprogramach. Zmienną taką można na przykład otworzyć w jednym podprogramie a następnie przetworzyć zbiór wynikowy i zamknąć zmienną w innym. Podprogramy mogą być napisane w różnych językach. Zmienne kursorowe można rówież przekazywać pomiędzy serwerem a klientem. Można zadeklarować zmienną po stronie klienta, a otworzyć ją i pobrać dane po stronie serwera, i na odwrót. Zmienne kursorowe nie mogą przyjmować parametrów.

W języku PL/pgSQL dostęp do kursorów odbywa się poprzez zmienne kursorów. Muszą być one definiowane w obrębie bloków transakcji. Można je powołać do życia na dwa sposoby: deklarując zmienną typu `refcursor` lub używając takiej składni (wersja 8.3 PostgreSQL):

```
DECLARE nazwa_kursora
  [ BINARY ] [ INSENSITIVE ] [ [ NO ] SCROLL ]
   CURSOR [ { WITH | WITHOUT } HOLD ] FOR zapytanie
```

Opcjonalne słowo `BINARY` powoduje, że wyniki są zwracane w postaci binarnej. Redukuje to koszty konwersji na serwerze i kliencie. Taki format jest dostępny jedynie z poziomu aplikacji użytkownika, ponieważ aplikacje takie jak `psql` oczekują danych tekstowych. `INSENSITIVE` zapewnia zaś, że kursor jest niewrażliwy na zmiany dokonane w danych już po otwarciu kursora. Opcja `INSENSITIVE` jest domyślnym sposobem działania kursorów w PL/pgSQL, więc nie trzeba jej jawnie używać. Opcja `SCROLL` pozwala na przemieszczanie się do tyłu (`BACKWARD`) po zbiorze kursora. Zadeklarowanie kursora jako `NO SCROLL` blokuje przechodzenie po kursorze do tyłu. Jeżeli w definicji kursora nie zostanie jawnie użyte ani `SCROLL` ani `NO SCROLL`, wówczas, jeżeli plan zapytania jest prosty i niezbyt kosztowny, kursor będzie się zachowywał jakby użyto `SCROLL` i pozwoli na przechodzenie po wierszach w obie strony. Może się jednak okazać, że tak zadeklarowany kursor pozwoli jednynie na przechodzenie w kierunku `FORWARD`, a `BACKWARD` będzie niedostępny. Opcja `WITH HOLD` pozwala na używanie kursora nawet wtedy, gdy transakcja, w której go utworzono, zostanie pozytywnie zakończona (`COMMIT`). `WITHOUT HOLD` mówi, że kursor nie może być użyty poza obrębem transakcji, w której został utworzony. Jest to domyślne działanie kursora, jeżeli ani `WITH HOLD`, ani `WITHOUT HOLD` nie zostanie jawnie użyta. Kursor z opcją `HOLD` jest zamykany albo w wyniku polecenia `CLOSE nazwa_kursora`, albo z chwilą zakończenia sesji. Można również zadeklarować kursor sparametryzowany. Oto przykładowe deklaracje w języku PL/pgSQL:

```
DECLARE
  kursor_1 refcursor;
  kursor_2 CURSOR FOR
    SELECT * FROM event_type_test
    ORDER BY event_type_val
```

```
    FOR UPDATE OF event_type_test;
  kursor_3 CURSOR (et_val INT) FOR
    SELECT * FROM event_type_test
    WHERE event_type_val > et_val
    ORDER BY event_type_val;
```

Od wersji 8.3 PostgreSQL można, tak jak w PL/SQL, po użyciu w zapytaniu kursora klauzuli `FOR UPDATE [OF lista_tabel]`, modyfikować lub usuwać dane za pomocą:

```
UPDATE/DELETE ... WHERE CURRENT OF nazwa_kursora;
```

Instrukcję tę można zastosować, gdy zapytanie kursora nie zawiera złączeń i grupowania.

Do otwarcia kursora służy instrukcja `OPEN`. Zmienną kursora, `refcursor`, można otworzyć na dwa sposoby: `FOR QUERY` oraz `FOR EXECUTE`. Pierwszy sposób wymaga podania instrukcji SQL bezpośrednio w poleceniu `OPEN`:

```
OPEN kursor_1 FOR
    SELECT * FROM event_type ORDER BY event_type_val;
```

W tym przypadku jest tworzony i przechowywany plan wykonania zapytania kursora. Drugi sposób wymaga podania instrukcji SQL jako ciągu literałów. Wykorzystywana jest instrukcja `EXECUTE` do dynamicznego wykonania zapytania kursora:

```
OPEN kursor_1 FOR EXECUTE
    ''SELECT * FROM event_type '' || p_zap ||
    ''ORDER BY event_type_val'';
```

Plan wykonania zapytania w tym przypadku nie jest przechowywany.

Podobnie jak w PL/SQL, do pobierania kolejnych wierszy służy instrukcja `FETCH`:

```
FETCH [ kierunek { FROM | IN } ] nazwa_kursora INTO zmienna;
```

gdzie kierunek może być określony jako:

```
NEXT, PRIOR, FIRST, LAST, ABSOLUTE liczba,
RELATIVE liczba, FORWARD or BACKWARD
```

Bez jawnego określenia kierunku, wiersze pobierane są tak, jakby podano `NEXT`. Po kursorze można się również poruszać bez pobierania wierszy za pomocą instrukcji:

```
MOVE [ kierunek { FROM | IN } ] nazwa_kursora;
```

W momencie rozpoczęcia pisania niniejszej pracy wersja PostgreSQL 8.3 jeszcze nie istniała. Dlatego poniższy przykład powstał dla wersji 8.2.5:

```
CREATE FUNCTION kursory (p_zap VARCHAR(50)) RETURNS INT AS'
DECLARE
  pewna_zmienna RECORD;
  v_event_type_val INT;
  v_event_type_desc VARCHAR(128);
  kursor_1 refcursor;
  kursor_2 CURSOR FOR
    SELECT * FROM event_type_test
    ORDER BY event_type_val
    FOR UPDATE OF event_type_test;
  kursor_3 CURSOR (et_val INT) FOR
    SELECT * FROM event_type_test
    WHERE event_type_val > et_val
    ORDER BY event_type_val;
BEGIN
  RAISE NOTICE ''Operacje na: kursor_2'';
  OPEN kursor_2;
  FETCH kursor_2 INTO pewna_zmienna;
  RAISE NOTICE ''Pewna zmienna: %, %'',
    pewna_zmienna.event_type_val,
    pewna_zmienna.event_type_desc;
  FETCH kursor_2 INTO pewna_zmienna;
  RAISE NOTICE ''Pewna zmienna: %, %'',
    pewna_zmienna.event_type_val,
    pewna_zmienna.event_type_desc;
  --ponizsza instrukcja nie zadziala pod 8.2.5
  --(ta wersja nie obsluguje WHERE CURRENT OF),
  --ale zadziala na wersji 8.3
  --UPDATE event_type_test
  --  SET event_type_desc = ''zmieniony''
  --  WHERE CURRENT OF kursor_2;
  CLOSE kursor_2;

  RAISE NOTICE ''Operacje na: kursor_1'';
  OPEN kursor_1 FOR
    SELECT * FROM event_type ORDER BY event_type_val;
  FETCH kursor_1 INTO v_event_type_val, v_event_type_desc;
  RAISE NOTICE ''Pobrane wartosci: %, %'',
    v_event_type_val, v_event_type_desc;
  CLOSE kursor_1;

  RAISE NOTICE ''Operacje na: kursor_3'';
  OPEN kursor_3 (4);
  FETCH kursor_3 INTO v_event_type_val, v_event_type_desc;
```

```
  RAISE NOTICE ''Pobrane wartosci: %, %'',
    v_event_type_val, v_event_type_desc;
  CLOSE kursor_3;

  RAISE NOTICE ''Operacje na: kursor_1 ze
    skladanym zapytaniem'';
  OPEN kursor_1 FOR EXECUTE
    ''SELECT * FROM event_type '' || p_zap ||
    ''ORDER BY event_type_val'';
  FETCH kursor_1 INTO v_event_type_val, v_event_type_desc;
  RAISE NOTICE ''Pobrane wartosci: %, %'',
    v_event_type_val, v_event_type_desc;
  CLOSE kursor_1;
RETURN 1;
END;
'LANGUAGE 'plpgsql';
```

W rezultacie wywołania funkcji *kursory* otrzymujemy:

```
postgres=# select kursory('WHERE event_type_val > 8');
NOTICE:  Operacje na: kursor_2
NOTICE:  Pewna zmienna: 0, Satellite
NOTICE:  Pewna zmienna: 1, Flash or Satellite
NOTICE:  Operacje na: kursor_1
NOTICE:  Pobrane wartosci: 0, Satellite
NOTICE:  Operacje na: kursor_3
NOTICE:  Pobrane wartosci: 5, Other
NOTICE:  Operacje na: kursor_1 ze skladanym zapytaniem
NOTICE:  Pobrane wartosci: 9, Saturated star
 kursory
---------
       1
(1 row)
```

W przykładzie pokazano:

- trzy sposoby deklarowania kursora,
- trzy sposoby otwierania kursora instrukcją `OPEN`, w tym dwa sposoby otwierania kursora wstępnie zadeklarowanego bez zapytania (czyli jako `refcursor`): `FOR QUERY` oraz `FOR EXECUTE`,
- otwieranie kursora bez parametrów i z parametrem,
- zamykanie kursora instrukcją `CLOSE`,
- używanie zmiennej typu `RECORD` oraz zwykłych zmiennych do pobrania danych z wiersza kursora.

Porównując sposób korzystania z kursorów w dotychczas omówionych językach PL/pgSQL i PL/SQL można zauważyć, że z kursorów w języku PL/pgSQL zadeklarowanych jako `DECLARE nazwa_kursora CURSOR...` korzysta się podobnie jak z kursorów w języku PL/SQL, zaś z kursorów zadeklarowanych instrukcją `DECLARE nazwa_kursora refcursor` podobnie jak ze zmiennych kursorowych w języku PL/SQL.

Na koniec przykład (w języku PL/pgSQL) pętli `FOR` iterującej po zbiorze wyników zapytania oraz pętli `LOOP` z bezpośrednim wykorzystaniem kursora:

```
CREATE FUNCTION kursory_for () RETURNS INT AS'
DECLARE
  v_event event_type%ROWTYPE;
  v_event_t event_type_test%ROWTYPE;
  kursor_2 CURSOR FOR
    SELECT * FROM event_type_test
    ORDER BY event_type_val;
BEGIN
  FOR v_event IN SELECT * FROM event_type
                 ORDER BY event_type_val LOOP
    RAISE NOTICE ''%, %'', v_event.event_type_val,
      v_event.event_type_desc;
  END LOOP;

  OPEN kursor_2;
  LOOP
    FETCH kursor_2 INTO v_event_t;
    EXIT WHEN NOT FOUND;
    RAISE NOTICE ''%, %'', v_event_t.event_type_val,
      v_event_t.event_type_desc;
  END LOOP;
  CLOSE kursor_2;
RETURN 1;
END;
'LANGUAGE 'plpgsql';
```

Do sprawdzenia, czy pomyślnie pobrano kolejny wiersz z kursora, wykorzytstano tu specjalną zmienną `FOUND`. Język PL/pgSQL nie udostępnia specjalnych atrybutów do sprawdzania stanu kursora (jak np. `%ISOPEN` w Oracle).

Podobnie jak PL/SQL, PL/pgSQL pozwala na przekazywanie kursorów (`refcursor`) jako parametrów wejściowych i wyjściowych funkcji. Można również przekazać `refcursor` do programu wywołującego. Wówczas kursor należy otworzyć wewnątrz funkcji. Program wywołujący może pobierać wiersze z kursora oraz zamknąć kursor. Kursor może również zostać zamknięty przez kończącą się transakcję.

W języku SQL PL z kursorów korzysta się podobnie jak w wyżej opisanych językach. Również tutaj kursory można podzielić na jawne i niejawne. Kursor jawny tworzy się instrukcją:

```
DECLARE nazwa_kursora CURSOR
  [WITH HOLD] [WITH RETURN [TO CALLER | TO CLIENT]]
  FOR zapytanie [FOR UPDATE [OF lista_kolumn]];
```

otwiera za pomocą `OPEN nazwa_kursora`, a kolejne wiersze pobiera instrukcją:

```
FETCH [FROM] nazwa_kursora INTO lista_zmiennych;
```

Na koniec kursor należy zamknąć poleceniem `CLOSE nazwa_kursora`. Tak jak w PL/SQL i PL/pgSQL, kursor w SQL PL można wykorzystać w celu przeprowadzania modyfikacji na danych. Należy wówczas skorzystać z klauzuli `FOR UPDATE` w deklaracji kursora, zaś instrukcje DML zapisać w przedstawiony już sposób:

```
UPDATE/DELETE .... WHERE CURRENT OF nazwa_kursora;
```

Oto przykład użycia kursora w języku SQL PL:

```
CREATE PROCEDURE kursory ()
  SPECIFIC kursory
  LANGUAGE SQL
P1: BEGIN
  DECLARE v_event_val INT;
  DECLARE v_event_desc VARCHAR(128);
  DECLARE SQLSTATE CHAR(5);

  DECLARE kursor_1 CURSOR
    FOR SELECT * FROM MAIN.event_type_test
    FOR UPDATE OF event_type_desc;

  OPEN kursor_1;
  FETCH kursor_1 INTO v_event_val, v_event_desc;
  WHILE (SQLSTATE = '00000') DO
    UPDATE MAIN.event_type_test
      SET event_type_desc = event_type_desc||' - zmieniony'
      WHERE CURRENT OF kursor_1;
    FETCH kursor_1 INTO v_event_val, v_event_desc;
  END WHILE;
  CLOSE kursor_1;
END P1
@
```

W przykładzie zaprezentowano również w jaki sposób można sprawdzić, czy został pobrany kolejny wiersz: `SQLSTATE` równe `'00000'` oznacza, że pobranie się powiodło.

Klauzula `WITH HOLD` w deklaracji kursora ma podobne działanie jak w PL/pgSQL, czyli po pomyślym zakończeniu transakcji (`COMMIT`) kursor pozostaje otwarty, a blokady nie są zwalniane. Jeżeli jednak transakcja zakończy się instrukcją `ROLLBACK`, wówczas kursory są zamykane a blokady zwalniane. Z `WITH HOLD` można skorzystać w następujący sposób:

```
DECLARE kursor_1 CURSOR WITH HOLD
  FOR SELECT * FROM MAIN.event_type_test
  FOR UPDATE OF event_type_desc;

OPEN kursor_1;

FETCH kursor_1 INTO v_event_val, v_event_desc;
UPDATE MAIN.event_type_test                      --(1)
  SET event_type_desc = 'zmieniony 1'
  WHERE CURRENT OF kursor_1;
COMMIT;

FETCH kursor_1 INTO v_event_val, v_event_desc;
COMMIT;                                          --(2)
UPDATE MAIN.event_type_test   --(3)
  SET event_type_desc = 'zmieniony 2'
  WHERE CURRENT OF kursor_1;

FETCH kursor_1 INTO v_event_val, v_event_desc;
UPDATE MAIN.event_type_test
  SET event_type_desc = 'zmieniony 3'
  WHERE CURRENT OF kursor_1;
ROLLBACK;                                        --(4)

FETCH kursor_1 INTO v_event_val, v_event_desc;  --(5)

CLOSE kursor_1;
```

W rezultacie wykonania kodu zostaną wykonane modyfikacje tylko z linii (1). Wykonanie `COMMIT` w linii (2) spowoduje, że aktulana pozycją będzie pozycja przed trzecim wierszem i `UPDATE` z linii (3) się nie wykona. `ROLLBACK` z linii (4) zwolni blokady i zamknie kursor, tak, że znajdująca się po nim instrukcja `FETCH` zgłosi błąd.

W języku SQL PL z kursorami związana jest pętla `FOR`, służąca do iteracji po zbiorze wyników zapytań. Wykorzystuje ona niejawnie kursor:

```
FOR v_row AS SELECT ccdx, ccdy FROM MAIN.Measurements
             WHERE star = p_hv_star DO
  SET v_cnt_hot = (SELECT COUNT(*) FROM MAIN.HotPixel
    WHERE ABS(hp_x - v_row.ccdx) <= COALESCE(hp_radius, 1)
    AND ABS(hp_y - v_row.ccdy) <= COALESCE (hp_radius, 1));
  IF (v_cnt_hot > 0) THEN
    RETURN 1;
  END IF;
END FOR;
```

Podobnie jak w PL/SQL i PL/pgSQL kursor w SQL PL może posłużyć do zwrócenia zbioru wyników z procedury. Kursor musi być wówczas zadeklarowany z klauzulą `RETURN`, pozostawiony otwarty (tak jak w PL/SQL i PL/pgSQL), a sama procedura musi zostać zadeklarowana jako zwracająca dynamiczny zbiór wyników (opcja `DYNAMIC RESULT SETS`).

W języku T-SQL deklarując kursor można zastosować składnię zgodną ze standardem SQL 92 (pierwsza z poniższych) lub rozszerzoną składnię T-SQL (druga z poniższych):

```
DECLARE nazwa_kursora [ INSENSITIVE ] [ SCROLL ]
  CURSOR FOR zapytanie
  [ FOR { READ ONLY | UPDATE [ OF lista_kolumn] } ][;]

DECLARE nazwa_kursra CURSOR[ LOCAL | GLOBAL ]
  [ FORWARD_ONLY | SCROLL ]
  [ STATIC | KEYSET | DYNAMIC | FAST_FORWARD ]
  [ READ_ONLY | SCROLL_LOCKS | OPTIMISTIC ]
  [ TYPE_WARNING ]
  FOR zapytanie[ FOR UPDATE [ OF lista_kolumn] ][;]
```

Opcja `STATIC` powoduje, że zostanie utworzona kopia danych zdefiniowanych przez zapytanie kursora. Modyfikacje danych w bazie danych nie będą w kursorze odzwierciedlone. Poprzez kursor zadeklarowany jako `STATIC` nie można też wykonywać modyfikacji danych. Natomiast kursor z opcją `DYNAMIC` odzwierciedla zmiany zachodzące w bazie danych. Liczba wierszy i zawarte w nich dane pobierane instrukcją `FETCH` z kursora mogą ulec zmianie. `DYNAMIC` jest domyślnym sposobem działania kursorów w T-SQL. Tak jak poprzednio, do obsługi kursora służą instrukcje: `OPEN`, `FETCH` i `CLOSE`. Składnia `FETCH` jest następująca:

```
FETCH [ [ NEXT | PRIOR | FIRST | LAST
        | ABSOLUTE { liczba_wierszy }
        | RELATIVE { liczba_wierszy} ] FROM ]
  { {[ GLOBAL ] nazwa_kursora} | @nazwa_zmiennej_kursora }
  [ INTO @lista_zmiennych ] [;]
```

Instrukcja `CLOSE` w T-SQL zamyka kursor i zwalnia blokady, ale pozostawia struktury kursora. W celu całkowitego usunięcia kursora należy po `CLOSE` wydać instrukcję `DEALLOCATE`, która zwolni struktury.

Podobnie do PL/SQL, w języku T-SQL znajdziemy specjalne funkcje skalarne do obsługi kursorów. Są to: `@@FETCH_STATUS`, `@@CURSOR_ROWS` oraz `CURSOR_STATUS`. Poniżej pokazano przykład kursora w T-SQL wraz z użyciem `@@FETCH_STATUS` do obsługi pętli:

```
BEGIN
  DECLARE @nr_kat INT;
  DECLARE @tytul  VARCHAR(50);
  DECLARE @autor  VARCHAR(50);
  DECLARE kursor_1 CURSOR
    FOR SELECT * FROM ksiazki
    FOR UPDATE OF tytul;
  OPEN kursor_1;
  FETCH NEXT FROM kursor_1 INTO @nr_kat, @tytul, @autor;
  WHILE @@FETCH_STATUS = 0
    BEGIN
      UPDATE ksiazki SET tytul = 'tytul zmieniony'
        WHERE CURRENT OF kursor_1;
      FETCH NEXT FROM kursor_1
        INTO @nr_kat, @tytul, @autor;
    END;
  CLOSE kursor_1;
  DEALLOCATE kursor_1;
END;
```

W języku T-SQL można również posługiwać się zmiennymi kursora. Oto przykład deklaracji i dwóch sposobów podstawienia wartości do zmiennej kursora:

```
DECLARE @zmienna_kur CURSOR;
DECLARE kursor_2 CURSOR FOR
  SELECT * FROM ksiazki;
SET @zmienna_kur = kursor_2;

DECLARE @zmienna_kur CURSOR;
SET @zmienna_kur = CURSOR FOR
  SELECT * FROM ksiazki;
END;
```

Po zadeklarowaniu kursora można skorzystać z czterech procedur wbudowanych zwracających informacje o kursorze:

– sp_cursor_list – zwraca listę kursorów obecnie widocznych w danym połączeniu wraz z ich atrybutami,

– sp_describe_cursor – zwraca atrybuty kursora,

– sp_describe_cursor_columns – opisuje kolumny w zbiorze wynikowym kursora,

– sp_describe_cursor_tables – opisuje tabele, z których pobierane są dane do kursora.

W języku T-SQL kursor może być zwrócony jako parametr typu `OUT` przez procedurę. Nie może natomiast być zwrócony przez funkcję.

Reasumując, w omawianych czterech językach można, pomimo pewnych różnic, w dość podobny sposób wykorzystać kursory w kodzie programu.

Funkcje wbudowane

Podczas przetwarzania danych warto pamiętać o wbudowanych funkcjach, udostępnianych przez poszczególne systemy zarządzania bazami danych. Większość z nich jest obsługiwana przez języki proceduralne. Można skorzystać z funkcji znakowych, liczbowych, obsługi dat i konwersji oraz innych. Przykładami funkcji operujących na ciągach znaków w PL/SQL mogą być: `LTRIM`, `SUBSTR` i `UPPER`. Ich odpowiedniki bez trudu znajdziemy w pozostałych językach. Przykłady funkcji liczbowych w SQL PL to: `ABS`, `COS`, `SIN` czy `TRUNC`. Ponownie znajdziemy ich odpowiedniki w PL/SQL, PL/pgSQL i T-SQL. Oto przykład wykorzystania wbudowanej funkcji zwracającej fragment wyrażenia (od zadanej pozycji i o określonej długości) w T-SQL:

```
BEGIN
  DECLARE @zm1 NVARCHAR(50);
  DECLARE @zm2 NVaRCHAR(20);

  SET @zm1 =
    (SELECT tytul FROM ksiazki WHERE nr_katalogowy = 1);
  PRINT '@zm1: ' + @zm1;
  SET @zm2 = SUBSTRING(@zm1, 7, 9);
  PRINT '@zm2: ' + @zm2;
END;
```

Rezultat działania powyższego bloku:

```
@zm1: tytul zmieniony 5
@zm2: zmieniony
```

W przedstawianych w niniejszej pracy przykładach w języku PL/SQL pojawiała się natomiast funkcja konwersji `TO_CHAR`, przekształcająca argument na wartość typu znakowego.

Każdy z omawianych tu języków oferuje obszerny zbiór funkcji wbudowanych. Wiele funkcji nazywa się i działa jednakowo we wszystkich językach. Są też funkcje specyficzne dla każdego z nich.

2.3.3 Tabele i typy tablicowe

Pracując z językiem PL/SQL warto zapoznać się z kolekcjami, czyli strukturami obsługującymi grupy obiektów tego samego typu. Pozwalają one przechowywać zbiory danych w wierszu tabeli lub w zmiennej. Kolekcje to listy, które mogą być uporządkowane bądź nieuporządkowane. W przypadku list uporządkowanych mamy do czynienia z indeksami w postaci niepowtarzalnych numerów porządkowych. W przypadku list nieuporządkowanych do indeksowania wykorzystywane są unikalne identyfikatory (mogą to być liczby czy łańcuchy znaków). Do budowania kolekcji mogą zostać użyte zarówno standardowe typy danych Oracle, jak również podtypy i typy zdefiniowane przez użytkownika. Oracle udostępnia trzy typy kolekcji:

- Tablice asocjacyjne – Znane wcześniej jako tabele indeksowane, a jeszcze wcześniej jako tabele PL/SQL. Elementy nie muszą być ponumerowane po kolei, tylko muszą mieć niepowtarzalne indeksy. Indeksami mogą być unikalne liczby lub łańcuchy znaków. Z tablic asocjacyjnych należy z korzystać wówczas, gdy rozmiar kolekcji jest nieokreślony i tablica nie będzie wykorzystywana w tabeli (tablice asocjacyjne istnieją tylko w PL/SQL i nie można użyć ich w tabeli relacyjnej).

- Tablice zagnieżdżone – Są definiowane jako tablice z sekwencyjną numeracją elementów. W wyniku usuwania rekordów indeksacja elementów może stać się nieciągła. Tablice zagnieżdżone można przechowywać w trwałych tabelach i wykorzystywać w kodzie SQL. Korzysta się z nich wówczas, gdy rozmiar fizyczny kolekcji jest nieznany.

- Tablice `VARRAY` – Tablice z sekwencyją numeracją elementów, działające jak tradycyjne tablice programistyczne. Podobnie jak tablice zagnieżdżone, mogą być przechowywane w trwałych tabelach i wykorzystywane w kodzie SQL. Tablic `VARRAY` używa się gdy znany jest rozmiar kolekcji.

PL/SQL udostępnia kilka operatorów działających na kolekcjach, podobnych do operatorów języka SQL. Są to: `MULTISET EXCEPT` (działający podobnie jak `MINUS` w SQL), `MULTISET INTERSECT`, `MULTISET UNION` oraz `SET` (odpowiednik `DISTINCT` z SQL).
W celu posłużenia się tabelą asocjacyjną należy najpierw zdefiniować typ tabeli wewnątrz bloku PL/SQL a następnie zadeklarować tabelę tego typu. Typ tabeli można zadeklarować w następujący sposób:

```
TYPE nazwa_typu IS TABLE OF typ
  INDEX BY [PLS_INTEGER | BINARY_INTEGER | VARCHAR2(n)];
```

`nazwa_typu` jest tu nazwą nowo definiowanego typu tabeli, `typ` zaś określa typ elementów tabeli. Oto prosty przykład wykorzystania dwóch tabel asocjacyjnych:

```
DECLARE
  TYPE TabelaNapisow IS TABLE OF VARCHAR(30)
    INDEX BY PLS_INTEGER;
  TYPE TabelaKsiazek IS TABLE OF ksiazki%ROWTYPE
    INDEX BY BINARY_INTEGER;
  z_Napisy TabelaNapisow;
  z_Ksiazki TabelaKsiazek;
BEGIN
  z_Napisy(1) := 'W 80 dni dookoła świata';
  z_Napisy(-7) := 'Wakacje z duchami';
  SELECT * INTO z_Ksiazki(1) FROM ksiazki
    WHERE nr_katalogowy = 3434;
  z_Ksiazki(2).tytul := 'Podróż w nieznane';
  z_Ksiazki(2).autor := 'Marek Welg';
  z_Ksiazki.DELETE(2);
END;
/
```

Jak widać, elementy tablicy tworzy się poprzez przypisanie im wartości. W szczególności pokazano tu dwa sposoby przypisywania wartości do elementów tablicy zadeklarowanej przy użyciu operatora `ROWTYPE`. Odwołanie do elementu tablicy ma postać `nazwa_tablicy(indeks)` lub, w przypadku odwołania do pola rekordu, postać `nazwa_tablicy(indeks).pole`. Wartości indeksu nie muszą być sekwencyjne. Odwołanie do nieistniejącego elementu, np.:

```
DBMS_OUTPUT.PUT_LINE
  ('z_Ksiazki(3).tytul = ' || z_Ksiazki(3).tytul);
```

zwróci błąd `ORA-01403 No data found`. Element można usunąć za pomocą metody `DELETE`.

Drugim typem tablic w PL/SQL są tabele zagnieżdżone. Dostęp do nich można uzyskać w języku PL/SQL i SQL. Typ tabeli zagnieżdżonej tworzy się instrukcją:

```
TYPE nazwa_typu IS TABLE OF typ [NOT NULL];
```

Następnie należy zadeklarować i zainicjować zmienną. Inicjowanie odbywa się za pomocą konstruktora o nazwie typu tabeli. Konstruktor przyjmuje argumenty, z których każdy powinien być wartością o typie zgodnym z zadeklarowanym typem elementów tabeli. Argumenty stają się kolejnymi elementami tabeli, rozpoczynając od 1. Przykład tabeli zagnieżdżonej:

```
DECLARE
  TYPE TablicaLiczb IS TABLE OF NUMBER;
```

```
  --tabela NULL
  z_Liczby1 TablicaLiczb;
  --tabela z jednym elementem o wartości NULL
  z_Liczby2 TablicaLiczb := TablicaLiczb();
  --tablica z elementami
  z_Liczby3 TablicaLiczb := TablicaLiczb(-1, 3, 7, 8, 12);
BEGIN
  z_Liczby3(2) := 98;
  DBMS_OUTPUT.PUT_LINE (z_Liczby3(2));

  -- próba przypisania poza zakresem zakończy się błędem
  -- z_Liczby3(6) := 76;

  -- rozszerzenie tablicy o kolejne 4 elementy
  z_Liczby3.EXTEND(4);
  z_Liczby3(6) := 76;
  z_Liczby3(7) := 79;
END;
/
```

W przykładzie utworzono typ tabeli liczb. Następnie zadeklarowano trzy zmienne. Pierwsza z nich jest tabelą pustą, ponieważ nie wywołano konstruktora. Trzecia natomiast jest tablicą 5-cio elementową. Liczba elementów wymienionych w konstruktorze staje się początkowym rozmiarem tabeli. Próba przypisania wartości do 6-stego elementu zakończyłaby się błędem. Tabele zagnieżdżone, mimo, że są nieograniczone, nie pozwalają na przypisywanie wartości do elementu, który nie istnieje, ponieważ spowodowałoby to zwiększenie rozmiaru tabeli. Można jednak zwiększyć rozmiar tabeli za pomocą metody `EXTEND`, której argumentem jest liczba nowych elementów. W zaprezentowanym przykładzie po rozszerzeniu tabeli można przypisać wartości do elementów o indeksach 6 i 7.

Trzecim typem tablic są tablice `VARRAY`. Tablice te tworzy się w następujący sposób:

```
TYPE nazwa_typu IS {VARRAY | VARYING ARRAY} (rozmiar_max)
  OF typ [NOT NULL];
```

Podczas definiowania typu tabel `VARRAY` określa się ich maksymalny rozmiar. Domyślnie w tablicy można umieścić wartosci `NULL`. Żeby umieszczenie `NULL` było niemożliwe, należy jawnie określić to w deklaracji tablicy. Elementy tablicy są numerowane kolejno, począwszy od 1. Poniższy przykład przedstawia sposób deklarowania i inicjowania (za pomocą konstruktora, tak jak w przypadku tabel zagnieżdżonych) tablicy `VARRAY`. Liczba elementów przekazana do konstruktora staje się początkowym rozmiarem tablicy i musi być mniejsza bądź równa rozmiarowi podanemu w definicji typu.

```
DECLARE
  -- Definicja teblicy VARRAY zwierającej
  --trzy elementy (czyli trzy wiersze)
  TYPE napisy_varray IS VARRAY(3) OF VARCHAR(20);

  -- Deklaracja i inicjowanie tablicy
  napisy_1 napisy_varray :=
    napisy_varray('Tablica', 'trzech', 'elementow');
BEGIN
  FOR i IN 1..3 LOOP
    DBMS_OUTPUT.PUT_LINE
      ('napisy_1 [' ||i|| '] = ' || TO_CHAR(napisy_1(i)));
  END LOOP;

  napisy_1(1) := 'Inne';
  napisy_1(2) := 'trzy';
  napisy_1(3) := 'elementy';

  DBMS_OUTPUT.PUT_LINE ('-.-.-.-.-.-.-.-.-.-.-.-.-.-.');

  FOR i IN 1..3 LOOP
    DBMS_OUTPUT.PUT_LINE
      ('napisy_1 [' ||i|| '] = ' || TO_CHAR(napisy_1(i)));
  END LOOP;
END;
/
```

Wynik działania bloku:

```
napisy_1 [1] = Tablica
napisy_1 [2] = trzech
napisy_1 [3] = elementow
-.-.-.-.-.-.-.-.-.-.-.-.-.-.
napisy_1 [1] = Inne
napisy_1 [2] = trzy
napisy_1 [3] = elementy
```

Tablica `VARRAY` (podobnie jak zagnieżdżona) musi zostać zainicjowana. Można zainicjować ją konkretnymi wartościami jak w przykładzie, wartościami `NULL` lub zainicjować pusty zbiór wierszy. Odwołanie do niezainicjowanej tabeli spowoduje błąd:

```
ORA-06531: Odwołanie do nie zainicjowanej kolekcji
```

Próba przypisania wartości do elementu poza zakresem również wywoła błąd. Można użyć metody `EXTEND` do rozszerzenia rozmiaru tabeli. Roz-

miar można jednak zwiększać jedynie do wartości maksymalnej, podanej w deklaracji typu.

W języku PL/SQL można również deklarować kolekcje wielopoziomowe, czyli kolekcje składające się z innych kolekcji. Ich deklaracja wygląda tak jak deklaracja kolekcji jednopoziomowych, z tą różnicą, że typ elementów kolekcji sam jest kolekcją.

```
DECLARE
  -- indeksowana tablica liczb
  TYPE Kolekcja1Poziomowa IS TABLE OF NUMBER
    INDEX BY BINARY_INTEGER;

  -- kolekcja 2 poziomowa indeksowana
  TYPE Kolekcja2PoziomowaI IS TABLE OF Kolekcja1Poziomowa
    INDEX BY BINARY_INTEGER;

  -- kolekcja 2 poziomowa zagnieżdżona
  TYPE Kolekcja2PoziomowaZ IS TABLE OF Kolekcja1Poziomowa;

  -- kolekcja 2 poziomowa VARRAY
  TYPE Kolekcja2PoziomowaV IS VARRAY(5)
    OF Kolekcja1Poziomowa;

  -- deklaracja zmiennej
  z_Liczby Kolekcja2PoziomowaI;
BEGIN
  z_Liczby(1)(1) := 12;
  DBMS_OUTPUT.PUT_LINE (TO_CHAR(z_Liczby(1)(1)));
END;
/
```

W celu umożliwienia zapisania kolekcji (tabeli zagnieżdżonej lub `VARRAY`) w tabeli bazy danych kolekcję należy zadeklarować za pomocą instrukcji `CREATE TYPE` (a nie wewnątrz bloku PL/SQL), np.:

```
CREATE OR REPLACE TYPE TablicaNapisow
  IS VARRAY(20) OF VARCHAR(50 CHAR);
```

a następnie tego typu użyć w instrukcji `CREATE TABLE` do zdefiniowania kolumny lub wiersza.
Kolekcją zapisaną w bazie można manipulować za pomocą instrukcji DML języka SQL. Dotyczy to kolekcji jako całości. Działania na poszczególnych elementach można wykonać za pomocą PL/SQL lub operatorów SQL. Do obsługi kolekcji służą metody kolekcji. Kilka z nich to: `COUNT`, `DELETE`, `EXISTS` (sprawdza, czy element istnieje w kolekcji), `EXTEND` oraz `NEXT`. Metody mogą być wywoływane z różnymi parametrami. Wszystkich metod jest

więcej, większość działa na każdym typie kolekcji, a niektóre mają ograniczony zakres zastosowania (np. nie działają dla tabel asocjacyjnych).

Z kolekcjami związany jest operator `TABLE`. Pozwala on użyć kolekcji tak, jakby była tabelą relacyjną. Operator ten przyjmuje kolekcję jako dane wejściowe i zwraca te same dane w postaci, do której bezpośrednio można skierować zapytanie za pomocą instrukcji `SELECT`. W przedstwionym poniżej przykładzie utworzono typ *TypKsiazki* a następnie typ kolekcji *TablicaKsiazek* oraz tabelę *ksiazki_2*, której elementy będą kolekcjami typu *TablicaKsiazek*. W kolejnym kroku wstawiono kilka przykładowych rekordów do tabeli *ksiazki_2*. Następnie została utworzona funkcja *ZwrocKolekcje*, która zwraca dane z tabeli *ksiazki_2* na podstawie przekazanego parametru.

```
CREATE OR REPLACE TYPE TypKsiazki AS OBJECT
( nr_kat NUMBER(6,0),
  tytul VARCHAR(100) );

CREATE OR REPLACE TYPE TablicaKsiazek
  AS TABLE OF TypKsiazki;

CREATE TABLE ksiazki_2 (
  id      NUMBER(2,0) PRIMARY KEY,
  ksiazki  TablicaKsiazek
)
NESTED TABLE ksiazki STORE AS k_tab;


DECLARE
  z_Ksiazka1 TypKsiazki := TypKsiazki(1, 'Tytul 1');
  z_Ksiazka2 TypKsiazki := TypKsiazki(2, 'Tytul 2');
  z_Ksiazka3 TypKsiazki := TypKsiazki(3, 'Tytul 3');
  z_Ksiazka4 TypKsiazki := TypKsiazki(4, 'Tytul 4');

  z_Ksiazki_1 TablicaKsiazek := TablicaKsiazek(null, null);
  z_Ksiazki_2 TablicaKsiazek := TablicaKsiazek(null, null);
BEGIN
  z_Ksiazki_1(1) := z_Ksiazka1;
  z_Ksiazki_1(2) := z_Ksiazka2;
  z_Ksiazki_2(1) := z_Ksiazka3;
  z_Ksiazki_2(2) := z_Ksiazka4;

  INSERT INTO ksiazki_2 (id, ksiazki)
    VALUES (1, z_Ksiazki_1);
  INSERT INTO ksiazki_2 (id, ksiazki)
    VALUES (2, z_Ksiazki_2);
```

```
  COMMIT;
END;
/

CREATE OR REPLACE FUNCTION ZwrocKolekcje
(p_ID IN ksiazki_2.id%TYPE)
RETURN TablicaKsiazek AS
  v_Kolekcja TablicaKsiazek := TablicaKsiazek(null);
BEGIN
  SELECT ksiazki INTO v_Kolekcja
    FROM ksiazki_2 WHERE id = p_ID;
  RETURN v_Kolekcja;
END;
/
```

Wywołanie funkcji

```
SELECT ZwrocKolekcje(1) FROM dual;
```

daje wynik w postaci:

```
TABLICAKSIAZEK(TYPKSIAZKI(1, 'Tytul 1'),
  TYPKSIAZKI(2, 'Tytul 2'))
```

Po zastosowaniu operatora `TABLE` dane wyglądają tak, jakby pobrano je z tabeli relacyjnej:

```
NR_KAT    TYTUL
------------------
3         Tytul 3
4         Tytul 4
```

Język PL/pgSQL również udostępnia tablice. Ich implementacja różni się jednak od dotychczas przedstawionej dla PL/SQL. Podobnie jak Oracle, PostgreSQL umożliwia definiowanie kolumn tabeli jako tablic wielowymiarowych. Można posługiwać się tablicami w kodzie PL/pgSQL, a następnie zapisywać je do odpowiednich kolumn tabel bazy danych.

Tablicę w PL/pgSQL deklaruje się używając nawiasów kwadratowych po typie danych. Dla tablic jednowymiarowych można użyć konstruktora `ARRAY`. Dopuszczalne są zarówno wbudowane typy danych, jak i typy zdefiniowane przez użytkownika. Deklarując tablicę bez użycia konstruktora nie trzeba podawać jej rozmiaru. W przypadku użycia kostruktora rozmiar tablicy jest wymagany. Rozmiar tablic, choć możliwy do zadeklarowania (a z konstruktorem konieczny), nie jest jednak jeszcze w pełni obsługiwany

(wersja PostgreSQL 8.3). Tablice z zadeklarowanym rozmiarem są traktowane tak samo jak tablice nieograniczone. Nie jest również w pełni obsługiwana liczba zadeklarowanych wymiarów. Przykładowo, po zadeklarowaniu zmiennej jako X-wymiarowej tablicy możliwe jest podstawienie pod tę zmienną tablicy Y-wymiarowej. Podobnie deklarując zmienną będącą tablicą o rozmiarze `X`, można przypisać jej tablicę o innym, większym rozmiarze `Y`. W obu przypadkach nie otrzymamy komunikatu o błędzie. Oto przykłady deklaracji tablic w PL/pgSQL:

```
v_Tablica_1 varchar(20)[];
v_tablica_2 integer ARRAY[3];
v_Tablica_3 integer[2][3];
```

oraz przykład postawienia i odczytania wartości całej tablicy oraz danego elementu:

```
-- 2 sposoby podstawienia wartosci bez konstruktora
v_Tablica_1 := ''{"Pierwszy", "Drugi", "Trzeci"}'';
v_Tablica_1 := ''{Pierwszy, Drugi, Trzeci}'';
-- uzycie konstruktora
v_Tablica_1 := ARRAY[''Raz'', ''Dwa'', ''Trzy''];
-- wyswietlenie calej tablicy
RAISE NOTICE ''v_Tablica_1: %'', v_Tablica_1;
-- i jednego elementu
RAISE NOTICE ''v_Tablica_1[2]: %'', v_Tablica_1[2];
```

Rezultat:

```
NOTICE:  v_Tablica_1: {Raz,Dwa,Trzy}
NOTICE:  v_Tablica_1[2]: Dwa
```

Indeksowanie elementów w tablicy standardowo rozpoczyna się od 1 i biegnie kolejno dalej. Elementy numerowane są z zachowaniem ciągłości numeracji. Pominięcie elementu o jakimś indeksie spowoduje niejawne utworzenie tego elementu i przypisanie mu wartości `NULL`. Można również jawnie użyć innej niż standardowa indeksacji elementów. Poniższy przykład pokazuje jak zastosować inne niż standarowe indeksowanie elementów oraz co się stanie jeżeli pewnien element zostanie pominięty:

```
FOR i IN -2..4 LOOP
  v_Tablica_2[i] := i;
END LOOP;
v_Tablica_2[6] := 6;
```

Po odczytaniu zawartości tablicy otrzymujemy:

```
v_Tablica_2: [-2:4]={-2,-1,0,1,2,3,4,NULL,6}
```

W przykładzie zaprezentowano również brak pełnej funkcjonalności związanej z rozmiarem tablic. Liczba przypisanych elementów odbiega od zadeklarowanego maksymalnego rozmiaru tablicy.

Rozmiar tablicy 1-wymiarowej można dowolnie rozszerzać. Próba odczytania elementów spoza zakresu również się powiedzie. Nieistniejące elementy zostaną przedstawione jako `NULL`. Zachowanie tablicy wielowymiarowej jest odrobinę inne. W poniższym kodzie, próba przypisania wartości do pustej jeszcze tablicy powiedzie się tylko częściowo:

```
v_Tablica_3[1][1] := 1;
v_Tablica_3[1][2] := 2;
```

Wykonanie drugiej linii zakończy się komunikatem o błędnym indeksie. Jeżeli jednak wcześniej zainicjujemy tablicę w ten sposób:

```
 v_Tablica_3 := ARRAY[[1,2,3], [4,5,6], [7,8,9]];
```

wówczas możliwe jest podstawianie wartości pod istniejące elementy. Odwołanie do elementu [1][2] tym razem już się powiedzie.

Jak zostało wcześniej wspomniane, tablicę z poziomu języka PL/pgSQL można zapisać do tabeli bazy danych. Wystarczy, by kolumna tabeli została zadeklarowana jako tablica:

```
CREATE TABLE tablice1 (
  nazwa varchar(20),
  liczby   integer[2][3]);
```

Następnie można zastosować zwykłą instrukcję `INSERT`:

```
INSERT INTO tablice1 VALUES (''Wiersz 1'', v_Tablica_2);
INSERT INTO tablice1 VALUES (''Wiersz 2'', v_Tablica_3);
```

Obie instrukcje zakończą się sukcesem, pomimo iż wstawiane wartości odbiegają od deklaracji kolumny:

```
postgres=# select * from tablice1;
  nazwa   |             liczby
----------+---------------------------------
 Wiersz 1 | [-2:6]={-2,-1,0,1,7,3,4,NULL,6}
 Wiersz 2 | {{1,2,3},{4,5,6},{12,13,9}}
(2 rows)
```

W przypadku gdy programista zastosował indeksy niestandardowe, są one wyświetlane. W drugim wierszu zakres indeksów nie został wyświetlony, co oznacza, że indeksowanie rozpoczyna się od elementu [1][1].

Podczas odczytu wartości z tablicy można również skorzystać z notacji zakresowej, na przykład:

```
RAISE NOTICE''v_Tablica_3[2:3][2:3]: %'',
  v_Tablica_3[2:3][2:3];
RAISE NOTICE''v_Tablica_3[2][2:3]: %'',
  v_Tablica_3[2][2:3];
```

W rezultacie otrzymamy:

```
NOTICE:  v_Tablica_3[2:3][2:3]: {{5,6},{13,9}}
NOTICE:  v_Tablica_3[2][2:3]: {{2,3},{5,6}}
```

Zapisy `v_Tablica_3[2][2:3]` i `v_Tablica_3[1:2][2:3]` są równoznaczne.

PL/pgSQL oferuje kilka funkcji do obsługi tablic. Trzy z nich to: *array_dims*, *array_upper* oraz *array_lower*. Pierwsza zwraca rozmiar tablicy, dwie kolejne indeks górny i dolny dla podanego wymiaru tablicy. Do odczytu indeksów używana jest jednak deklaracja tablicy, a nie jej faktyczny rozmiar. Oto przykład wywołań funkcji wraz z rezultatami:

```
RAISE NOTICE ''Rozmiar tablicy: %',array_dims(v_Tablica_3);
                            ... array_upper(v_Tablica_3, 1);
                            ... array_lower(v_Tablica_3, 1);
                            ... array_upper(v_Tablica_3, 2);
                            ... array_lower(v_Tablica_3, 2);
                            ... array_upper(v_Tablica_3, 3);
                            ... array_lower(v_Tablica_3, 3);
```

```
NOTICE:  Rozmiar tablicy: [1:3][1:3]
NOTICE:  array_upper(v_Tablica_3, 1): 3
NOTICE:  array_lower(v_Tablica_3, 1): 1
NOTICE:  array_upper(v_Tablica_3, 2): 3
NOTICE:  array_lower(v_Tablica_3, 2): 1
NOTICE:  array_upper(v_Tablica_3, 3): <NULL>
NOTICE:  array_lower(v_Tablica_3, 3): <NULL>
```

Inne funkcje to *array_prepend*, *array_append* oraz *array_cat*, za pomocą których można konstruować tablice. Zamiast nich zalecane jest stosowanie operatora konkatenacji. Natomiast do wyszukiwania wartości w tablicy znajdującej się w tabeli bazy danych można użyć, oprócz odwołania do konkretnego indeksu, również operatorów `ALL` i `ANY`:

```
SELECT * FROM tablice1 WHERE 13 = ANY(liczby);
```

Język SQL PL nie udostępnia tablic takich jak w PL/SQL oraz PL/pgSQL. Podobnie język T-SQL. W przypadku języka T-SQL można jednak deklarować zmienne tabelaryczne. Zmienne takie zachowują się jak inne zmienne lokalne. Istnieją jedynie w obrębie bloku kodu, funkcji czy procedury, wewnątrz której zostały zadeklarowane. Deklaruje się je za pomocą instrukcji:

```
DECLARE @nazwa_tabeli TABLE (definicja_kolumn)[;]
```

Możliwe jest zdefiniowanie ograniczeń dla kolumn, takich jak `[NOT] NULL`, `CHECK`, `UNIQUE`, czy `PRIMARY KEY`. Można również zdefiniować ograniczenia poziomu tabeli. Wewnątrz bloku kodu, po zadeklarowaniu tabeli, można obsługiwać ją podobnie jak trwałą tabelę bazy danych. Przykładowa deklaracja i wstawienie danych do zmiennej tabelarycznej może wyglądać następująco:

```
DECLARE @tabClientRecord TABLE
 (clName CHAR(64),
  Company VARCHAR(255),
  fname VARCHAR(255));

INSERT INTO @tabClientRecord(clName, Company, fname)
  SELECT clName, Company, fname FROM inet.dbo.Clients
  WHERE CID = @CID;
```

Zmiennych tabelarycznych nie można zapisać do trwałych tabel bazy danych. W rzeczywistości korzystanie ze zmiennych tabelarycznych niewiele różni się od korzystania z tabel tymczasowych bazy danych.

Tabele tymczasowe

W przypadku gdy nie można skorzystać z tablic programistycznych (dotyczy to w szczególności języka SQL PL), można posłużyć się tabelami tymczasowymi. Są to obiekty bazy danych dostępne we wszystkich omawianych tu systemach zarządzania bazami danych. Z uwagi na fakt, że języki PL/SQL i PL/pgSQL udostępniają tablice programistyczne, tabele tymczasowe w systemach Oracle i PostgreSQL zostaną przedstawione w sposób bardziej zwięzły niż tabele tymczasowe w systemach DB2 i MS SQL Server 2005.

Tabele tymczasowe w systemie Oracle tworzy się za pomocą instrukcji

```
CREATE GLOBAL TEMPORARY TABLE nazwa_tabeli_tymczasowej
...
[ ON COMMIT {DELETE ROWS | PRESERVE ROWS} ];
```

Jest to jedna z form instrukcji `CREATE TABLE`. Większość opcji dostępnych dla tabel trwałych jest dostępna również dla tabel tymczasowych. Są jednak opcje, których dla tabel tymczasowych stosować nie wolno, na przykład partycjonowanie, czy definicja kluczy obcych.

Tabele tymczasowe służą do przechowywania danych w czasie jednej sesji lub jednej transakcji. Definicja tabeli tymczasowej jest trwała i widoczna dla wszystkich sesji. Jednak dane zawarte w tabeli są widoczne jedynie dla sesji, w której zostały wprowadzane do tabeli. Klauzula `ON COMMIT` w definicji

tabeli pozwala określić, czy dane będą istniały w obrębie sesji czy też tylko w obrębie transakcji. Opcja `DELETE ROWS` klauzli `ON COMMIT` powoduje, że wiersze będą usuwane po zakończeniu transakcji. Jest to zachowanie domyślne. Opcja `PRESERVE ROWS` powoduje natomiast, że wiersze są usuwane po zakończeniu sesji. Oto przykład użycia tabeli tymczasowej w Oracle:

```
CREATE GLOBAL TEMPORARY TABLE temp_tab_1 (
  nr     NUMBER(4,0)   PRIMARY KEY,
  tresc VARCHAR2(100) NOT NULL
)
ON COMMIT PRESERVE ROWS;

INSERT INTO temp_tab_1 VALUES (1, 'opis');
SELECT * FROM temp_tab_1;
DROP TABLE temp_tab_1;
```

Sposób tworzenia tabel tymczasowych w systemie PostgreSQL jest bardzo zbliżony do zaprezentowanego dla systemu Oracle. Istrukcja definiująca tabelę przybiera tu postać:

```
CREATE [GLOBAL | LOCAL] {TEMPORARY | TEMP}
  TABLE nazwa_tabeli_tymczasowej
...
[ ON COMMIT {DELETE ROWS | PRESERVE ROWS | DROP} ];
```

Podobnie jak w Oracle, jest to forma instrukcji `CREATE TABLE`. Również nie wszystkie opcje dostępne dla tabel trwałych są dostępne dla tabel tymczasowych. Słowa kluczowe `LOCAL` i `GLOBAL` istnieją jedynie w celu zachowania kompatybilności ze standardem SQL, lecz nie mają praktycznego znaczenia. Ich zastosowanie nie przynosi żadnych efektów.

W odróżnieniu od tabel tymczasowych w Oracle, tabele tymczasowe w PostgreSQL są usuwane wraz z końcem sesji lub transakcji. Tak jak w Oracle, klauzula `ON COMMIT` określa, czy dane w tabeli tymczasowej będą przechowywane w ramach całej sesji, czy tylko w ramach danej transakcji. Opcja `PRESERVE ROWS` zachowuje dane w tabeli przez okres całej sesji. Jest to zachowanie domyślne. Opcja `DELETE ROWS` wymusza usuwanie wierszy po zakończeniu transakcji, zaś dodatkowa opcja `DROP` powoduje usunięcie tabeli po zakończeniu transakcji. Przykład zastosowania tabeli tymczasowej w PostgreSQL:

```
postgres=# create temp table t_3 (
  nr integer,
  tresc varchar(100))
  on commit delete rows;
CREATE TABLE
```

```
postgres=# insert into t_3 values (1, 'przed transakcją');
INSERT 0 1
postgres=# select * from t_3;
 nr | tresc
----+-------
(0 rows)

postgres=# begin transaction;
BEGIN
postgres=# insert into t_3 values (1, 'przed transakcją');
INSERT 0 1
postgres=# select * from t_3;
 nr |      tresc
----+------------------
  1 | przed transakcją
(1 row)

postgres=# commit;
COMMIT
postgres=# select * from t_3;
 nr | tresc
----+-------
(0 rows)

postgres=#
```

Programiści języka SQL/PL, ze względu na brak tablic programistycznych w tym języku, częściej sięgają po mechanizm tabel tymczasowych niż programiści języków PL/SQL czy PL/pgSQL.

Analogicznie do PostgreSQL, tabele tymczasowe w systemie DB2 to tablice deklarowane i istniejące tylko w ramach danego połączenia. Po jego zamknięciu są automatycznie usuwane. Zachowują sie bardzo podobnie do trwałych tabel bazy danych. Żeby można było korzystać z tablic tymczasowych, w bazie danych musi istnieć tymczasowa przestrzeń tabel USER. Można ją, przykładowo, utworzyć w taki sposób:

```
CREATE USER TEMPORARY TABLESPACE USERTEMPSPACE2
MANAGED BY DATABASE
USING (FILE '/tmp/usertempspace2.f1' 500)
EXTENTSIZE 32;
```

Do tabeli tymczasowej ma dostęp jedynie ta aplikacja lub sesja, w której została ona utworzona. Dostęp do tabeli odbywa się poprzez odwołanie przez sesję, stąd dwie aplikacje mogą utworzyć tabele o takich samych nazwach,

a nazwy te wciąż pozostaną unikalne. Możliwość odwołania do tabeli tymczasowej tylko z jednego połączenia eliminuje problemy współdostępu.

Ogólna składania deklaracji tabeli tymczasowej jest następująca:

```
DECLARE GLOBAL TEMPORARY TABLE nazwa_tabeli
  ( definicje_kolumn )
  [WITH REPLACE]
  [ON COMMIT DELETE ROWS | ON COMMIT PRESERVE ROWS]
  NOT LOGGED [IN nazwa_przestrzeni_tabel]
```

Użyta w deklaracji `nazwa_przestrzeni_tabel` musi być nazwą tymczasowej przestrzeni tabel USER. Należy pamiętać, że systemowa tymczasowa przestrzeń tabel `TEMPSPACE1` nie może być używana w celu tworzenia tabel tymczasowych.
W tabeli tymczasowej nie można wykorzystać wszystkich typów danych. Między innymi `BLOB` i `CLOB`, referencji czy typów strukturalnych. Definicje kolumn mogą zawierać ograniczenia, np.: `NOT NULL`. Można również zdefiniować kolumnę `IDENTITY`.
Klauzula `NOT LOGGED` wynika z natury tabel tymczasowych: tabel tych nie da się odzyskać, nie ma też powodu do rejestracji zmian w nich zachodzących.
Opcja `ON COMMIT` pozwala określić, czy po wykonaniu instrukcji `COMMIT` dane w tabeli zostaną zachowane czy usunięte.
Żeby możliwe było powtórne utworzenie tabeli o tej samej nazwie w tej samej sesji, pierwsza tabela musi zostać jawnie usunięta. Alternatywą jest zastosowanie opcji `WITH REPLACE` w deklaracji tabeli. Dobrą praktyką jest używanie jednocześnie opcji `WITH REPLACE` i kasowanie tabeli, gdy przestaje być potrzebna.
Na tabeli, podobnie jak na trwałej tabeli w bazie danych, można utworzyć indeks. Tabele tymczasowe można wykorzystywać w procedurach.
Oto przykład deklaracji prostej tabeli tymczasowej:

```
DECLARE GLOBAL TEMPORARY TABLE tab_msg_CheckClouds
  (nr INTEGER,
   msg VARCHAR (1050))
  WITH REPlACE NOT LOGGED IN USERTEMPSPACE2;
```

oraz instrukcji wstawiającej wiersz i jednocześnie pokazującej sposób odwoływania się do tej tabeli:

```
INSERT INTO session.tab_msg_CheckClouds
  VALUES (v_tab_msg_id, v_msg);
```

Tabelę tymczasową można utworzyć w jednej procedurze, a następnie odwoływać się do niej w innych procedurach w tej samej sesji. Należy jedynie zadbać o odpowiednią kolejność wywoływania procedur.
Tabelę tymczasową można również przekazać do innej procedury:

```
CALL MAIN.output_msg ('session.tab_msg_CheckClouds');
```

W celu zachowania danych zapisanych do tabeli w czasie wykonywania jednej transakcji i udostępnienia ich dla innych transakcji należy, tak jak w Oracle i PostgreSQL, tabelę tymczasową zadeklarować z opcją `ON COMMIT PRESERVE ROWS`, a po wykonaniu instrukcji DML na tabeli umieścić instrukcję `COMMIT`. Wydanie instrukcji `ROLLBACK` może zakończyć się na dwa sposoby:

- Jeżeli tabela tymczasowa została zadeklarowana w procedurze, w której wykonano `ROLLBACK`, wówczas cała tabela zostanie usunięta;
- Jeżeli `ROLLBACK` wykonano w innej procedurze niż zadeklarowano tabelę tymczasową, zostaną usunięte wszystkie dane z tej tabeli.

Na zakończenie pracy z tabelą tymczasową, w celu jej usunięcia, należy wydać instrukcję `DROP`, np.:

```
DROP TABLE session.tab_msg_CheckClouds;
```

Oferowane przez SQL PL tabele tymczasowe są wystarczające na potrzeby przetwarzania danych wewnątrz procedur składowanych. Nie zapiszemy ich jednak do tabel bazy danych, jak można było to uczynić w przypadku kolekcji i tablic w językach PL/SQL i PL/pgSQL.

W MS SQL Server 2005 i związanym z nim języku T-SQL zachowanie podobne do omówionych wyżej tabel SQL PL można uzyskać stosując lokalne tabele tymczasowe. Są one dostępne w ramach danej sesji. Tworząc tabelę tymczasową, należy przed jej nazwą umieścić znak `#`. Oto przykład deklaracji tabeli tymczasowej:

```
BEGIN
  CREATE TABLE #temp_tab
    (kol1 smallint PRIMARY KEY,
     kol2 int DEFAULT (0),
     kol3 nvarchar(30) NULL);

  INSERT INTO #temp_tab VALUES (1, 345, 'pozycja 1');
END;
```

Jeżeli w obrębie tej samej sesji wykonamy instrukcję

```
SELECT * FROM #temp_tab;
```

otrzymamy w wyniku jeden wiersz. Jeżeli tą samą instrukcję wykonamy w innej sesji, wówczas zwrócony zostanie komunikat o błędzie:

```
Msg 208, Level 16, State 0, Line 1
Invalid object name '#temp_tab'.
```

Tabelę tymczasową należy usunąć instrukcją `DROP`:

```
DROP TABLE #temp_tab;
```

Można również utworzyć globalną tabelę tymczasową. Jej nazwa musi zaczynać się od dwóch znaków specjalnych `#`, np.: `##global_temp_tab`. Globalne tabele tymczasowe są dostępne we wszystkich sesjach, bez żadnych ograniczeń. Można je jawnie usunąć w dowolnej sesji. Niejawnie tabela tyczasowa globalna zostanie usunięta przez serwer dopiero wówczas, gdy zakończy się sesja, w której tabela została utworzona, zakończą się wszelkie instrukcje odwołujące się do tej tabeli w innych sesjach oraz gdy zostaną zwolnione wszystkie blokady założone dla tej tabeli.

Zarówno lokalne, jak i globalne tabele tymczasowe są tworzone w bazie `tempdb`.

Podczas tworzenia procedury w języku T-SQL można używać parametrów typu `TABLE`. Problem pojawia się, gdy chcemy by funkcja zwróciła dane o tym typie. Niestety typ zwracany przez funkcję nie może być ani kursorem ani tabelą. Sposobem na ominięcie tego ograniczenia jest zwracanie przez funkcję ciągu znaków, w którym kolejne elementy będą rozdzielone separatorem, np.:

```
;xxx;xxx;xxx;x;xxx;
```

Można wówczas wykorzystać funkcje `CHARINDEX` oraz `PATINDEX` do wyłuskania poszczególnych elementów.

2.3.4 Obsługa błędów

Podczas wykonywania programu mogą pojawić się błędy. Dobrze napisany kod powinien być na nie przygotowany. Oznacza to, że zaistniałe błędy należy przechwycić i obsłużyć.

W języku PL/SQL do obsługi błędów służą wyjątki oraz bloki obsługi wyjątków. Przechwytywać można zarówno standardowe błędy Oracle, jak i błędy niestandardowe, zdefiniowane przez użytkownika. W kodzie programu mogą zaistnieć błędy czasu kompilacji oraz błędy czasu wykonania. Pierwsze z nich są zgłaszane przez kompilator języka PL/SQL i nie mogą być obsłużone w kodzie programu. Drugie mogą zostać obsłużone w sekcji obsługi błędów programu. Odstępstwem od tej reguły jest mechanizm dynamicznego języka SQL. Błędy kompilacji pochodzące z fragmentów kodu zbudowanego za pomocą dynamicznego SQL zostaną zgłoszone w czasie wykonania kodu. Można je obsłużyć w bloku obsługi wyjątków wraz z innymi błędami czasu wykonania.

W chwili zaistnienia błędu zgłaszany jest wyjątek. Instrukcje następujące po instrukcji, która spowodowała błąd nie zostaną wykonane. Sterowanie

przekazywane jest do sekcji obsługi błędów `EXCEPTION`. Wyjątki predefiniowane Oracle nie wymagają deklaracji. Wystarczy je jedynie obsłużyć w sekcji `EXCEPTION`. Wyjątki zdefiniowane przez użytkownika muszą zostać zadeklarowane w sekcji deklaracji (służy do tego specjalny typ `EXCEPTION`). Wywołuje się je jawnie w ciele bloku za pomocą instrukcji `RAISE`.

```
DECLARE
  deklaracja wyjątków użytkownika;
BEGIN
  zgłaszanie błędów;
EXCEPTION
  obsługa błędów;
END;
```

W sekcji obsługi wyjątków należy umieścić cały kod reagujący na błędy w programie (zarówno te predefiniowane, jak i zdefiniowane przez użytkownika). Dzięki temu ciało bloku staje się czytelniejsze, a reakcja na dany błąd jest zdefiniowana jednokrotnie, zamiast występować po każdej instrukcji, która ten błąd może wywołać.

Wyjątki zdefiniowane przez programistę nie muszą wynikać z błędów Oracle. Mogą oznaczać pewien niepożądany stan danych. Przykładowa deklaracja takiego wyjątku:

```
DECLARE
  zle_dane EXCEPTION;
```

Udostępniony przez PL/SQL zbiór wyjątków wbudowanych to wyjątki odpowiadające najczęściej występującym błędom. Wśród nich znajdziemy np.: `DUP_VAL_ON_INDEX`, `INVALID_CURSOR` czy `NO_DATA_FOUND`.

Błędy zgłaszane przez DBMS można powiązać z wyjątkiem użytkownika. Służy do tego dyrektywa `EXCEPTION_INIT`. Poniżej zamieszczono jej składnię i przykład użycia:

```
PRAGMA EXCEPTION_INIT (nazwa_wyjątku, numer_błędu_Oracle);

PRAGMA EXCEPTION_INIT (pobrano_NULL, -1405);
```

gdzie `-1405` to następujący błąd:

```
ORA-01405: pobraną wartością kolumny jest NULL.
```

Możliwe jest indywidualne obsłużenie poszczególnych błędów bądź jednakowe obsłużenie wszystkich błędów, które nie zostały oddzielnie wyszczególnione. Służy do tego następująca konstrukcja sekcji `EXCEPTION`:

```
EXCEPTION
   WHEN nazwa_wyjątku_1 THEN
```

```
      instrukcje_1;
    WHEN nazwa_wyjątku_2 THEN
      instrukcje_2;
    ...
    [ WHEN OTHERS THEN
      instrukcje; ]
END;
```

Kod zawarty w bloku `OTERS` zostanie wykonany dla wszystkich innych wyjątków, niewyszczególnionych wcześniej w kolejnych instrukcjach `WHEN`. Jeden blok obsługi `WHEN` może zawierać kod dla kilku wyjątków. Natomiast jeden wyjątek może być obsługiwany tylko przez jeden blok `WHEN`. Blok `OTERS` musi być wymieniony jako ostatni. Jeżeli zostanie wymieniony jako pierwszy, wówczas przechwyci wszystkie wyjątki zgłoszone w trakcie wykonania programu i następujące po nim bloki obsługi dla konkretnych wyjątków nigdy nie zostaną uruchomione.

Obsługując wyjątki można skorzystać z funkcji `SQLCODE` i `SQLERRM`, zwracających odpowiednio kod błędu i komunikat powiązany z danym błędem. Należy pamiętać, że komunikaty o błędach są przechowywane na stosie i pobierając informację np. za pomocą `SQLERRM` trzeba zadbać o ograniczenie wielkości pobieranych danych. Alternatywnie można użyć funkcji z pakietu `DBMS_UTILITY`: `FORMAT_ERROR_STACK` oraz `FORMAT_ERROR_BACKTRACE`.

Oprócz definiowania wyjątków możliwe jest również definiowanie własnych komunikatów o błędach. W tym celu stosuje się procedurę `RAISE_APPLICATION_ERRROR`. Składnia jej wywołania jest następująca:

```
RAISE_APPLICATION_ERROR (numer_błędu, komunikat_o_błędzie,
  [opcja_zachowania_błędów]);
```

Numer błędu musi być liczbą od -20000 do -20999. Komunikat może zawierać maksymalnie 512 znaków. Ostatni parametr może przyjmować wartość `TRUE` lub `FALSE`. W przypadku wartości `TRUE` nowy błąd zostanie dodany do listy już zgłoszonych. W przypadku `FALSE` nowy błąd zastąpi bieżącą listę błędów. Jako że `RAISE_APPLICATION_ERROR` jest wywoływana wewnątrz programu i treść komunikatu jest tworzona dynamicznie, możliwe jest zbudowanie komunikatu zawierającego dane. Na przykład te, które były przyczyną błędu. Procedury `RAISE_APPLICATION_ERROR` warto używać dla błędów, które mają zostać wyświetlone dla użytkownika. Dla błędów obsługiwanych wewnątrz programu należy używać instrukcji `RAISE`.

Z wyjątkami łączy się też pojęcie propagacji błędów. Przekazywanie wyjątków zgłoszonych w ciele bloku odbywa się następująco:

– jeżeli wyjątek nie zostanie obsłużony w sekcji obsługi wyjątków bieżącego bloku, zostaje on przekazany do bloku nadrzędnego;

– jeżeli w bloku nadrzędnym w sekcji `EXCEPTION` błąd także nie zostanie obsłużony, a kolejnego bloku nadrzędnego już nie ma, wówczas wyjątek zostaje przekazany do środowiska wywołującego.

W ten sposób wyjątek może wędrować poprzez wiele warstw kolejnych bloków nadrzędnych aż do chwili obsłużenia go lub przekazania do środowiska wywołującego. Wyjątki zgłoszone w sekcji deklaracji będź sekcji obsługi błędów są od razu przekazywane do bloku nadrzędnego. W sekcji `EXCEPTION` wyjątki mogą być wywoływane instrukcją `RAISE`. Mogą też pojawić się błędy czasu wykonania. Propagacja dotyczy zarówno błędów zdefiniowanych przez użytkownika, jak i błędów predefiniowanych.

Wyjątki, podobnie jak zmienne, mają określony zasięg. Można definiować je w pakietach PL/SQL. Pojawienie się wyjątku nie kończy transakcji.

Pracując z PL/SQL można również włączyć opcję, dzięki której w trakcie kompilacji kodu będą wyświetlane nie tylko błędy, ale i komunikaty ostrzegawcze. W tym celu należy odpowiednio ustawić wartość parametru kompilacji `PLSQL_WARNINGS`. Dopuszczalne wartości to `ALL`, `SEVERE`, `INFORMATIONAL` oraz `PERFORMANCE`. Na przykład:

```
ALTER SESSION
  SET PLSQL_WARNINGS='ENABLE:PERFORMANCE';

ALTER PROCEDURE bledy
  COMPILE PLSQL_WARNINGS='ENABLE:PERFORMANCE'
  REUSE SETTINGS;

ALTER SESSION
  SET PLSQL_WARNINGS='DISABLE:ALL';
```

Ostrzeżenia nie będą wyświetlane dla bloków anonimowych (niezależnie od wartości parametru).
Oto przykład procedury zawierającej obsługę wyjątków:

```
CREATE OR REPLACE PROCEDURE bledy AS
  zm_1 PLS_INTEGER;
  zm_2 VARCHAR2(100);
  blad_pierwszy EXCEPTION;
  blad_dzielenia_przez_zero EXCEPTION;
  PRAGMA EXCEPTION_INIT(blad_dzielenia_przez_zero, -1476);
BEGIN
  FOR i IN 1..10 LOOP
    zm_1:= i;
    IF (zm_1 = 7) THEN
      RAISE blad_pierwszy;
    END IF;
```

```
    END LOOP;

    zm_1 := 16/0;
  EXCEPTION
    WHEN blad_pierwszy THEN
      DBMS_OUTPUT.PUT_LINE ('Wystąpił błąd pierwszy.');
    WHEN blad_dzielenia_przez_zero THEN
      DBMS_OUTPUT.PUT_LINE
        ('Wystąpił błąd dzielenia przez zero.');
    WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE ('Jakiś inny błąd.');
  END;
  /
```

W wyniku wywołania procedury *bledy* otrzymamy komunikat:

```
Wystąpił błąd pierwszy.
```

W języku PL/pgSQL do przechwytywania i obsługi błędów służy specjalna klauzula bloku BEGIN–END: `EXCEPTION`. Tworząc kod z użyciem tej klauzuli, uzyskujemy strukturę programu znaną już z PL/SQL:

```
[ DECLARE
  deklaracje; ]
BEGIN
  instrukcje;
EXCEPTION
  WHEN nazwa_błędu_1 [ OR nazwa_błędu_2 ... ] THEN
    instrukcje_obsługi_błędów_1;
  WHEN nazwa_błędu_3 [ OR nazwa_błędu_4 ... ] THEN
    instrukcje_obsługi_błędów_2;
  ...
  [ WHEN OTERS THEN
    instrukcje; ]
END;
```

Nazwa błędu musi być nazwą błędu predefiniowanego. Listę dostępnych błędów można znaleźć w dokumentacji PostgreSQL. Działanie bloków `WHEN` i `OTHERS` jest takie samo jak w PL/SQL. Podobnie też, jeżeli błąd zgłoszony w ciele bloku (`BEGIN-END`) nie zostanie obsłużony w klauzuli `EXCEPTION`, zostanie przekazany do bloku nadrzędnego. Błędy zaistniałe w klauzuli obsługi błędów są od razu (tak jak w PL/SQL) przekazywane do bloku nadrzędnego. W przypadku wystąpienia błędu i przekazania sterowania do części `EXCEPTION`, operacje na danych w bazie danych wykonane w ciele bloku są wycofywane.

Wewnątrz bloku `EXCEPTION` można wykorzystać zmienne `SQLSTATE` oraz `SQLERRM` zwracające odpowiednio kod i komunikat błędu. Zmienne te nie istnieją poza obrębem klauzuli `EXCEPTION`.

Oto przykład przechwytywania i obsługi błędów w języku PL/pgSQL:

```
--Przyklad napisany na PostgreSQL 8.2.5
CREATE OR REPLACE FUNCTION bledy1() RETURNS VOID AS
$$
DECLARE
  v_zm_1 int;
BEGIN
  INSERT INTO ksiazki VALUES
    (2, 3, 'Tytul z bledem', 'Weronika Krol', 2007, 67.00);
  v_zm_1 := 17/0;
EXCEPTION
  WHEN division_by_zero THEN
    RAISE NOTICE 'Wystapil blad dzielenia przez zero';
END;
$$
LANGUAGE 'plpgsql';
```

Oraz rezultat wywołania powyższej funkcji:

```
postgres=# select bledy1();
NOTICE:  Wystapil blad dzielenia przez zero
```

Rekord nie został wstawiony do tabeli bazy danych:

```
postgres=# SELECT COUNT(*)
  FROM ksiazki WHERE nr_katalogowy = 2;
 count
-------
     0
(1 row)
```

Do zgłaszania błędów służy instrukcja `RAISE`. Jej nazwa kojarzy się z `RAISE` z języka PL/SQL, jednak `RAISE` z PL/pgSQL jest bardziej podobne do `RAISE_APPLICATION_ERROR` występującego w PL/SQL niż do `RAISE` w tym języku. Składnia `RAISE` w PL/pgSQL jest następująca:

```
RAISE poziom 'tekst_komunikatu' [, wyrażenie [, ...]];
```

Możliwe poziomy to: `DEBUG`, `LOG`, `INFO`, `NOTICE`, `WARNING` oraz `EXCEPTION`. `EXCEPTION` używane jest dla błędów, których zaistnienie oznacza przerwanie transakcji. Pozostałe poziomy służą do zgłaszania wiadomości z różnym priorytetem. Wywołanie `RAISE` powoduje wysłanie komunikatu do programu klienckiego, zapis komunikatu do dziennika zdarzeń PostgreSQL lub oba

jednocześnie.
Wyświetlanie i zapisywanie komunikatów można konfigurować za pomocą funkcji `log_min_messages` i `client_min_messages`. Pierwsza pozwala określić, które poziomy komunikatów będą zapisywane do dziennika PostgreSQL. Druga, natomiast, które poziomy komunikatów będą przesyłane do klienta. Domyślna konfiguracja poziomów jest następująca: `INFO` powoduje tylko wysłanie komunikatu do klienta, `LOG` jednynie zapis do dziennika, natomiast poziomy `EXCEPTION`, `NOTCE` oraz `WARNING` wysłanie informacji do klienta i rejestrację komunikatu w dzienniku (zazwyczaj plik *serverlog*).
Komunikaty `DEBUG` działają wyłącznie dla bazy ustawionej w tryb śledzenia, dla bazy znajdującej się w trybie produkcyjnym są ignorowane.
Przed wyświetlonym lub zapisanym komunikatem podawany jest jego poziom. Wewnątrz komunikatu można natomiast wyszczególnić listę zmiennych i wyrażeń, których wartości mają być w nim umieszczone. Służy do tego znak `%`. Umieszcza się go w `tekscie_komunikatu` instrukcji `RAISE`. W miejsce znaków `%` zostaną podstawione `wyrażenia` podane na końcu instrukcji, w kolejności ich wystąpienia. Instrukcja `RAISE` wystąpiła już wielokrotnie w wielu przykładach w niniejszej pracy. Została również zastosowana w ostatnio pokazanej funkcji *bledy*. Jeżeli w funkcji tej instrukcję

```
RAISE NOTICE 'Dzielenie przez zero';
```

zastąpimy taką:

```
RAISE EXCEPTION 'Exception %, %', SQLSTATE, SQLERRM;
```

otrzymamy:

```
postgres=# select bledy();
NOTICE:  v_zm_1: 3
ERROR:  Exception 22012, division by zero
```

Zobrazowano tu sposób użycia zmiennych `SQLSTATE` i `SQLERRM`.

Mechanizm obsługi błędów w języku SQL PL jest nieco odmienny. Nie ma tu wyróżniej sekcji obsługi wyjątków w bloku, jak jest to w językach PL/SQL i PL/pgSQL. Jest natomiast dostępny kod zaistniałego błędu poprzez zmienne `SQLCODE` i `SQLSTATE`.
`SQLSTATE` to pięciocyfrowy kod wyjątku, zgodny ze standardem ISO/ANSI SQL 92. Kod `SQLCODE` jest numerem błędu według numeracji producenta. Wartości obu zmiennych dotyczą zawsze ostatnio wykonanej instrukcji SQL. Można je sprawdzić w dowolnej części kodu. Dla porównania, w PL/pgSQL kod błędu był dostępny jedynie w części `EXCEPTION` bloku.
W odróżnieniu od PL/SQL i PL/pgSQL, w celu skorzystania z `SQLSTATE` lub `SQLCODE` należy jawnie zadeklarować zmienną `SQLSTATE` bądź `SQLCODE`. Zmienne te mogą być deklarowane jedynie w najbardziej zewnętrznym bloku, czyli w przypadku procedury na samym jej początku:

```
CREATE PROCEDURE przyklad (IN p1 INT, IN p2 INT)
SPECIFIC przyklad
LANGUAGE SQL
BEGIN
  DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
  DECLARE SQLCODE INT DEFAULT 0;

  ....
END
```

Deklaracja zmiennych `SQLCODE` i `SQLSTATE` zawsze wygląda tak samo. Są to zarezerwowane nazwy zmiennych.

Zmienne niosą ze sobą informację o kodzie błędu. Natomiast żeby błąd przechwycić i obsłużyć, należy posłużyć się *uchwytem błędu*. Uchwyty deklaruje się na początku danego bloku i mają one zasięg bloku, w którym zostały zadeklarowane. Są wykonywane gdy wystąpi określony wyjątek. Deklarując uchwyt błędu należy określić typ uchwytu oraz wyjątek, dla którego będzie wywołany. Dostępne typy wyjątków to: `SQLEXCEPTION`, `SQLWARNING`, `NOT_FOUND` oraz wyjątki zefiniowane przez użytkownika. Można również posłużyć się numerem błędu. `SQLEXCEPTION` i `SQLWARNING` są klasami ogólnymi. Ich funkcjonalność odpowiada `OTHERS` w PL/SQL i PL/pgSQL. Typ uchwytu określa natomiast gdzie zostanie przekazane sterowanie po wykonaniu uchwytu (czyli obsłużeniu błędu). Dostępne typy uchwytów to: `EXIT`, `CONTINUE` oraz `UNDO`. Typ `EXIT` powoduje wykonanie uchwytu, a następnie opuszczenie bloku, w którym był zadeklarowany. Typ `CONTINUE` przekazuje sterowanie do instrukcji następującej po tej, która wywołała wyjątek. Typ `UNDO` zachowuje się podobnie jak `EXIT`, z tą różnicą, że wszystkie zmiany w bazie wykonane w opuszczanym bloku zostaną odwinięte. Uchwyty `UNDO` można deklarować jedynie w blokach typu `ATOMIC`. Ogólna składnia deklaracji uchwytu jest następująca:

```
DECLARE { CONTINUE | EXIT | UNDO } HANDLER FOR
  { SQLSTATE kod_wyjątku | nazwa_wyjątku_użytkownika |
    SQLEXCEPTION | SQLWARNING | NOT FOUND }
  instrukcja;
```

Jeżeli w ramach jednego uchwytu ma być wykonane kilka instrukcji, wówczas należy zawrzeć je w bloku `BEGIN`–`END`. Deklaracje uchwytów muszą być ostatnimi deklaracjami ze wszystkich deklaracji w danym bloku.

Jak wspomniano wyżej, uchwyty mogą być deklarowane dla błędów zdefiniowanych przez użytkownika. Definiowanie wyjątku w SQL PL polega na definiowaniu go na podstawie dostępnego `SQLSTATE` lub nazwy wyjątku. Przypomina to wiązanie błędów zdefiniowanych przez użytkownika z błędami predefiniowanymi w PL/SQL. Oto sposób tworzenia własnego wyjątku w SQL PL:

```
DECLARE nazwa_wyjątku CONDITION FOR
  {SQLSTATE kod_wyjątku | nazwa_wyjątku_predefiniowanego};
```

Oraz przykład zadeklarowania wyjątku użytownika a następnie utworzenia dla niego uchwytu:

```
DECLARE FOREGIN_KEY_VIOLATION FOR SQLSTATE '23503';
DECLARE EXIT HANDLE FOR FOREGIN_KEY_VIOLATION;
```

Deklarując uchwyty błędów należy pamiętać o zasadzie „od szczegółu do ogółu" (podobnie jak tworząc kolejne instrukcje `WHEN` w części `EXCEPTION` bloku w językach PL/SQL oraz PL/pgSQL).

Do wywoływania wyjątków w SQL PL służy instrukcja `SIGNAL`. Jest ona odpowiednikiem instrukcji `RAISE_APPLICATION_ERROR` w PL/SQL oraz `RAISE` w PL/pgSQL. Jej deklaracja to:

```
SIGNAL { SQLSTATE kod_wyjątku | nazwa_wyjątku }
  [ SET MESSAGE_TEXT = { nazwa_zmiennej | ciag_znakow } ];
```

Podobnie jak Oracle, IBM określa jakie numery błędów (`SQLSTATE`) mogą być definiowane przez użytkownika.
Jeżeli dla błędu wywołanego instrukcją `SIGNAL` został utworzony uchwyt, wówczas zostanie on użyty. W przeciwnym przypadku zostanie wywołany uchwyt dla `SQLEXCEPTION` lub `SQLWARNING`. Jeżeli one również nie zostały zadeklarowane, wyjątek zostanie przekazany do aplikacji jako błąd bazy danych.
SQL PL udostępnia też, zbliżoną do `SIGNAL`, instrukcję `RESIGNAL`. `RESIGNAL` może być użyta albo do wywołania wyjątku wewnątrz uchwytu błędu (wewnątrz uchwytu instrukcja `SIGNAL` nie może być użyta), albo w celu wznowienia wyjątku. Umożliwia przekazanie odebranego błędu pod innym numerem i z innym komunikatem.

Ponadto, w celu obsłużenia ewentualnych błędów, w SQL PL można skorzystać z instrukcji `GET DIAGNOSTICS`. Zwraca ona informacje dotyczące ostatniej wykonanej instrukcji. Informacje przekazywane są przez zmienne `ROW_COUNT` i `RETURN_STATUS`. Jeżeli `GET DIAGNOSTICS` zastosujemy po instrukcji `INSERT`, `UPDATE` lub `DELETE`, wówczas `ROW_COUNT` będzie zawierała liczbę przetworzonych wierszy. (Instrukcja `GET DIAGNOSTICS` nie jest przeznaczona do użytku w odniesieniu do instrukcji `SELECT`.) Jeżeli zastosujemy ją po instrukcji `PREPARE` [7], zwróci przewidywaną liczbę wierszy. Natomiast po wywołaniu procedury instrukcją `CALL`, zmienna `RETURN_STATUS` będzie zawierała wartość zwróconą przez procedurę instrukcją `RETURN`.

Zamieszczony poniżej przykład procedury obrazuje sposób pracy z błędami w języku SQL PL:

[7] Dynamiczny język SQL zostanie omówiony w dalszej części pracy.

```
CREATE PROCEDURE MAIN.output_msg
                        (IN p_tempTable VARCHAR(50))
  SPECIFIC output_msg
  DYNAMIC RESULT SETS 1
  LANGUAGE SQL
om: BEGIN
  DECLARE v_dynStm VARCHAR(200);

  DECLARE TAB_NOTFOUND CONDITION FOR SQLSTATE '42704';

  DECLARE selectstm STATEMENT;
  DECLARE c_temptab CURSOR
    WITH RETURN TO CLIENT FOR selectstm;

  DECLARE EXIT HANDLER FOR TAB_NOTFOUND
    RESIGNAL SQLSTATE '70000'
    SET MESSAGE_TEXT = 'MAIN.output_msg: Table not found';

  DECLARE EXIT HANDLER FOR SQLEXCEPTION
    RESIGNAL SQLSTATE '70001'
    SET MESSAGE_TEXT = 'MAIN.output_msg: SQLEXCEPTION';

  SET v_dynStm = 'SELECT * FROM ' || p_tempTable;
  PREPARE selectstm FROM v_dynStm;
  OPEN c_temptab;

END om
@
```

Język T-SQL obecnie również oferuje rozbudowany mechanizm obsługi błędów. Do wersji SQL Server 2000 głównym narzędziem pracy z błędami była funkcja `@@ERROR`. Od wersji SQL Server 2005 Microsoft wprowadził nowy, podobny do istniejących w innych systemach zarządzania bazami danych, mechanizm. Nadal można jednak korzystać z funkcji `@@ERROR`.

Funkcja `@@ERROR` zwraca liczbę całkowitą, reprezentującą sposób zakończenia ostatniej, poprzedzającej ją instrukcji. Zero oznacza sukces, każda inna liczba – błąd. `@@ERROR` często używana jest w połączeniu z `@@ROWCOUNT`, funkcją zwracającą liczbę wierszy przetworzonych przez ostatnią instrukcję. Należy pamiętać, że w celu jednoczesnego skorzystania z `@@ERROR` oraz `@@ROWCOUNT` należy zrobić to jednym poleceniem:

```
instrukcja_mogąca_spowodować_błąd;
SELECT @blad = @@ERROR, @liczba_wierszy = @@ROWCOUNT;
```

Użycie dwóch osobnych instrukcji:

```
instrukcja_mogąca_spowodować_błąd;
SET @blad = @@ERROR;
SET @liczba_wierszy = @@ROWCOUNT;
```

spowoduje, że warość zwrócona przez `@@ROWCOUNT` będzie dotyczyła poprzedzającej ją instrukcji `SET` a nie instrukcji, która może być zródłem błędu.

Jednym z elementów nowego mechanizmu obsługi błędów jest struktura `TRY - CATCH`. Są to dwa bloki:

```
BEGIN TRY
   instrukcje;
END TRY

BEGIN CATCH
   instrukcje;
END CATCH
```

W bloku `TRY` należy umieszczać instrukcje, które mają być wykonane w ramach logiki biznesowej. Blok `CATCH` jest miejscem, gdzie należy umieścić kod obsługi błędów, które mogą wystąpić podczas przetwarzania bloku `TRY`. W przypadku wystąpienia błędu w bloku `TRY`, następuje jego przechwycenie i przekazanie sterowania do najbliższego bloku `CATCH`. Jeżeli w bloku `TRY` nie wystąpi błąd, blok `CATCH` zostanie pominięty.
Mechanizm `TRY - CATCH` przechwytuje błędy o poziomie ważności wyższym niż 10, niepowodujące zamknięcia połączenia z bazą danych.
Propagacja wyjątków jest tu podobna do opisanej dla poprzednich języków. Jeżeli błąd z bloku `TRY` nie zostanie obsłużony w skojarzonym z nim bloku `CATCH`, wówczas zostanie przekazany do bloku nadrzędnego. Jeżeli blok najwyższego poziomu nie będzie zawierał kodu do obsłużenia propagowanego błędu, błąd zostanie przekazany do środowiska wywołującego.

W języku T-SQL do każdego błędu przypisany jest poziom ważności. Poziomy 0–9 to raczej informacje niż błędy, poziomy 10–19 błędy, zaś 20–25 błędy krytyczne, uniemożliwiające normalną pracę systemu zarządzania bazami danych (np. awarie sprzętu). Mechanizm obsługi błędów pozwala na przechwytywanie wszystkich błędów, z wyjątkiem błędów o poziomie ważności 20 i wyższym.

Podczas pracy z T-SQL można skorzystać z oferowanego przez ten język zestawu funkcji do obsługi wyjątków. Są to:

- `ERROR_NUMBER()` – zwracająca numer błędu,
- `ERROR_MESSAGE()` – zwracająca komunikat skojarzony z błędem,
- `ERROR_SEVERITY()` – zwracająca poziom ważności błędu,
- `ERROR_STATE()` – zwracająca stan błędu,

– ERROR_LINE() – zwracająca numer linii błędu

– oraz ERROR_PROCEDURE() – zwracająca nazwę procedury, w której wystąpił błąd.

Powyższe funkcje są przeznaczone do używania wewnątrz bloku CATCH. Poza nim zwracają wartość NULL.

W celu przekazania błędu należy posłużyć się instrukcją RAISERROR:

```
RAISERROR ( { nr_błędu | komunikat | @nazwa_zmiennej }
  { ,poziom_ważności ,stan }
  [ ,argumenty ] )
  [ WITH { LOG | NOWAIT | SETERROR } ]
```

nr_błędu jest numerem zdefiniowanym przez użytkownika i przechowywanym w katalogu sys.messages. Błędy definiowane przez programistów powinny mieć numery większe niż 50000. Jeżeli numer błędu nie zostanie jawnie wyspecyfikowany, RAISERROR przekaże błąd o numerze 50000.
komunikat to treść informacji, jaka zostanie przekazana przez RAISERROR. Jeżeli treść komunikatu jest zdefiniowana, wówczas przekazywany jest błąd o numerze 50000. Zamiast podawać komunikat jako ciąg znaków, można użyć do tego zmiennej odpowiedniego typu (@nazwa_zmiennej).
Stan jest liczbą od 1 do 127.

W celu dodania komunikatu dla określonego numeru błędu do katalogu sys.messages należy posłużyć się wbudowaną procedurą sp_addmessage:

```
sp_addmessage [ @msgnum = ] nr_błędu ,
              [ @severity = ] poziom_ważności ,
              [ @msgtext = ] 'komunikat'
              [ , [ @lang = ] 'język' ]
              [ , [ @with_log = ] 'TRUE' | 'FALSE'  ]
              [ , [ @replace = ] 'ciąg_nadpisujący' ]
```

Ciąg nadpisujący należy zdefiniować, jeżeli komunikat dla danego numeru błędu nr_błędu już istnieje. Oto przykład zastosowania procedury składowanej sp_addmessage:

```
EXEC sp_addmessage @msgnum = 50005, @severity = 16,
  @msgtext = N'Komunikat dla numeru błędu 50005.';
```

Informacje można następnie pobrać instrukcją:

```
SELECT * FROM sys.messages WHERE message_id = 50005;
```

Można również posłużyć się instrukcją FORMATMESSAGE, która konstruuje komunikat na podstawie już istniejącego w sys.messages:

```
EXEC sp_addmessage @msgnum = 50012, @severity = 16,
  @msgtext = N'Komunikat %s dla numeru błędu 50012.';
DECLARE @zm1 VARCHAR(100)
SELECT @zm1 = FORMATMESSAGE(50012, 'CZERWONY')
```

Zmienna `@zm1` będzie zawierała ciąg:

```
Komunikat CZERWONY dla numeru błędu 50012.
```

Zamieszczony poniżej przykład prezentuje obsługę wyjątków w języku T-SQL:

```
DECLARE @tab table(kol_1 INT NOT NULL, kol_2 INT NOT NULL);

BEGIN TRY
  INSERT INTO @tab VALUES (1, 10);
  INSERT INTO @tab VALUES (2, 20);
  INSERT INTO @tab VALUES (3, 'trzy');
  SELECT * FROM @tab;
END TRY

BEGIN CATCH
  PRINT CHAR(10) + 'Opis uzyskany funkcjami ERROR_NUMBER()
    i ERROR_MESSAGE():';
  PRINT CONVERT(VARCHAR(6), ERROR_NUMBER()) + ' - ' +
    CONVERT(VARCHAR(100), ERROR_MESSAGE()) + CHAR(10);
  IF (ERROR_NUMBER() = 245) -- błąd konwersji typów
    BEGIN
      PRINT 'Podano ciąg znakowy zamiast liczby.';
      RAISERROR ('Wystąpił błąd.', 16, 1);
      --RAISERROR (50005, 16, 1);
  END;
END CATCH
```

W rezultacie jego wykonania otrzymujemy:

```
Opis uzyskany funkcjami ERROR_NUMBER() i ERROR_MESSAGE():
245 - Conversion failed when converting the varchar value
  'trzy' to data type int.

Podano ciąg znakowy zamiast liczby.
Msg 50000, Level 16, State 1, Line 16
Wystąpił błąd.
```

Jeżeli zakomentujemy pierwszą instrukcję `RAISERROR` a odkomentujemy drugą, rezultat będzie taki:

```
Opis uzyskany funkcjami ERROR_NUMBER() i ERROR_MESSAGE():
245 - Conversion failed when converting the varchar value
  'trzy' to data type int.

Podano ciąg znakowy zamiast liczby.
Msg 50005, Level 16, State 1, Line 17
Komunikat dla numeru błędu 50005.
```

W MS SQL Server 2005 sesja może znajdować się w trzech stanach: bez otwartej transakcji, w stanie z aktywną i zatwierdzalną transakcją oraz w stanie transakcji nieudanej. Z trzecim stanem mamy do czynienia gdy wystąpi błąd w bloku `TRY` podczas przetwarzania jawnie otwartej, aktywnej i zatwierdzalnej transakcji. Wówczas transakcja przechodzi w stan *martwy*: pozostaje otwarta i utrzymywane są blokady, ale nie można jej już jawnie zatwierdzić. Transakcja w stanie martwym nie może modyfikować danych, ale może je odczytywać. Przed wykonaniem modyfikacji należy transakcję wycofać.
Przyczyną przejścia transakcji w stan transakcji nieudanej są zwykle błędy o poziomie ważności równym 17 lub wyższym. Ustawienie opcji `XACT_ABORT` sesji na wartość `ON` powoduje, że wszystkie błędy będą powodowały przejście transakcji w stan martwy. Można wówczas odpytać dane w celu ustalenia przyczyny błędu, a dopiero później transakcję wycofać.
Do sprawdzania bieżącego stanu transakcji służy funkcja `XACT_STATE()`. Zwraca ona 0 jeśli nie istnieje żadna otwarta transakcja, 1 dla transakcji aktywnej i zatwierdzalnej oraz -1 dla transakcji aktywnej ale niezatwierdzalnej.

Podsumowując, obsługa błędów w każdym systemie zarządzania bazami danych jest inna. Istnieją jednak podobieństwa. Najbardziej zbliżone do siebie pod tym względem są języki PL/SQL i PL/pgSQL.

Rozdział 3

Proceduralny SQL – zaawansowane możliwości

3.1 Funkcje i procedury składowane

Funkcje i procedury to bloki nazwane. Często określa się je jedną wspólną nazwą – podprogramy. Podprogramy są składowane w bazie danych i można je bezpośrednio wywoływać w innych blokach proceduralnego języka SQL. Zarówno procedury jak i funkcje mogą przyjmować parametry i zwracać wartości. Wywołanie procedury jest niezależną instrukcją. Funkcje natomiast muszą być wywoływane jako część wyrażenia.

W języku PL/SQL można tworzyć obie formy podprogramów: funkcje oraz procedury. Jako, że są one blokami proceduralnego SQL, mogą zawierać sekcję deklaracji, sekcję wykonawczą oraz sekcję obsługi wyjątków. Niezbędna jest tylko sekcja wykonawcza. Pozostałe dwie są opcjonalne. Ponadto w podprogramach nie stosuje się słowa kluczowego `DECLARE` w celu rozpoczęcia sekcji deklaracji. Podstawowa składnia instrukcji tworzącej procedurę jest następująca:

```
CREATE OR REPLACE PROCEDURE nazwa_procedury
  [lista_parametrów]
{IS | AS}
  deklaracje;
BEGIN
  instrukcje;
EXCEPTION
  obsługa_wyjątków;
END [nazwa_procedury];
```

Składnia instrukcji tworzącej funkcje jest bardzo podobna. Różni się klauzulą `RETURN`:

```
CREATE OR REPLACE FUNCTION nazwa_funkcji
  [lista_parametrów] RETURN typ_zwracanej_wartości
{IS | AS}
  deklaracje;
BEGIN
  instrukcje;
EXCEPTION
  obsługa_wyjątków;
END [nazwa_funkcji];
```

Słowo kluczowe `REPLACE` pozwala na zastąpienie istniejącego już podprogramu jego nową wersją, bez potrzeby uprzedniego usuwania go, czyli wydawania instrukcji:

```
DROP {PROCEDURE | FUNCTION} nazwa_podprogramu;
```

Zwracanie wartości przez funkcję odbywa się za pomocą, umieszczanej w ciele funkcji, instrukcji `RETURN wyrażenie`. Typ wyrażenia powinien być zgodny z typem określonym w definicji funkcji w klauzuli `RETURN`. Instrukcji `RETURN` może wewnątrz funkcji być kilka, jednak wykonana zostanie tylko jedna z nich. Instrukcję `RETURN` (już bez `wyrażenia`) można także zastosować w procedurze.

Zarówno funkcje jak i procedury mogą przyjmować parametry w trzech trybach: wejściowym, wyjściowym i wejściowo-wyjściowym. Składnia deklaracji parametru jest następująca:

```
nazwa_parametru [IN | OUT | IN OUT] [NOCOPY] typ_parametru
  { := | DEFAULT } wartość_domyślna
```

Jeżeli tryb parametru nie zostanie jawnie określony, wówczas stosowany jest tryb domyślny, czyli `IN`.
Parametry w trybie `IN` wewnątrz podprogramu nie mogą być modyfikowane. Można je jedynie odczytywać. Wartości parametrów w trybie `OUT` w chwili wywołania podprogramu są ignorowane (działają jak niezainicjowana zmienna o wartości `NULL`). Wewnątrz podprogramu można je zapisywać i odczytywać. Parametry `IN OUT` są połączeniem dwóch powyższych typów. Mogą być od samego początku odczytywane i modyfikowane.
Parametry występujące w definicji podprogramu nazywa się *parametrami formalnymi*, zaś użyte w wywołaniu podprogramu określa się mianem *argumentów*. Parametry formalne w momencie wywołania podprogramu przyjmują wartości argumentów. W wywołaniu podprogramu parametry typu `OUT` i `IN OUT` muszą być zmiennymi, gdyż są lokalizacją, do której zapisywana jest zwracana wartość. Ponadto, definiując parametry formalne, nie można ograniczyć typów `CHAR` i `VARCHAR2` co do długości oraz typu `NUMBER` co do precyzji lub skali. W tym przypadku używane są ograniczenia z argumentów.

W celu uniknięcia niespójności, tam gdzie to możliwe, warto deklarować parametry przy użyciu operatorów `%TYPE` oraz `%ROWTYPE`. Liczba parametrów jest nieograniczona.
Parametry można przekazywać na dwa sposoby: przez wartość lub przez referencję. Przekazywanie przez referencję jest szybsze, ponieważ pozwala uniknąć kopiowania. Warto z niego korzystać zwłaszcza dla parametrów w postaci kolekcji. Domyślnie PL/SQL przesyła parametry `IN` przez referencję, zaś `OUT` oraz `IN OUT` przez wartość. Modyfikator `NOCOPY` w definicji parametru formalnego jest wskazówką dla kompilatora, by przekazać parametr przez referencję, a nie przez wartość. Należy pamiętać, że użycie `NOCOPY` dla parametru `IN` wywoła błąd. Wskazówka `NOCOPY` może zostać zignorowana i nie skutkuje to błędem. Dzieje się tak na przykład wtedy, gdy argument jest elementem tablicy asocjacyjnej.
Jeżeli podczas wykonywania podprogramu wystąpi błąd i nie zostanie on obsłużony w bloku `EXCEPTION`, sterowanie jest przekazywane do środowiska wywołującego. W tej sytuacji, jeżeli parametry formalne typu `OUT` i `IN OUT` były zadeklarowane bez modyfikatora `NOCOPY`, ich wartości nie są zwracane do argumentów. Wartości argumentów pozostają niezmienione. Jeżeli był użyty modyfikator `NOCOPY`, wówczas wartości argumentów ulegną zmianie.

Podprogram można wywołać stosują jedną z dwóch notacji argumentów: opartą na pozycji lub opartą na nazwach. W pierwszej z nich pozycje argumentów muszą odpowiadać pozycjom parametrów formalnych. Notacja oparta na nazwach wymaga podania par *parametr* => *arument* i pozwala na zmianę kolejności argumentów w wywołaniu podprogramu. Notacje można łączyć, na przykład:

```
nazwa_procedury(arg1, p_3 => arg3, p_2 => arg2);
```

Jeżeli podprogram nie przyjmuje parametrów, wówczas ani w jego deklaracji, ani w wywołaniu nie stosuje się nawiasów.

Podprogramy można wywoływać na dwa sposoby. Wewnątrz bloków języka PL/SQL procedury wywołuje się poprzez podanie nazwy procedury (ewentualnie wraz z parametrami), zaś funkcje wywoływane są jako część wyrażeń:

```
BEGIN
  MojaProcedura ('Marta');
  z_Nazwisko := MojaFunkcja('Marta');
END;
```

Z poziomu kodu SQL podprogramy składowane można wywołać za pomocą instrukcji `CALL`:

```
CALL nazwa_podprogramu (lista_argumentów)
  INTO zmienna_serwera.
```

Zmienna serwera służy do pobierania wartości zwracanych przez funkcje. W przypadku instrukcji `CALL` nawiasy w wywołaniu są zawsze potrzebne, nawet jeśli podprogram nie przyjmuje parametrów.
Stosowanie instrukcji `CALL` w blokach języka PL/SQL jest niedozwolone, poza jednym przypadkiem: można jej używać w dynamicznym języku SQL (`EXECUTE IMMEDIATE`).
Funkcje można również wywoływać w instrukcjach SQL.

O typie podprogramu (czy powinien być procedurą, czy funkcją) decyduje logika biznesowa. Ponadto można przyjąć zasadę, że jeżeli podprogram ma zwracać więcej niż jedną wartość, wówczas powinien być procedurą. Jeżeli zaś ma zwracać tylko jedną wartość, wówczas może być funkcją. Pomimo, że funkcje, tak samo jak procedury, mogą zwracać więcej wartości poprzez parametry, to takie rozwiązania są oznaką złego stylu programowania i nie są zalecane.

W sekcji deklaracji podrogramów składowanych można deklarować podprogramy lokalne. Podprogramy lokalne można wywoływać tylko w podprogramie, w którym zostały zadeklarowane. Deklaracja podprogramu musi być umieszczona na końcu sekcji deklaracji. Jeżeli podprogramy lokalne są od siebie zależne, konieczne jest zastosowanie deklaracji uprzedzających. Podprogramy lokalne można przeciążać, podobnie jak podprogramy w pakietach.

Pakiet to struktura języka PL/SQL umożliwiająca przechowywanie powiązanych ze sobą obiektów w jednym miejscu. Składa się z dwóch odrębnych części: specyfikacji oraz ciała. Każda z nich jest oddzielnie przechowywana w słowniku bazy danych. Pakiety muszą być składowane w bazie, nie można definiować ich lokalnie.

Przykłady procedur w języku PL/SQL pojawiały się w niniejszej pracy już kilkakrotnie, przy okazji omawiania innych zaganień. Poniżej przedstawiono przykład procedury z parametrem `OUT`:

```
CREATE OR REPLACE PROCEDURE PobierzKlienta
                (p_IdKlienta IN klienci.id_klienta%TYPE,
                 p_Klient OUT NOCOPY VARCHAR2) AS
  z_imie klienci.imie%TYPE;
  z_nazwisko klienci.nazwisko%TYPE;
BEGIN
  SELECT imie, nazwisko INTO z_imie, z_nazwisko
    FROM klienci WHERE id_klienta = p_IdKlienta;
  p_Klient := z_imie || ' ' || z_nazwisko;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE
      ('Nie istnieje klient o podanym numerze.');
   WHEN OTHERS THEN
```

```
    DBMS_OUTPUT.PUT_LINE ('Jakiś inny błąd.');
END;
/
```

Oto sposób i rezultat wywołania procedury *PobierzKlienta*:

```
DECLARE
  z_IdKlienta klienci.id_klienta%TYPE;
  z_Klient VARCHAR(103);
BEGIN
  z_IdKlienta := 3;
  PobierzKlienta (z_IdKlienta, z_Klient);
  DBMS_OUTPUT.PUT_LINE
    ('Klient (' || z_IdKlienta || '): ' || z_Klient);

  z_IdKlienta := 6;
  PobierzKlienta
    (p_Klient => z_Klient, p_IdKlienta => z_IdKlienta);
  DBMS_OUTPUT.PUT_LINE
    ('Klient (' || z_IdKlienta || '): ' || z_Klient);
END;
/

Klient (3): Alicja Wolska
Klient (6): Waldemar Pawlikowski
```

Pokazano tu wykorzystanie notacji pozycyjnej oraz notacji opartej na nazwach parametrów formalnych.

Poniższy przykład przedstawia funkcję w języku PL/SQL:

```
CREATE OR REPLACE FUNCTION PobierzMiastoKlienta
               (p_IdKlienta IN klienci.id_klienta%TYPE)
               RETURN klienci.miasto%TYPE AS
  z_miasto klienci.miasto%TYPE;
BEGIN
  SELECT miasto INTO z_miasto FROM klienci
    WHERE id_klienta = p_IdKlienta;
  IF z_miasto IS NULL THEN
    z_miasto := 'Brak danych';
  END IF;
  RETURN z_miasto;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    RETURN ('Brak klienta.');
  WHEN OTHERS THEN
```

```
    RETURN ('Wystąpił błąd.');
END;
/
```

Przykładowe wywołanie funkcji *PobierzMiastoKlienta* wraz z rezultatem:

```
DECLARE
  z_miasto klienci.miasto%TYPE;
BEGIN
  z_miasto := PobierzMiastoKlienta(11);
  DBMS_OUTPUT.PUT_LINE
    ('Miasto klienta (11): ' || z_miasto);
  z_miasto := PobierzMiastoKlienta(3);
  DBMS_OUTPUT.PUT_LINE
    ('Miasto klienta (3): ' || z_miasto);
END;
/

Miasto klienta (11): Brak danych
Miasto klienta (3): Gdańsk
```

Jak wspomniano już wcześniej, powiązane ze sobą obiekty można zebrać w pakiet. Poniżej przedstawiono przykładową specyfikację oraz fragment ciała pakietu zawierającego cztery funkcje:

```
CREATE OR REPLACE PACKAGE AlgorytmyKontrolne AS

  PRAGMA RESTRICT_REFERENCES
    (DEFAULT, WNDS, RNDS, WNPS, RNPS);
  FUNCTION SprawdzRegon (p_Regon IN VARCHAR2)
    RETURN BOOLEAN;
  FUNCTION SprawdzNip (p_Nip IN VARCHAR2) RETURN BOOLEAN;
  FUNCTION SprawdzPesel (p_Pesel IN VARCHAR2,
    p_DataUr IN DATE, p_Plec IN CHAR) RETURN BOOLEAN;
  FUNCTION SprawdzPesel(p_Pesel IN VARCHAR2)
    RETURN BOOLEAN;

END AlgorytmyKontrolne;
/


CREATE OR REPLACE PACKAGE BODY AlgorytmyKontrolne AS

  FUNCTION SprawdzRegon (p_Regon IN VARCHAR2)
    RETURN BOOLEAN AS
```

```
    z_Dlugosc NUMBER;
    z_Wynik NUMBER;
    z_Poprawny BOOLEAN := FALSE;
  BEGIN
    ... ciało funkcji ...
  END SprawdzRegon;


  FUNCTION SprawdzNip (p_Nip IN VARCHAR2)
    RETURN BOOLEAN AS
    z_Wynik NUMBER;
    z_Poprawny BOOLEAN := FALSE;
  BEGIN
    ... ciało funkcji ...
  END SprawdzNip;


  FUNCTION SprawdzPesel
    (p_Pesel IN VARCHAR2, p_DataUr IN DATE, p_Plec IN CHAR)
    RETURN BOOLEAN IS
    ...
  BEGIN
    ... ciało funkcji ...
  END SprawdzPesel;


  FUNCTION SprawdzPesel(p_Pesel IN VARCHAR2)
    RETURN BOOLEAN IS
    ...
  BEGIN
  ... ciało funkcji ...
  END SprawdzPesel;

END AlgorytmyKontrolne;
/
```

Funkcja *SprawdzPesel* jest funkcją przeciążoną.
Definiując obiekt w ciele pakietu nie używa się słów kluczowych `CREATE OR REPLACE`.

Podczas pierwszego wywołania obiektu z pakietu (np. wywołania programu lub użycia zmiennej) pakiet jest inicjowany. Często zachodzi potrzeba wykonania pewnych operacji podczas inicjacji pakietu, na przykład przypisania początkowych wartości do zmiennych pakietowych (każda sesja ma własną kopię takich zmiennych). Realizuje się to, poprzez dodanie do cia-

ła pakietu sekcji *kodu inicjującego*. Kod ten należy umieścić po wszystkich innych obiektach:

```
CREATE OR REPLACE PACKAGE BODY nazwa_pakietu {IS | AS}
  ...
BEGIN
  kod_inicjujący;
END [nazwa_pakietu];
```

Ponadto, wszystkie obiekty zadeklarowane w nagłówku pakietu są widoczne poza pakietem. Obiekty zadeklarowane jedynie w ciele pakietu są widoczne tylko w obrębie ciała pakietu (są obiektami lokalnymi dla pakietu) i nie można się do nich odwoływać spoza pakietu.

Pakiety w pokazanej tu postaci istnieją jedynie w języku PL/SQL. (Z pojęciem *pakietu* można również spotkać się w bazie DB2, lecz ma ono inne znaczenie.) Funkcje zdefiniowane w pakietach, podobnie jak funkcje samodzielne, mogą być wywoływane w instrukcjach SQL.

W instrukcjach SQL można wywoływać funkcje składowane zwracające jedną bądź wiele wartości. W przypadku funkcji składowanych zdefiniowanych przez użytkownika, zwracających jedną wartość skalarną, funkcje te wywoływane są w taki sam sposób jak funkcje wbudowane. Funkcje takie muszą jednak spełniać określone kryteria, by można było je wywoływać w instrukcjach SQL. Między innymi:

– funkcja wywoływana w instrukcji `SELECT` nie może modyfikować żadnych tabel bazy danych,

– funkcja wywoływana w instrukcji DML (`INSERT`, `UPDATE` lub `DELETE`) nie może kierować zapytań do tabel związanych z danym poleceniem ani modyfikować ich (może używać innych tabel),

– funkcja może przyjmować wyłącznie parametry w trybie `IN`, parametry `OUT` i `IN OUT` są niedozwolone.

Warunków tych jest więcej, powyżej wymieniono jedynie kilka z nich.

W celu ułatwienia kontroli nad funkcjami i kryteriami, które muszą spełniać by mogły być stosowane w instrukcjach SQL, Oracle definiuje tak zwane *poziomy czystości*. Określają one, jakie struktury danych dana funkcja wczytuje i modyfikuje. Dostępne są cztery poziomy:

– WNDS – oznacza, że funkcja nie modyfikuje żadnych tabel bazy danych za pomocą instrukcji DML,

– RNDS – oznacza, że funkcja nie odczytuje żadnych tabel bazy danych (za pomocą instrukcji `SELECT`),

– WNPS – oznacza, że funkcja nie modyfikuje żadnych zmiennych pakietu,

– RNPS – oznacza, że funkcja nie odczytuje wartości żadnych zmiennych pakietu.

Według przedstawionych powyżej informacji funkcja wywoływana w zapytaniu musi mieć przynajmniej poziom czystości WNDS. W celu wymuszenia na kompilatorze sprawdzanie poziomów czystości funkcji należy zastosować dyrektywę:

```
PRAGMA RESTRICT_REFERENCES (nazwa_podprogramu_lub_pakietu),
  [WNDS][, WNPS][, RNDS][, RNPS] [TRUST] [DEFAULT]);
```

W przedstawionym wcześniej przykładzie pakietu została użyta dyrektywa `PRAGMA RESTRICT_REFERENCES` z opcją `DEFAULT`. Oznacza ona, że wszyskie występujące po niej podprogramy muszą mieć określony przez nią poziom czystości. Słowo kluczowe `TRUST` powoduje, że nie są sprawdzane ograniczenia narzucone w dyrektywie (przyjmuje się, że są one przestrzegane).

W celu poprawy wydajności można posłużyć się w deklaracji funkcji słowem kluczowym `DETERMINISTIC`. Stosuje się go dla funkcji zwracających zawsze te same dane na podstawie tych samych danych wejściowych. Umożliwia to przechowywanie wyników w pamięci podręcznej. Zastosowanie słowa `DETERMINISTIC` nie powoduje sprawdzenia, czy funkcja rzeczywiście jest deterministyczna. Ten aspekt leży w gestii programisty.

W przypadku funkcji zwracających zbiór wierszy, do których można kierować zapytania SQL tak, jakby to były tabele bazy danych (tak zwane funkcje tabelowe), w celu poprawienia wydajności można zastosować mechanizm potokowych funkcji tabelowych. Potokowa funkcja tabelowa to taka funkcja, która zwraca poszczególne wiersze zbioru wynikowego w miarę ich konstruowania, nie zaś wszystkie na raz. Potokową funkcję tabelową można zdefiniować wówczas, gdy możliwe jest pojedyncze określenie wierszy zwracanego zbioru. Do tworzenia funkcji potokowych służy słowo kluczowe `PIPELINED`, zaś poszczególne wiersze zwracane są za pomocą instrukcji `PIPE ROW`. Poniżej znajdują się przykłady funkcji tabelowych:

```
CREATE TYPE Ksiazka AS OBJECT (
  nr_kat NUMBER(6,0),
  tytul VARCHAR2(100)
);
/

CREATE TYPE KsiazkiTab AS TABLE OF Ksiazka;
/

CREATE OR REPLACE FUNCTION PobierzKsiazki
  (p_Kategoria ksiazki.id_kategorii%TYPE)
  RETURN KsiazkiTab AS
```

```
  z_Ksiazki KsiazkiTab := KsiazkiTab();
  CURSOR k_Ksiazki IS
    SELECT nr_katalogowy, tytul
    FROM ksiazki
    WHERE id_kategorii = p_Kategoria;
BEGIN
  FOR z_Rekord IN k_Ksiazki LOOP
    z_Ksiazki.EXTEND;
    z_Ksiazki(z_Ksiazki.LAST) :=
      Ksiazka(z_Rekord.nr_katalogowy, z_Rekord.tytul);
  END LOOP;
  RETURN z_Ksiazki;
END PobierzKsiazki;
/

-- SELECT * FROM TABLE (PobierzKsiazki(3));

CREATE OR REPLACE FUNCTION PobierzKsiazki_P
  (p_Kategoria ksiazki.id_kategorii%TYPE)
  RETURN KsiazkiTab PIPELINED AS

  z_Ksiazki KsiazkiTab := KsiazkiTab();
  CURSOR k_Ksiazki IS
    SELECT nr_katalogowy, tytul
    FROM ksiazki
    WHERE id_kategorii = p_Kategoria;
BEGIN
  FOR z_Rekord IN k_Ksiazki LOOP
    PIPE ROW
      (Ksiazka(z_Rekord.nr_katalogowy, z_Rekord.tytul));
  END LOOP;
  RETURN;
END PobierzKsiazki_P;
/
```

Pokazano tu jak zwracać wiersze z funkcji tabelowej potokowej za pomocą instrukcji `PIPE ROW`. W tym przypadku w instrukcji `RETURN` nie umieszcza się wyrażenia.

W języku PL/pgSQL nie istnieją osobne instrukcje do tworzenie funkcji i procedur. Wszystkie nazwane bloki kodu tworzy się tu za pomocą instrukcji `CREATE FUNCTION`:

```
CREATE OR REPLACE FUNCTION nazwa_funkcji
```

```
  (argumenty) RETURNS typ AS {$[ciąg_tekstowy]$ | '}
  DECLARE
    deklaracje;
  BEGIN
    instrukcje;
  EXCEPTION
    obsługa_błędów;
  END;
{$[ciąg_tekstowy]$ | '} LANGUAGE 'plpgsql';
```

Podobnie jak w języku PL/SQL, funkcja składa się z sekcji deklaracji, wykonawczej oraz obsługi błędów, z których jednynie sekcja wykonawcza jest obowiązkowa. Jeżeli używana jest sekcja deklaracji, wówczas, rozpoczynające ją słowo kluczowe `DECLARE` jest niezbędne. Jest to różnica w stosunku do PL/SQL, gdzie w podprogramach nie stosuje się tego słowa. Różnica występuje także w formie słowa określającego typ zwracanego wyniku. W PL/pgSQL słowo to ma postać `RETURNS`, podczas gdy w PL/SQL postać `RETURN`. Ponadto, w języku PL/pgSQL treść funkcji musi być ograniczona znakami specjalnymi (pojedynczym apostrofem lub podwójnym znakiem dolara). Zaś na końcu definicji funkcji należy podać w jakim języku została utworzona. Zarówno w PL/SQL jak i PL/pgSQL można natomiast użyć instrukcji tworzącej funkcję w postaci `CREATE OR REPLACE`. Wewnątrz funkcji napisanej w języku PL/pgSQL, tak jak wewnątrz podprogramu w PL/SQL, do zwracania wartości służy instrukcja `RETURN`. Funkcja w języku PL/pgSQL również może przyjmować parametry w trybach `IN`, `OUT` oraz `INOUT` (lub `IN OUT`). Nie można tu jednak zastosować modyfikatora `NOCOPY`. Tryb parametru może być podany zarówno przed jak i po nazwie parametru. Wynika to stąd, że nazwa parametru nie jest w PL/pgSQL obowiązkowa. Przykład funkcji przyjmującej parametry w różnym trybie podano poniżej:

```
--Przyklad napisany na PostgreSQL 8.2.5
CREATE FUNCTION funkcje
  (IN p_1 int, IN p_2 int, IN OUT p_3 int, OUT p_4 int) AS
$$
BEGIN
  p_3 := p_1 + p_2;
  p_4 := p_1 * p_2;
END;
$$
LANGUAGE 'plpgsql';
```

W przypadku zastosowania parametrów typu `OUT` stosowanie słowa kluczowego `RETURNS` nie jest konieczne. Jeśli mimo wszystko programista chce go użyć, wówczas typ zwracanej przez funkcję wartości musi odpowiadać paramatrom zadeklarowanym w trybach `OUT` lub `IN OUT`. W powyższym

przykładzie w instrukcji RETURNS należałoby zadeklarować typ rekordowy. W przypadku stosowania parametrów OUT nie jest również konieczne zwracanie wartości instrukcją RETURN.
W przeciwieństwie do PL/SQL, w języku PL/pgSQL nie można nadać parametrowi wartości domyślnej. Jeśli natomiast tryb parametru nie zostanie określony, wówczas w obu językach stosowany jest tryb domyślny IN.
Przykładowe wywołanie funkcji *funkcje* w języku PL/pgSQL wygląda następująco:

```
postgres=# select funkcje(4, 6, 1);
 funkcje
---------
 (10,24)
 (1 row)
```

Podobnie jak w PL/SQL, parametry w trybie IN są wewnątrz funkcji traktowane jako stałe i nie można przypisać im innej wartości. Również podobnie, definiując parametry formalne nie należy ograniczać typów (np. VARCHAR) co do rozmiaru. Warto więc i tym razem korzystać z operatora %TYPE. Do wersji 8.3 nie można natomiast zadeklarować parametru korzystając z %ROWYPE. Funkcja nie może również przyjmować parametru typu RECORD. Może natomiast zwracać wynik tego typu.
Funkcje w języku PL/pgSQL mogą przyjmować argumenty skalarne jak i tablice wszystkich typów danych wspieranych przez serwer PostgreSQL.
Jeżeli funkcja nie zwraca żadnej wartości można użyć składni RETURN VOID.

Tworząc funkcję w PL/pgSQL można pominać nazwy parametrów formalnych, wówczas wewnątrz funkcji należy się do nich odwoływać poprzez notację $1, $2, ... Można też utworzyć alias dla takiego parametru:

```
nazwa ALIAS FOR $n;
```

Wywołanie funkcji w języku PL/pgSQL zawsze musi zawierać nawiasy, nawet jeżeli funkcja nie przyjmuje wartości. Najprostszy sposób wywołania to zastosowanie instrukcji SELECT, jak zaprezentowano to w powyższym przykładzie. Ponadto funkcje zdefiniowane przez użytkownika mogą być wywoływane wszędzie tam, gdzie można stosować funkcje wbudowane.

Funkcję usuwa się instrukcją DROP. W instrukcji należy podać typy przyjmowanych przez funkcję parametrów:

```
DROP FUNCTION funkcje(int, int, int);
```

Wynika to z możliwości przeciążania funkcji w języku PL/pgSQL.

W przeciwieństwie do języka PL/SQL, w funkcji napisanej w języku PL/pgSQL nie zadeklarujemy podprogramu lokalnego.

W przypadku funkcji zwracających zbiory (SETOF pewien_typ_danych) można zastosować instrukcje RETURN NEXT wyrażenie oraz RETURN QUERY

`zapytanie`. Dodają one kolejne elementy do zbioru wynikowego funkcji. Instrukcja `RETURN NEXT` dodaje jeden element. Może być używana zarówno ze skalarnymi jak i złożonymi typami danych. `RETURN QUERY` dołącza natomiast cały zbiór wynikowy wykonanego zapytania.
Instrukcje `RETURN NEXT` oraz `RETURN QUERY` obecnie jedynie budują zbiór danych, nie powodując zwrócenia poszczególnych elementów do klienta. Stąd konieczność stosowania po nich instrukcji `RETURN` bez argumentów, wymuszającej zwrócenie całego zbudowanego zbioru. Być może kolejne wersje PostgreSQL pozwolą, by poszczególne elementy zbioru wynikowego funkcji były zwracane od razu, tak jak dzieje się to w potokowych funkcjach tabelowych w języku PL/SQL.
Oto przykład wykorzystania instrukcji `RETURN NEXT`:

```
-- W bazie istnieje tabela ksiazki, wypełniona danymi.
CREATE OR REPLACE FUNCTION PobierzKsiazki()
RETURNS SETOF ksiazki AS
$proc$
DECLARE
  z_rekord ksiazki%ROWTYPE;
BEGIN
  FOR z_rekord IN SELECT * FROM ksiazki LOOP
    RETURN NEXT z_rekord;
  END LOOP;
  RETURN;
END
$proc$
LANGUAGE 'plpgsql';
```

Wywołanie funkcji *PobierzKsiazki* wygląda następująco:

```
SELECT * FROM PobierzKsiazki();
```

Instrukcja tworząca funkcję w języku PL/pgSQL wymaga ograniczenia ciała funkcji znakami dolara lub apostrofu. Jeżeli zdecydujemy się na apostrof, wówczas użycie apostrofu w treści funkcji będzie wymagało podwojenia go:

```
CREATE OR REPLACE FUNCTION apostrofy
(tekst INOUT VARCHAR) AS'
BEGIN
  tekst := ''Poczatek -- '' || tekst || '' -- Koniec'';
END
'LANGUAGE 'plpgsql';
```

Ponadto, język PL/pgSQL, w przeciwieństwie do języka PL/SQL nie zezwala na użycie instrukcji zatwierdzania transakcji wewnątrz funkcji. Jako że funkcja w języku PL/pgSQL jest wykonywana w obrębie transakcji, zastosowanie instrukcji kończącej transakcję zakończyłoby działanie funkcji.

W języku PL/pgSQL nie istnieje dyrektywa PRAGMA, określająca poziom czystości. Można jednak udzielać wskazówek optymalizatorowi poprzez użycie na końcu definicji funkcji specjalnych słów kluczowych, między innymi: IMMUTABLE, STABLE oraz VOLATILE. IMMUTABLE oznacza, że funkcja nie modyfikuje stanu bazy danych i jest deterministyczna. STABLE należy użyć dla funkcji, które są deterministyczne w obrębie jednego skanowania tabeli i nie modyfikują tabel bazy danych. VOLATILE jest domyślnym zachowaniem funkcji i oznacza funkcję niedeterministyczną. Jak widać opcja IMMUTABLE zawiera w sobie jednocześnie Oracle'ową opcję DETERMINISTIC i odpowiednią dyrektywę PRAGMA.
Poniżej przedstawiono przykład wykorzystania opcji IMMUTABLE w języku PL/pgSQL:

```
CREATE OR REPLACE FUNCTION deterministic
(p_1 int, p_2 int) RETURNS int AS
$$
BEGIN
  RETURN p_1 + p_2;
END
$$
LANGUAGE 'plpgsql' IMMUTABLE;
```

Podobie jak w PL/SQL, liczba parametrów funkcji w języku PL/pgSQL jest nieograniczona.

W celu wywołania funkcji z pominięciem zwracanej przez nią wartości można posłużyć się instrukcją PERFORM:

```
CREATE OR REPLACE FUNCTION WstawKsiazke
  (p_nr_kat ksiazki.nr_katalogowy%TYPE,
   p_kategoria ksiazki.id_kategorii%TYPE,
   p_tytul ksiazki.tytul%TYPE,
   p_autor ksiazki.autor%TYPE,
   p_rok ksiazki.rok_wydania%TYPE,
   p_cena ksiazki.cena%TYPE)
RETURNS int AS
$$
BEGIN
  INSERT INTO ksiazki (nr_katalogowy,
   id_kategorii, tytul, autor, rok_wydania, cena)
  VALUES
  (p_nr_kat, p_kategoria, p_tytul, p_autor, p_rok, p_cena);

  RETURN 1;
END
$$
```

```
LANGUAGE 'plpgsql';

CREATE OR REPLACE FUNCTION WstawKsiazke2
  (p_nr_kat ksiazki.nr_katalogowy%TYPE,
   p_kategoria ksiazki.id_kategorii%TYPE,
   p_tytul ksiazki.tytul%TYPE,
   p_autor ksiazki.autor%TYPE,
   p_rok ksiazki.rok_wydania%TYPE,
   p_cena ksiazki.cena%TYPE)
RETURNS VOID AS
$$
BEGIN
  PERFORM WstawKsiazke
  (p_nr_kat, p_kategoria, p_tytul, p_autor, p_rok, p_cena);
END
$$
LANGUAGE 'plpgsql';
```

W języku PL/pgSQL w przypadku zmiany typu zwracanego przez funkcję można spodziewać się komunikatu o błędzie:

```
psql:/Marta/zrob_5.sql:163: ERROR:
  cannot change return type of existing function
HINT:  Use DROP FUNCTION first.
```

Dzieje się tak, gdy utworzyliśmy funkcję instrukcją `CREATE OR REPLACE`, a następnie chcemy ponownie skompilować tę samą funkcję, tylko ze zmienionym typem danych w klauzuli `RETURNS`. Błędu nie otrzymamy w każdym przypadku. Pojawi się, gdy na przykład funkcja pierwotnie zwracała typ `integer`, a chcemy by zwracała typ `RECORD` lub nie zwracała nic (`VOID`). W takiej sytuacji funkcję należy usunąć wprost instrukcją `DROP` i utworzyć ją na nowo.

Język SQL PL wyróżnia zarówno funkcje jak i procedury. Podstawowa składnia instrukcji tworzącej procedurę jest następująca:

```
CREATE PROCEDURE nazwa_procedury (lista_parametrów)
  [ SPECIFIC nazwa_specyficzna ]
  [ DYNAMIC RESULT SET 0 |
    DYNAMIC RESULT SETS liczba_integer ]
  [ MODIFIES SQL DATA | CONTAINS SQL | READS SQL DATA ]
  [ NOT DETERMINISTIC | DETERMINISTIC ]
  [ CALLED ON NULL INPUT ]
  [ LANGUAGE SQL ]
  [ EXTERNAL ACTION | NO EXTERNAL ACTION ]
  ciało_procedury
```

Parametry definiuje się w poniższy sposób:

```
[ IN | OUT | INOUT ] nazwa_parametru typ_danych
```

Jako typ parametru można podać każdy typ, który można stosować w instrukcji `CREATE TABLE`. Nie można zadeklarować parametrów typu `LONG VARCHAR`, `LONG VARGRAPHIC` oraz `REFERENCE`. Niedozwolone jest również zastosowananie typów strukturalnych zdefiniowanych przez użytkownika.

Składnia tworzenia funkcji w języku SQL PL (w wersji uproszczonej) jest następująca:

```
CREATE FUNCTION nazwa_funkcji (lista parametrów)
RETURNS zwracany_typ_danych
[ SPECIFIC nazwa_specyficzna ]
[ LANGUAGE SQL ]
[ NOT DETERMINISTIC | DETERMINISTIC ]
[ EXTERNAL ACTION | NO EXTERNAL ACTION ]
[ MODIFIES SQL DATA | CONTAINS SQL | READS SQL DATA ]
[ CALLED ON NULL INPUT ]
ciało_funkcji
```

W przypadku funkcji deklaracja parametru nie zawiera trybu parametru:

```
nazwa_parametru typ_danych
```

Funkcje mogą zarówno zwracać jak i przyjmować wbudowane typy danych, typy strukturalne, jak i referencje `REF (nazwa_typu)`. W odróżnieniu do języków PL/SQL i PL/pgSQL, instrukcje tworzące podprogramy w języku SQL PL nie zawierają słowa kluczowego `REPLACE`. Przed utworzeniem nowej wersji podprogramu wymagane jest jawne usunięcie starej wersji instrukcją:

```
DROP PROCEDURE nazwa_procedury.
```

Konieczne jest używanie słowa `DECLARE` podczas deklarowania elementów. Słowo kluczowe `RETURNS` w definicji funkcji ma natomiast formę taką jak w języku PL/pgSQL.

Zarówno procedury jak i funkcje w języku SQL PL mogą być przeciążane. Można utworzyć procedury o tej samej nazwie, ale z różną liczbą parametrów. Nie jest dozwolone utworzenie procedur o tej samej nazwie i z tą samą liczbą parametrów, nawet jeśli są różnych typów. W przypadku funkcji, można utworzyć różne funkcje o tej samej nazwie i tej samej liczbie parametrów, tylko o różnym typie.

Wewnątrz funkcji, tak jak w innych językach, wartości zwracane są za pomocą instrukcji `RETURN`.

Tryby parametrów mają zastosowanie jedynie w odniesieniu do procedur. Ponownie domyślnym jest tryb `IN`. W przeciwieństwie do PL/SQL i PL/pgSQL, w języku SQL PL można w ciele podprogramu przypisywać wartości do parametru w trybie `IN`:

```
CREATE PROCEDURE param
  (IN p_1 INT, INOUT p_2 INT, OUT p_3 INT, OUT p_4 INT)
  SPECIFIC param
  LANGUAGE SQL
p1: BEGIN
  SET p_1 = 10;
  SET p_2 = 20;
  SET p_3 = p_1 * p_2;
  SET p_4 = p_1;
END p1
@
```

Kolejną różnicą między językiem SQL PL a PL/SQL i PL/pgSQL jest to, że parametry formalne typów takich jak `VARCHAR` czy `DECIMAL` muszą być ograniczone co do długości, rozmiaru i precyzji. Brak tego ograniczenia spowoduje błąd.
SQL PL nie oferuje też modyfikatora `NOCOPY`, takiego jaki jest dostępny w deklaracji parametru formalnego w języku PL/SQL.

Liczba deklarowanych parametrów, w przypadku procedur, podobnie jak w Oracle i PostgreSQL, jest dowolna. W przypadku funkcji liczba parametrów jest ograniczona do dziewięćdziesięciu.
Jeżeli procedura w języku SQL PL nie przyjmuje żadnych parametrów, wówczas w jej deklaracji i wywołaniu można pominąć nawiasy. W przypadku funkcji nawiasów nie można pominąć ani w deklaracji, ani w wywołaniu.

W celu uruchomienia procedury należy posłużyć się instrukcję `CALL`:

```
CALL nazwa_procedury (lista_argumentów);
```

Instrukcja `CALL` służy również do wywoływania procedury z wnętrza innej procedury (procedury zagnieżdżone).
W samodzielnym wywołaniu procedury (np. z poziomu narzędzia DB2), w miejscu parametrów w trybie `OUT` należy podać znaki '`?`', na przykład dla procedury *param* wywołanie będzie wyglądało następująco:

```
db2 => CALL param (7, 2, ?, ?)

  Wartości parametrów wyjściowych
  -------------------------
  Nazwa parametru: P_2
  Wartość parametru: 20

  Nazwa parametru: P_3
  Wartość parametru: 200

  Nazwa parametru: P_4
```

```
  Wartość parametru: 10

  Status powrotu= 0
db2 =>
```

W przypadku wywołania procedury zawierającej parametry wyjściowe wewnątrz innej procedury, należy wartości wspomnianych parametrów wyjściowych podpisać pod przygotowane w tym celu zmienne.
Oto przykład wywołania procedury *param* wewnątrz drugiej procedury – *wywolaj_param*. W tym przypadku wartości parametrów wyjściowych z procedury *param* zostały bezpośrednio przypisane do parametrów zwracanych przez zewnętrzną procedurę *wywolaj_param*:

```
CREATE PROCEDURE wywolaj_param
  (IN p_1 INT, INOUT p_2 INT, OUT p_5 INT, OUT p_6 INT)
  SPECIFIC wywolaj_param
  LANGUAGE SQL
wp: BEGIN
  CALL param (p_1, p_2, p_5, p_6);
END wp
@
```

Deklarując podprogram w języku SQL PL można obok nazwy właściwej nadać mu nazwę specyficzną (klauzula `SPECIFIC`). Jest to zalecane w przypadku programów przeciążonych. Pozwala w prosty sposób rozróżnić podprogramy o tej samej nazwie właściwej lecz o różnych parametrach.
Nadawanie podprogramom nazw specyficznych jest opcjonalne. Jeżeli nazwa specyficzna nie zostanie jawnie nadana, system sam tę nazwę wygeneruje. Będzie miała ona wówczas postać `SQLyymmddhhmmsshhn`.
Podprogram z nadaną nazwą specyficzną można usunąć na dwa sposoby: poprzez nazwę właściwą lub poprzez nazwę specyficzną.
Poniższy przykład przedstawia dwie funkcje o tej samej nazwie właściwej i różnych nazwach specyficznych, oraz sposoby ich wywołania i usunięcia:

```
CREATE FUNCTION MAIN.funkcja ()
  RETURNS DOUBLE
  NO EXTERNAL ACTION
  SPECIFIC fun1
BEGIN ATOMIC
  RETURN 1;
END
@

CREATE FUNCTION MAIN.funkcja (p_1 INT)
  RETURNS DOUBLE
```

```
  NO EXTERNAL ACTION
  SPECIFIC fun2
BEGIN ATOMIC
  RETURN 2*p_1;
END
@

SELECT MAIN.funkcja() FROM SYSIBM.SYSDUMMY1
SELECT MAIN.funkcja(3) FROM SYSIBM.SYSDUMMY1

DROP FUNCTION MAIN.funkcja()
DROP FUNCTION MAIN.funkcja(INT)
DROP SPECIFIC FUNCTION MAIN.fun1
DROP SPECIFIC FUNCTION MAIN.fun2
```

W przypadku usuwania poprzez nazwę właściwą należy podać typy parametrów formalnych przyjmowanych przez przeciążony program. W przypadku usuwania poprzez nazwę specyficzną należy zastosować słowo kluczowe `SPECIFIC` bez podawania typów parametrów. Podanie typów parametrów w tym przypadku wywoła błąd.
W przykładzie pokazano również tworzenie i usuwanie programów z danego schematu (tutaj schemat `MAIN`). Jeżeli nazwa schematu nie zostanie podana przed nazwą programu, wówczas jako schemat zostanie użyty schemat bieżący, określony przez zmienną `CURRENT SCHEMA`.
Jeżeli podprogram nie jest przeciążony, wówczas podczas usuwania go poprzez podanie nazwy właściwej nie jest konieczne używanie nawiasów i podawanie typów parametrów.

Jeżeli procedura ma zwracać dane w postaci zbiorów, wówczas w deklaracji procedury należy określić liczbę zwracanych zbiorów. Służy do tego klauzula `DYNAMIC RESULT SETS`. W przypadku, gdy procedura nie zwraca zbiorów danych, klauzulę tę można pominąć (zaleca się jednak jej używanie z zerową liczbą zbiorów).
Zarówno dla procedury jak i dla funkcji w języku SQL PL, można w deklaracji zawrzeć klauzulę określającą zachowanie podprogramu. Słowa kluczowe `CONTAINS SQL` oznaczają, że wewnątrz podprogramu mogą być wykonywane jedynie takie instrukcje, które nie odczytują i nie modyfikują danych w bazie danych (dopuszczalne są np. `GLOBAL TEMPORARY TABLE`, `PREPARE`). Opcja `READS SQL DATA` określa, że program może odczytywać dane, zaś opcja `MODIFIES SQL DATA` – że może je również modyfikować. Podobnie jak w PL/SQL, można zastosować słowa `NOT DETERMINISTIC` lub `DETERMINISTIC` jako podpowiedzi dla optymalizatora. Opcja `NOT DETERMINISTIC` jest zachowaniem domyślnym.
Elementy deklaracji podprogramu w języku SQL PL, określające zachowanie programu w stosunku do danych zgromadzonych w bazie (klauzu-

la [MODIFIES SQL DATA | CONTAINS SQL | READS SQL DATA]) oraz określające, czy podprogaram zachowuje się deterministycznie (klauzula [NOT DETERMINISTIC | DETERMINISTIC]) kojarzą się z poziomami czystości oraz klauzulą DETERMINISTIC w Oracle oraz opcjami IMMUTABLE, STABLE oraz VOLATILE w PostgreSQL.

Podobnie jak w języku PL/pgSQL, w SQL PL istnieje opcjonalna klauzula LANGUAGE. Dla podprogramów w proceduralnym języku SQL można ją pominąć. Język proceduralny SQL jest tu językiem domyślnym.

W językach PL/SQL i PL/pgSQL ciało podprogramu dzieli się na trzy części: sekcję deklaracji, sekcję wykonawczą i sekcję obsługi wyjątków. W języku SQL PL struktura bloku nie wyszczególnia odrębnych części dla deklaracji i obsługi wyjątków. Cała treść podprogramu zawiera się między słowami BEGIN - END. Wymagana jest jednak pewna kolejność elementów w bloku. Wszystkie deklaracje muszą znajdować się tuż za słowem BEGIN, w określonym porządku:

```
deklaracje zmiennych;
deklaracje wyjątków (conditions);
deklaracja wartości zwracanej (return code dla procedur);

deklaracje kursorów;
deklaracje uchwytów błędów (handlers);
```

Procedura w języku SQL PL może zwracać, za pomocą instrukcji RETURN, pojedynczą wartość typu integer. Teoretycznie może to być wartość o dowolnym znaczeniu, przyjmuje się jednak, że instrukcją RETURN zwraca się wartość określającą status powrotu procedury.

Poniżej przedstawiono przykład procedury w języku SQL PL:

```
-- <ScriptOptions statementTerminator="@" />

-- Procedura zwraca (przez RETURN STATUS):
--  1 jesli dodawany jest nowy rekord do tabeli HotPixel
--  0 jesli rekord nie jest dodawany;
-- -1 jesli wykonanie zakonczy sie bledem SQLEXCEPTION

CREATE PROCEDURE MAIN.AddHotPixel
    (IN p_x DOUBLE, IN p_y DOUBLE, IN p_cam INTEGER)
  SPECIFIC AddHotPixel
  LANGUAGE SQL
ahp: BEGIN
  DECLARE v_return_code INTEGER DEFAULT 0;
  DECLARE v_hv_count INTEGER;

  DECLARE EXIT HANDLER FOR SQLEXCEPTION
```

```
      SET v_return_code = -1;

    atom1: BEGIN ATOMIC
      SELECT COUNT(*) INTO v_hv_count FROM MAIN.HotPixel
        WHERE ABS(hp_x-p_x)<=1
        AND ABS(hp_y-p_y)<=1 AND hp_cam=p_cam;

  IF ( v_hv_count < 1 )THEN
    BEGIN
      INSERT INTO MAIN.HotPixel (hp_x,hp_y,hp_cam)
        VALUES (p_x,p_y,p_cam);
      SET v_return_code = 1;
    END;
  ELSE
    SET v_return_code = 0;
  END IF;
    END atom1;

    RETURN v_return_code;

  END ahp
  @
```

Wywołanie procedury *MAIN.AddHotPixel* może mieć następującą postać:

```
db2 => call MAIN.AddHotPixel(17.3, 19.4, 3)
  Status powrotu= 0
db2 =>
```

Status powrotu (czyli `RETURN STATUS`) to wartość zwracana przez instrukcję `RETURN` z procedury.

W języku SQL PL nie można zadeklarować podprogramu lokalnego, tak jak jest to możliwe w SQL/PL. Można jednak wywoływać (tak jak w innych językach) podprogramy we wnętrzu innych podprogramów, przy czym zabroniona jest rekurencja w przypadku funkcji. Po wywołaniu procedury zagnieżdżonej, można odczytać zwracany przez nią status za pomocą omówionej już wcześniej instrukcji:

```
GET DIAGNOSTICS nazwa_zmiennej = {ROW_COUNT | RETURN_STATUS};
```

Do odczytu statusu należy użyć `GET DIAGNOSTICS` wraz z `RETURN_STATUS`:

```
-- Procedura zwarca (przez RETURN STATUS):
-- liczbe zaktualizowanych rekordow,
--     jesli wykonanie jest bez bledow
-- -1 jesli wykonanie zakonczy sie bledem.
```

```
-- DROP PROCEDURE MAIN.FillObsStat @

CREATE PROCEDURE MAIN.FillObsStat ()
  SPECIFIC FillObsStat
  DYNAMIC RESULT SETS 1
  LANGUAGE SQL
fos: BEGIN
  DECLARE v_update_ret INTEGER;
  DECLARE v_ret INTEGER;

  DECLARE EXIT HANDLER FOR SQLEXCEPTION
    SET v_ret = -1;

  atom1: BEGIN ATOMIC
    SET v_ret = 0;

    FOR v_row AS SELECT sobject, idaynight, fra, fdec
    FROM MAIN.Frame ORDER BY idaynight DO
      CALL MAIN.UpdateFrameStat (v_row.sobject,
        v_row.idaynight, v_row.fra, v_row.fdec);
      GET DIAGNOSTICS v_update_ret = RETURN_STATUS;

      IF (v_update_ret = -1) THEN
        BEGIN
          SET v_ret = -1;
          SIGNAL SQLSTATE '80002'
            SET MESSAGE_TEXT = 'Blad przetwarzania SQL:
             FIllObsStat: UpdateFramedStat: SQLEXCEPTION.';
        END;
      ELSE
        SET v_ret = v_ret + v_update_ret;
      END IF;
    END FOR;
  END atom1;

  RETURN v_ret;
END fos
@
```

W sytuacji, gdy procedura zagnieżdżona zwraca zbiór danych, można ten zbiór danych przekazać albo bezpośrednio do aplikacji wywołującej albo do procedury, która wywołuje procedurę zagnieżdżoną. W tym celu należy zastosować klauzulę `WITH RETURN` w deklaracji zwracanego kursora:

```
DECLARE nazwa_kursora CURSOR
  [ WITH HOLD ]
  WITH RETURN { TO CALLER | TO CLIENT }
  FOR zapytanie_kursora;
```

Opcja `TO CALLER` powoduje zwrócenie wyniku do programu bezpośrednio wywołującego procedurę zagnieżdżoną (np. do innej procedury). Jeśli zbiór wynikowy procedury zagnieżdżonej nie musi być dostępny dla pośrednich, wywołujących ją procedur SQL i ma być przekazany w niezmienionym stanie do aplikacji zewnętrznej, wówczas można zastosować opcję `TO CLIENT`. Dane zwrócone przez kursor zostaną przekazane do aplikacji zewnętrznej, zaś procedury pośrednie nie będą miały do nich dostępu.
Poniższy przykład przedstawia dwie procedury: jedną zwracającą wyniki do klienta i jedną zwracającą do programu wywołującego:

```
CREATE PROCEDURE to_client ()
  SPECIFIC to_client
  DYNAMIC RESULT SETS 1
  LANGUAGE SQL
tc: BEGIN
  DECLARE v_kur CURSOR WITH RETURN TO CLIENT
    FOR SELECT * FROM MAIN.EVENT_TYPE;
  OPEN v_kur;
END tc
@


CREATE PROCEDURE to_caller ()
  SPECIFIC to_caller
  DYNAMIC RESULT SETS 1
  LANGUAGE SQL
tc: BEGIN
  DECLARE v_kur CURSOR WITH RETURN TO CALLER
    FOR SELECT * FROM MAIN.EVENT_TYPE;
  OPEN v_kur;
END tc
@
```

Procedury te można następnie wywołać w innych:

```
CREATE PROCEDURE zewnetrzna_1 ()
  SPECIFIC zewnetrzna_1
  DYNAMIC RESULT SETS 1
  LANGUAGE SQL
z:BEGIN
```

```
  CALL to_client();
END z
@


CREATE PROCEDURE zewnetrzna_2 ()
  SPECIFIC zewnetrzna_2
  LANGUAGE SQL
z:BEGIN
  CALL to_caller();
END z
@
```

Uruchamiając procedurę *zewnetrzna_1* otrzymamy taki sam zbiór wyników, jaki zwraca procedura *to_client*. W wyniku wywołania procedury *zewnetrzna_2* nie otrzymamy tego zbioru – został on zwrócony jedynie do wnętrza procedury *zewnetrzna_2*.

Z poziomu procedury wywołującej procedurę zagnieżdżoną dostęp do zwracanego przez procedurę zagnieżdżoną zbioru można uzyskać za pomocą instrukcji `ASSOCIATE LOCATOR` oraz `ALLOCATE CURSOR`:

```
CREATE PROCEDURE zewnetrzna_3 ()
  SPECIFIC zewnetrzna_3
  DYNAMIC RESULT SETS 1
  LANGUAGE SQL
z:BEGIN
  DECLARE v_licznik INTEGER;
  DECLARE v_event_type_val INTEGER;
  DECLARE v_event_type_desc VARCHAR(128);
  DECLARE v_zbior RESULT_SET_LOCATOR VARYING;
  DECLARE SQLSTATE CHAR(5) DEFAULT '00000';

  CALL to_caller();
  ASSOCIATE RESULT SET LOCATOR (v_zbior)
    WITH PROCEDURE to_caller;
  ALLOCATE v_zbiorKursor CURSOR FOR RESULT SET v_zbior;

  SET v_licznik = 0;
  WHILE (SQLSTATE = '00000') DO
    SET v_licznik = v_licznik + 1;
    FETCH FROM v_zbiorKursor
      INTO v_event_type_val, v_event_type_desc;
  END WHILE;

  ...
```

```
  RETURN v_licznik;
END z
@
```

Pokazane dotychczas procedury zwracały pojedynczy zbiór wynikowy. Zwracanie wielu zbiorów wygląda tak samo. Należy tylko podać odpowiednią liczbę w klauzuli `DYNAMIC RESULT SETS`, zaś wewnątrz procedury powtórzyć stosowne operacje dla każdego zbioru, tak jak to zostało pokazane dla jednego zbioru. Należy też pamiętać o odebraniu zbiorów z procedury zagnieżdżonej w odpowiedniej kolejności (zbiory są identyfikowane przez kolejność w jakiej zostały zwrócone).

Procedura, w odróżnieniu od funkcji, może wywoływać samą siebie (rekursja). W tym celu należy zastosować dynamiczny SQL. DB2 obsługuje maksymalnie 16 poziomów zagnieżdżenia procedur oraz 16 poziomów rekurencji procedur. Nie ma ograniczenia poziomów zagnieżdżania funkcji.

Z tworzeniem funkcji w języku SQL PL wiąże się kilka ograniczeń. Wewnątrz funkcji nie można stosować wszystkich instrukcji, które są dostępne dla procedur. Nie można deklarować uchwytów błędów, używać instrukcji `SELECT INTO` (zamiast niej należy posłużyć się instrukcją `SET`) czy stosować dynamicznego SQL. Instrukcje dostępne dla funkcji to: `DECLARE <zmienna>`, `FOR`, `GET DIAGNOSTOCS`, `IF`, `ITERATE`, `LEAVE`, `SIGNAL`, `WHILE` oraz `SET`. Ciało funkcji powinno się znajdować w bloku `ATOMIC`. Użycie bloku `NOT ATOMIC` spowoduje błąd. Funkcje tworzone przez użytkownika mogą zwracać wartość skalarną, wiersz lub tabelę.
Poniżej znajduje się przykład funkcji zwracającej wynik w postaci tabeli:

```
CREATE FUNCTION zwroc_tabele (p_event_type_val INTEGER)
  RETURNS TABLE
    (lp INTEGER,
     event_type_val INTEGER,
     event_type_desc VARCHAR(128))
  SPECIFIC zwroc_tabele
  RETURN
    SELECT ROW_NUMBER() OVER(),
      e.event_type_val, e.event_type_desc
      FROM MAIN.event_type e
      WHERE e.event_type_val >= p_event_type_val
@
```

Funkcję tabelową *zwroc_tabele* należy wywołać w taki sposób:

```
SELECT * FROM TABLE (zwroc_tabele (2))
```

Powyższy zapis kojarzy się z wywoływaniem funkcj tabelowych w języku PL/SQL.

Dla funkcji zwracających tabelę można zastosować w deklaracji funkcji opcję MODIFIES SQL DATA (opcja ta jest dostępna jedynie dla funkcji tabelowych). Przedstawiona poniżej funcja *test_sql_2* zwracaca wynik w postaci tabeli. Została zadeklarowana z użyciem opcji MODIFIES SQL DATA, zaś wewnątrz niej wykonano instrukcję wstawiającą dane do tabeli:

```
CREATE FUNCTION test_sql_2 ()
  RETURNS TABLE (kol1 INTEGER, kol2 INTEGER)
  MODIFIES SQL DATA
  SPECIFIC test_sql_2
  BEGIN ATOMIC
    DECLARE v_zap VARCHAR(100);

    SET v_zap = 'SELECT event_type_desc
      FROM MAIN.event_type WHERE event_type_val = 1';

    INSERT INTO MAIN.event_type
      VALUES (999, 'ala ma kota');

    RETURN
      SELECT 1, 5 FROM sysibm.SYSDUMMY1;
  END
@
```

Jeżeli zmienimy deklarację funnkcji *test_sql_2* tak, by zamiast tabeli zwracała wynik typu INTEGER, wówczas nie możemy posłużyć się opcją MODIFIES SQL DATA (jeśli jej użyjemy, kompilator zasygnalizuje wystąpienie błędu). Instrukcja INSERT nie może być teraz wykonana:

```
CREATE FUNCTION test_sql_2 ()
  RETURNS INTEGER
  LANGUAGE SQL
  READS SQL DATA
  SPECIFIC test_sql_2
  BEGIN ATOMIC
    DECLARE v_zap VARCHAR(100);

    SET v_zap = 'SELECT event_type_desc
      FROM MAIN.event_type WHERE event_type_val = 1';

    -- ta instrukcja potrzebuje opcji MODIFIES SQL DATA
    --INSERT INTO MAIN.event_type
    --  VALUES (999, 'ala ma kota');

    RETURN 5;
```

```
  END
@
```

W DB2 spotkamy się również z pojęciem *pakietu*. Jest to obiekt tworzony w fazie prekompilacji programu. Przechowuje skompilowane instrukcje SQL i ścieżki dostępu wybrane przez optymalizator. Jak widać, struktura ta, poza nazwą, nie jest w niczym podobna do pakietu w języku PL/SQL.

Język Transact-SQL, tak jak PL/SQL i SQL PL, pozwala na tworzenie obu typów podprogramów: procedur i funkcji. Procedury tworzy się za pomocą instrukcji `CREATE PROCRDURE`:

```
CREATE { PROC | PROCEDURE }
  [nazwa_schematu.] nazwa_procedury [ ; numer ]
  lista_parametrów
  [ WITH [ ENCRYPTION | RECOMPILE | klauzula_EXECUTE_AS ] ]
  [ FOR REPLICATION ]
AS
  ciało procedury
[;]
```

Ciało procedury może być zbiorem instrukcji (niezamkniętym w znaczniki `BEGIN - END`), blokiem `BEGIN - END`, bądź też odniesieniem do metody napisanej w środowisku CLR (Common Language Runtime – środowisko uruchomieniowe, składnik platformy .NET firmy Microsoft). Dalsza charakterystyka procedur w języku T-SQL będzie dotyczyła podprogramów napisanych w proceduralnym SQL, nie w CLR.
Maksymalny rozmiar procedury to 128 MB.

Parametry procedury definiuje się następująco:

```
@nazwa_parametru [ nazwa_schematu_typu. ] typ_danych
  [ VARYING ] [ = wartość_domyślna ] [ OUT | OUTPUT ]
```

Maksymalna akceptowana liczba parametrów to 2100. Nazwa parametru, tak jak nazwa zmiennej w języku T-SQL, musi rozpoczynać się znakiem `@`. Parametrom można przypisywać wartości domyślne.
Dla parametru można zastosować dowolny typ danych, za wyjątkiem tablic. Pewne ograniczenie dotyczy kursorów – ten typ może być stosowany jedynie jako parametr w trybie `OUTPUT`. Musi być wówczas podana opcja `VARYING`, określająca zbiór wynikowy dla parametru `OUTPUT` (opcja ta jest używana jednynie z kursorami). Można zadeklarować wiele parametrów typu kursor dla jednej procedury.
W języku T-SQL nie istnieje jawnie tryb `IN` parametru. Parametry zadeklarowane bez modyfikatora `OUTPUT` (synonim `OUT`) są traktowane jako parametry wejściowe. W przeciwieństwie do PL/SQL i PL/pgSQL, a podobnie

jak w SQL PL, parametrom wejściowym można przypisywać wartości w ciele procedury. Język T-SQL nie udostępnia trybu `IN OUT`. Można natomiast w tym celu wykorzystać parametr `OUT`. W ten sposób wykorzystanie parametrów wyjściowych staje się w T-SQL nieco nieintuicyjne. Oto przykład manipulacji parametrami w różnych trybach. Na początek procedura:

```
CREATE PROCEDURE przyklad  @imie VARCHAR(50),
                           @nazwisko VARCHAR(50),
                           @nazwa1 VARCHAR(100) OUT,
                           @nazwa2 text OUTPUT,
                           @nazwa3 VARCHAR(100) OUTPUT
AS
BEGIN
  -- można przypisywć wartości do parametrów wejściowych
  SET @imie = 'Ala';
  SET @nazwa2 = @nazwa1;
  SET @nazwa1 = @imie + ' ' + @nazwisko;
  SET @nazwa3 = @nazwa1;
END;
```

Następnie różne warianty wywołania procedury *przyklad*:

```
BEGIN
  DECLARE @in_imie VARCHAR(50);
  DECLARE @in_nazwisko VARCHAR(50);
  DECLARE @out_nazwa1 VARCHAR(100);
  DECLARE @out_nazwa2 VARCHAR(100);
  DECLARE @out_nazwa3 VARCHAR(100);

  SET @in_imie = 'Agnieszka';
  SET @in_nazwisko = 'Kowalska';
  SET @out_nazwa1 = 'Kwiat';

  exec przyklad @in_imie, @in_nazwisko, @out_nazwa1,
    @out_nazwa2 OUTPUT, @out_nazwa3 OUTPUT;
  PRINT ('@in_imie: ' + @in_imie);
  PRINT ('@in_nazwisko: ' + @in_nazwisko);
  PRINT ('@out_nazwa1: ' + @out_nazwa1);
  PRINT ('@out_nazwa2: ' + @out_nazwa2);
  PRINT ('@out_nazwa3: ' + @out_nazwa3);
  PRINT CHAR(10);
  exec przyklad @in_imie, @in_nazwisko, @out_nazwa1 OUTPUT,
    @out_nazwa2 OUTPUT, @out_nazwa3 OUTPUT;
  PRINT ('@in_imie: ' + @in_imie);
  PRINT ('@in_nazwisko: ' + @in_nazwisko);
```

```
  PRINT ('@out_nazwa1: ' + @out_nazwa1);
  PRINT ('@out_nazwa2: ' + @out_nazwa2);
  PRINT ('@out_nazwa3: ' + @out_nazwa3);
  PRINT CHAR(10);
  exec przyklad @in_imie OUTPUT, @in_nazwisko,
    @out_nazwa1 OUTPUT, @out_nazwa2 OUTPUT,
    @out_nazwa3 OUTPUT;
  PRINT ('@in_imie: ' + @in_imie);
  PRINT ('@in_nazwisko: ' + @in_nazwisko);
  PRINT ('@out_nazwa1: ' + @out_nazwa1);
  PRINT ('@out_nazwa2: ' + @out_nazwa2);
  PRINT ('@out_nazwa3: ' + @out_nazwa3);
END
```

Oto otrzymane wyniki:

```
@in_imie: Agnieszka
@in_nazwisko: Kowalska
@out_nazwa1: Kwiat
@out_nazwa2: Kwiat
@out_nazwa3: Ala Kowalska

@in_imie: Agnieszka
@in_nazwisko: Kowalska
@out_nazwa1: Ala Kowalska
@out_nazwa2: Kwiat
@out_nazwa3: Ala Kowalska

Msg 8162, Level 16, State 2, Procedure przyklad, Line 0
The formal parameter "@imie" was not declared as
  an OUTPUT parameter, but the actual parameter
  passed in requested output.
@in_imie: Agnieszka
@in_nazwisko: Kowalska
@out_nazwa1: Ala Kowalska
@out_nazwa2: Kwiat
@out_nazwa3: Ala Kowalska
```

Jedynie próba wykorzystania parametru bez trybu jako parametru w trybie `OUTPUT` zakończyła się błędem. Wykorzystanie wartości wejściowych w miejsce parametru `OUT` powiodło się.

Język Transact-SQL nie udostępnia opcji `REPLACE`. Procedurę można jednak zmienić bez uprzedniego usuwania jej. Służy do tego instrukcja

```
ALTER PROCEDURE nazwa_procedury
... [;]
```

W wersji SQL Serwer 2005 dostępna jest opcja `number` dla deklarowanej procedury. Pozwala ona grupować procedury o tej samej nazwie. T-SQL nie pozwala na przeciążanie procedur w sposób znany z innych języków. Opcja `number` pozwala obejść to ograniczenie i zadeklarować procedury o jednakowej nazwie i różnych ciałach. Grupa takich procedur może być następnie usunięta jednym poleceniem:

```
DROP PROCEDURE nazwa_procedury [;]
```

Poniższy przykład przedstawia dwa podprogramy o tej samej nazwie z nadanymi numerami:

```
DROP PROCEDURE lokalny
GO

CREATE PROCEDURE lokalny;1 (@liczba INT)
AS
BEGIN
  DECLARE @z_liczba INT;
  SET @z_liczba = @liczba + 10;
  SELECT @z_liczba;
END
GO

CREATE PROCEDURE lokalny;2 (@liczba1 INT, @liczba2 INT)
AS
BEGIN
  DECLARE @z_liczba INT;
  SET @z_liczba = @liczba1 + @liczba2;
  SELECT @z_liczba;
END
GO

exec lokalny;1 6
exec lokalny;2 6, 5
```

Dla procedur numerowanych nie można stosować jako parametrów typu xml oraz typów zdefiniowanych przez użytkownika.
Opcja `number` ma zostać usunięta w kolejnej wersji SQL Server.

Procedury można tworzyć z trzema opcjami: `ENCRYPTION` powoduje zaszyfrowanie procedury, `RECOMPILE` wymusza kompilację procedury przy każdym jej uruchomieniu, zaś opcja `EXECUTE AS` definiuje kontekst wykonania dla procedury składowanej.
Kontekst wykonania można zdefiniować zarówno dla procedury jak i dla funkcji:

```
{ EXEC | EXECUTE } AS
  { CALLER | SELF | OWNER | 'nazwa_użytkownika' }
```

`EXECUTE AS CALLER` wykorzystuje do wykonania procedury dane uwierzytelniające użytkownika wywołującego. Jest to zachowanie domyślne. Opcja `EXECUTE AS SELF` wykorzystuje dane uwierzytelniające użytkownika, który jako ostatni modyfikował procedurę, zaś opcja `EXECUTE AS OWNER` wykorzystuje dane użytkownika, który jest właścicielem procedury składowanej. `EXECUTE AS 'nazwa_użytkownika'` wykorzystuje natomiast dane uwierzytelniające wskazanego użytkownika.
Oto przykład procedury wykorzystującej kontekst wykonania, zwracającej kursor:

```
CREATE PROCEDURE dbo.kontekst
  @kursor CURSOR VARYING OUTPUT
  WITH EXECUTE AS CALLER
AS
  SET @kursor = CURSOR FOR
    SELECT * FROM dbo.ksiazki
  OPEN @kursor
```

Oraz sposób wywołania powyższego podprogramu jako procedury zagnieżdżonej:

```
ALTER PROCEDURE zewnetrzna
AS
BEGIN
  DECLARE @out_kursor CURSOR;
  DECLARE @z_nr INT;
  DECLARE @z_tytul VARCHAR(50);
  DECLARE @z_autor VARCHAR(50);

  BEGIN TRY
    EXEC dbo.kontekst @out_kursor OUTPUT;
    DECLARE @tabKsiazki TABLE
    (nr int, tytul VARCHAR(50), autor VARCHAR(50));

    FETCH NEXT FROM @out_kursor
      INTO @z_nr, @z_tytul, @z_autor;
    WHILE (@@FETCH_STATUS = 0)
    BEGIN
      INSERT INTO @tabKsiazki (nr, tytul, autor)
        VALUES (@z_nr, @z_tytul, @z_autor);
      FETCH NEXT FROM @out_kursor
        INTO @z_nr, @z_tytul, @z_autor;
```

```
    END;
    CLOSE @out_kursor;
    DEALLOCATE @out_kursor;
  END TRY

  BEGIN CATCH
    SELECT
      ERROR_NUMBER() AS Numer_bledu,
      ERROR_MESSAGE() AS Komunikat;
  END CATCH

  SELECT * FROM @tabKsiazki;
END
```

Jak pokazano wcześniej, procedury wywowołuje się instrukcją `EXEC` (lub `EXECUTE`). Instrukcję tę stosuje się również w przypaku procedur zagnieżdżonych. W przykładzie zaprezentowano także użycie konstrukcji `TRY - CATCH` do obsługi błędów.

Procedurę można również zadeklarować z opcją `FOR REPLICATION`. Tak zadeklarowane procedury są uruchamiane jedynie podczas replikacji i nie mogą przyjmować żadnych parametrów. Z opcją `FOR REPLICATION` nie można używać opcji `RECOMPILE`.

Procedury w SQL Serwer 2005 mogą być deklarowane również w wersji tymczasowej lokalnej i globalnej. Nazwa procedur tymczasowych lokalnych musi rozpoczynać się znakiem `#`, zaś tymczasowych globalnych od znaków `##`. Procedury tymczasowe są zawsze tworzone w bazie `tempdb`. Lokalne procedury tymczasowe są widoczne tylko dla połączenia, w którym zostały utworzone i są usuwane wraz z zakończeniem sesji. Globalne procedury tymczasowe są widoczne dla wszystkich połączeń i są usuwane wraz z zakończeniem ostatniej, wykorzystującej je sesji.

Procedury w języku T-SQL mogą być zagnieżdżane. Maksymalny poziom zagniedżenia wynosi 32.

Konstrukcja procedur w języku T-SQL nie jest usystematyzowana: nie ma większych wymogów odnośnie kolejności deklaracji elementów, czy struktury ciała procedury. Przykładowo, w procedurze *zewnetrzna* zadeklarowano tablicę w bloku `TRY`, już poza innymi deklaracjami. Wartości z tablicy odczytywane są poza blokiem `TRY`. Procedura wykona się poprawnie. Jako że nie ma wyszczególnionej sekcji deklaracji, konieczne jest stosowanie słowa kluczowego `DECLARE`. Podczas deklarowania parametrów typu `VARCHAR`, `DECIMAL` czy `NUMERIC` nie trzeba ograniczać ich co do długości, skali lub precyzji. Warto jednak to robić, w szczególności dla typów liczbowych, gdyż brak ograniczenia może spowodować różnice w dokładności wykonywanych obliczeń.

Wywołując procedurę można wykorzystać zarówno notację pozycyjną parametrów jak i odwołanie przez nazwę. Notacje te można łączyć, pod warunkiem, że odwołania przez pozycję nie wystąpią po odwołaniach przez nazwę:

```
exec przyklad @in_imie,
              @in_nazwisko,
              @liczba = 7.6,
              @nazwa2 = @out_nazwa2 OUTPUT,
              @nazwa3 = @out_nazwa3 OUTPUT,
              @liczba4 = @out_liczba4 OUTPUT,
              @nazwa1 = @out_nazwa1 OUTPUT;
```

Wywołując procedurę nie używa się nawiasów. Ich zastosowanie spowoduje błąd.

Wewnątrz podprogramu w T-SQL nie zadeklarujemy podprogramu lokalnego, tak jak można to zrobić w PL/SQL.

Język T-SQL pozwala na tworzenie funkcji skalarnych i tablicowych. Funkcje tablicowe dzielą się na funkcje *inline* oraz *multistatement*. Funkcje tablicowe *inline* to funkcje, w których zwracana tabela określona jest przez pojedynczą instrukcję `SELECT`. Funkcje *multistatement* pozwalają natomiast zwrócić dowolną tabelę utworzoną w ciele funkcji. Instrukcja tworzenia funkcji jest następująca:

```
CREATE FUNCTION [ nazwa_schematu. ] nazwa_funkcji
(lista_parametrów)
RETURNS {typ_danych | TABLE
  | @zwracana_zmienna TABLE <definicja_tabeli>}
[ WITH { [ ENCRYPTION ] | [ SCHEMABINDING ]
  | [ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ]
  | [ EXECUTE_AS_Clause ] } ][,...n]
[ AS ]
BEGIN
  ciało_funkcji
  RETURN [skalar | [(] zapytanie [)] ]
END
[ ; ]
```

Deklaracja parametru:

```
 @nazwa_parametru [ AS ][ nazwa_schematu_typu. ] typ_danych
  [ = wartość_domyślna]
```

Brak ograniczenia dla typów `VARCHAR` czy `DECIMAL` nie spowoduje błędu, wpłynie natomiast na otrzymywane rezultaty.
Poniżej zamieszczono przykład funkcji przyjmującej parametr typu `VARCHAR`.

W pierwszej wersji funkcji podano ograniczenie dla typu parametru, zaś w drugiej nie. Wynik otrzymany z drugiej wersji funkcji różni się od wyniku z wersji pierwszej: nie zawiera całego, przypisanego mu w ciele funkcji, ciągu znaków.

```
CREATE FUNCTION dbo.funkcja_3 (@p1 VARCHAR(20))
RETURNS VARCHAR(18)
BEGIN
  RETURN 'Ograniczenie' + ' ' + @p1;
END;

Wywołanie funkcji:
SELECT dbo.funkcja_3 ('typu.');

Otrzymany rezultat:
Ograniczenie typu.


ALTER FUNCTION dbo.funkcja_3 (@p1 VARCHAR)
RETURNS VARCHAR(18)
BEGIN
  RETURN 'Ograniczenie' + ' ' + @p1;
END;

Wywołanie funkcji:
SELECT dbo.funkcja_3 ('typu.');

Otrzymany rezultat:
Ograniczenie t
```

Podobnie zachowanie możemy zaobserwować dla parametru typu `DECIMAL`:

```
CREATE FUNCTION dbo.funkcja_4 (@p1 DECIMAL(4,2))
RETURNS DECIMAL(4,2)
BEGIN
  RETURN @p1 + 2.10;
END;

Wywołanie funkcji:
SELECT dbo.funkcja_4 (3.20);

Otrzymany rezultat:
5.30
```

```
ALTER FUNCTION dbo.funkcja_4 (@p1 DECIMAL)
RETURNS DECIMAL(4,2)
BEGIN
  RETURN @p1 + 2.10;
END;

Wywołanie funkcji:
SELECT dbo.funkcja_4 (3.20);

Otrzymany rezultat:
5.10
```

W celu zwrócenia wartości z funkcji w języku T-SQL należy zastosować następujące reguły:

- Dla funkcji zwracających wartości skalarne należy posłużyć się klauzulą `RETURNS typ_danych` w połączeniu z `RETURN skalar`;
- Dla funkcji tablicowych *inline* należy użyć klauzuli `RETURNS TABLE` w połączeniu z `RETURN [() zapytanie ()]`. `zapytanie` musi być pojedynczym zapytaniem;
- W definicji funkcji tablicowej *multistatement* należy zastosować zapis `RETURNS @zwracana_zmienna TABLE <definicja_tabeli>` wraz z instrukcją `RETURN` w ciele funkcji. W tym przypadku instrukcja `RETURN` występuje bez wyrażenia. `definicja_tabeli` składa się natomiast z definicji kolumn i ograniczeń (zarówno na poziomie kolumn, jak i tabeli).

Funkcje w języku T-SQL mogą zwracać wszystkie skalarne typy danych, oprócz `timestamp`. Nie mogą też zwracać kursorów.

Funkcja może przyjmować do 1024 parametrów. Nazwa parametru, tak jak w przypadku procedury, musi rozpoczynać się od znaku `@`. Dozwolone są wszystkie skalarne typy danych, za wyjątkiem typu `timestamp`. Funkcje nie mogą przyjmować parametrów typu `CURSOR` oraz `TABLE`. Przy wywołaniu funkcji z domyślną wartością danego parametru, w przeciwieństwie do procedur, parametru tego nie wolno ominąć. Należy w jego miejsce zastosować słowo `DEFAULT`:

```
CREATE FUNCTION dbo.funkcja_2
  (@liczba1 INT, @liczba2 INT = 61)
RETURNS INT
BEGIN
  RETURN (@liczba1 + @liczba2);
END
GO
```

```
DECLARE @z_wynik INT;
BEGIN
  SET @z_wynik = dbo.funkcja_2 (3, DEFAULT);
  SELECT @z_wynik;
END
```

Nawiasy w definicji i wywołaniu funkcji są obowiązkowe, nawet jeśli funkcja nie przyjmuje żadnych parametrów.

Podobnie jak dla procedur, tak i dla funkcji dostępne są instrukcje:

```
ALTER FUNCTION nazwa_funkcji ... [;]
DROP FUNCTION nazwa_funkcji [;]
```

Deklarując funkcję można skorzystać z opcji `SCHEMABINDING`. Chroni ona przed usunięciem obiektów bazy danych, od których funkcja jest zależna.

Funkcje w języku T-SQL nie mogą modyfikować stanu bazy danych. Mogą być stosowane tak, jak funkcje wbudowane SQL Serwer 2005. Funkcje zwracające wartości skalarne mogą być wywoływane instrukcją `EXECUTE` – tak jak procedury. Wówczas nie należy stosować nawiasów:

```
EXECUTE  dbo.funkcja_2 3, 4
```

Wewnątrz funkcji niedozwolona jest konstrukcja `TRY - CATCH`. Niedozwolona jest również związana z kursorami instrukcja `FETCH` zwracająca dane bezpośrednio do klienta. Można stosować jedynie instrukcję `FETCH INTO lokalne_zmienne`.

Jako, że funkcje nie mogą modyfikować stanu bazy danych, dlatego instrukcje `INSERT`, `UPDATE` oraz `DELETE` są dopuszczalne jedynie w odniesieniu do tabel lokalnych dla funkcji (zmiennych tabelarycznych).

Funkcji w T-SQL nie można przeciążać. Można je jednak zagnieżdżać. Tak jak dla procedur, istnieje ograniczenie do 32 poziomów.

Funkcje w języku T-SQL, podobnie jak we wcześniej omówionych, mogą być deterministyczne będź niedeterministyczne. Ta właściwość funkcji jest jednak określana przez serwer, użytkownik nie może jej ustawić. Może jedynie odczytać wartość jaka została podpisana pod właściość *IsDeterministic*:

```
SELECT OBJECTPROPERTYEX
  (OBJECT_ID(N'AdventureWorks.dbo.funkcja_2'),
   'IsDeterministic');
```

W ten sam sposób dostępnych jest jeszcze kilka innych właściwości funkcji.

Poniżej pokazano przykład prostej funkcji tablicowej:

```
CREATE FUNCTION dbo.Inline
  (@nr_kat_min INT, @nr_kat_max INT)
RETURNS TABLE
```

```
AS
RETURN
  (SELECT * FROM ksiazki
  WHERE nr_katalogowy BETWEEN @nr_kat_min AND @nr_kat_max);
```

Sposób wywołania funkcji *dbo.Inline* kojarzy się z wywoływaniem funkcji tablicowych we wcześniej omówionych językach:

```
SELECT * FROM dbo.InLine (2, 4);
SELECT tytul AS 'Tytul książki' FROM dbo.InLine (2, 4)
```

Podsumowując, w zakresie tworzenie funkcji i procedur omawiane w niniejszej pracy cztery języki proceduralnego SQL wykazują wiele różnic między sobą. Wciąż jednak najbardziej zbliżone są PL/SQL i PL/pgSQL. Język T-SQL znacznie odbiega od pozostałych.

3.2 Wyzwalacze

Wyzwalacze, podobnie jak procedury i funkcje, są nazwanymi blokami proceduralnego SQL. Są to procedury uruchamiane automatycznie w momencie wystąpienia określonego zdarzenia. Wyzwalacze nie przyjmują argumentów. Zdarzeniem wyzwalającym może być operacja DML, zdarzenie systemowe (takie jak otwarcie lub zamknięcie bazy danych) lub określone operacje DDL.

Język PL/SQL wyróżnia trzy podstawowe rodzaje wyzwalaczy:

- DML – uruchamiane w wyniku wykonania operacji DML. Są wykonywane obok wywołujących je instrukcji DML;
- Zastępujące – definiowane wyłącznie dla widoków. Są uruchamiane zamiast wyzwalających je instrukcji DML;
- Systemowe – uruchamiane w wyniku zaistnienia zdarzenia systemowego, np. otwarcia bazy danych. Mogą reagować także na operacje DDL, na przykład utworzenie tabeli. Są uruchamiane obok inicjujących je instrukcji.

Wyzwalacze PL/SQL tworzy się następującą instrukcją:

```
CREATE [OR REPALCE] TRIGGER [schemat.]nazwa_wyzwalacza
  {BEFORE | AFTER | INSTEAD OF} zdarzenie_wyzwalające
  [klauzula_REFERENCING]
  [FOR EACH ROW]
  [FOLLOWS schemat.nazwa_wyzwalacza]
  [ENABLE | DISABLE]
  [WHEN warunek_wyzwalacza]
ciało_wyzwalacza;
```

Wyzwalacze DML definiuje się dla instrukcji `INSERT`, `UPDATE` i `DELETE` dla tabeli bazy danych. Można zdefiniować wyzwalacze DML poziomu wiersza lub instrukcji. Wyzwalacze poziomu wiersza działają jeden raz dla każdego zmodyfikowanego wiersza. Wyzwalacze poziomu instrukcji – jeden raz dla każdej instrukcji. Dodatkowo wyzwalcze mogą działać przed lub po wykonaniu operacji na tabeli bazy danych.
Wyzwalacz może być zdefiniowany dla więcej niż jednej instrukcji DML. Kolejne instrukcje należy połączyć słowem kluczowym `OR`, np.:

```
AFTER INSERT OR UPDATE OR DELETE
```

Tabela zaś może być powiązana z wieloma wyzwalaczami.
Może istnieć wiele wyzwalaczy dla jednego zdarzenia.
Kod wyzwalacza DML wykonywany jest w ramach jednej transakcji wraz z instrukcją uruchamiającą wyzwalacz. `zdarzenie_wyzwalające` wyzwalaczy DML obejmuje nazwę tabeli i kolumny, których wywołanie uruchamia wyzwalacz. W klauzuli `WHEN` można umieścić warunek, jaki musi być spełniony, żeby wyzwalacz został uruchomiony.
Podobnie jak podprogramy, wyzwalcze mogą składać się z sekcji deklaracji, wykonawczej i obsługi wyjątków. Jedynie sekcja wykonawcza jest obowiązkowa. W odróżnieniu od podprogramów, w przypadku zastosowania sekcji deklaracji, obowiązkowo należy użyć słowa kluczowego `DECLARE`.
W przypadku istnienia wielu wyzwalaczy dla danej tabeli, w pierwszej kolejności uruchamiane są wyzwalacze `BEFORE` poziomu instrukcji, następnie `BEFORE` poziomu wiersza. Po wykonaniu operacji, która wyzwoliła wyzwalacz, wywoływane są wyzwalacze `AFTER` poziomu wiersza i na końcu `AFTER` poziomu instrukcji. Kolejność uruchamiania wyzwalaczy tego samego typu nie jest zdefiniowana. Kolejnością wykonania można sterować za pomocą klauzuli `FOLLOWS` (dla wyzwalaczy tego samego typu, dla tej samej tabeli).
Rozmiar wyzwalacza nie może przekroczyć 32 kilobajtów. Jako że wyzwalacze wywoływane są bardzo często, ich ciało powinno być możliwie najkrótsze. Ciało wyzwalacza może być instrukcją `CALL`, wywołującą inny podprogram.
W wyzwalaczach poziomu wiersza można uzyskać dostęp do danych przetwarzanego wiersza za pomocą identyfikatorów korelacji: `:OLD` i `:NEW`. Dla operacji `INSERT` jest zdefiniowany tylko operator `:NEW`, zaś dla `DELETE` jedynie `:OLD`. Przy `UPDATE` można korzystać z obu operatorów. Identyfikatory korelacji odnoszą się do wierszy, jednak możliwe jest odwoływanie się tylko do poszczególnych pól, np.:

```
:NEW.pole
```

Oracle definiuje dodatkowy operator: `:PARENT`. Jeżeli wyzwalacz działa dla tabeli zagnieżdżonej, zmienne związane `:NEW` i `:OLD` dotyczą tabeli zagnieżdżonej, zaś `:PARENT` wskazuje na bieżący wiersz tabeli nadrzędnej. Identyfikatory korelacji są dostępne jedynie w wyzwalaczach poziomu wiersza,

zastosowanie ich w wyzwalaczu poziomu instrukcji wywoła błąd kompilacji. Klauzula `REFERENCING` pozwala nadać inne nazwy identyfikatorom korelacji:

```
REFERENCING [OLD AS nazwa_dla_old] [NEW AS nazwa_dla_new]
            [PARENT AS nazwa_dla_parent]
```

W ciele wyzwalacza można posługiwać się wówczas nowymi nazwami identyfikatorów.
Również tylko dla wyzwalaczy poziomu wiersza poprawna jest klauzula `WHEN warunek_wyzwalacza`. Wyzwalacz zostanie uruchomiony jedynie dla tych wierszy, dla których warunek będzie miał wartość `TRUE`.
W wyzwalaczach można korzystać z predykatów `INSERTING`, `UPDATING` oraz `DELETING`. Przykładowo, `INSERTING` zwraca `TRUE` jeśli instrukcją, która wyzwoliła wyzwalacz, była instrukcja `INSERT`.

Wyzwalacze zastępujące (`INSTEAD OF`) można definiować jedynie dla widoków. Są uruchamiane zamiast wywołującej je instrukcji DML. Wyzwalacze zastępujące zawsze działają na poziomie wierszy, niezależnie od zastosowania bądź pominięcia klauzuli `FOR EACH ROW`.

Wyzwalacze systemowe są natomiast uruchamiane w odpowiedzi na zaistnienie określonego zdarzenia systemowego. Może to być otwarcie bazy danych czy zalogowanie się użytkownika. Wyzwalacze te mogą też reagować na określone operacje DDL (`CREATE`, `ALTER`, `DROP`). Składnia wyzwalaczy systemowych jest następująca:

```
CREATE [OR REPLACE] TRIGGER [schemat.]nazwa_wyzwalacza
  {BEFORE | AFTER}
  [lista_zdarzeń_DDL] [lista_zdarzeń_bazy_danych]
  ON {DATABASE | [schemat.]SCHEMA}
  [klauzula WHEN]
ciało_wyzwalacza;
```

Wyzwalacze systemowe są definiowane na poziomie schematu lub bazy danych. W ciele wyzwalaczy systemowych dostępna jest grupa funkcji (podobnych do omówionych wcześniej `INSERTING`, `UPDATING` oraz `DELETING`), dla przykładu: `ORA_DATABASE_NAME` czy `ORA_SYSEVENT`. Funkcje te można wywoływać nie tylko w wyzwalaczach, ale mogą wówczas zwrócić nieprawidłowy wynik. Wyzwalacze systemowe, w zależności od zdarzenia je uruchamiającego, są uruchamiane albo jako odrębna transakcja, albo jako część bieżącej transakcji użytkownika. Podczas stosowania klauzuli `WHEN` w wyzwalaczach systemowych należy pamiętać o pewnych ograniczeniach, na przykład wyzwalacze otwierania i zamykania bazy danych nie mogą mieć żadnych warunków.

Z wyzwalaczami związanych jest kilka ograniczeń. Najistotniejsze to takie, że nie można używać w nich instrukcji kontroli transakcji. Również wywoływane procedury i funkcje nie mogą ich zawierać (za wyjątkiem transakcji autonomicznych). Istnieje również problem tabel mutujących: tabel,

które są aktualnie modyfikowane przez instrukcje DML. Wyzwalacze nie mogą pobierać ani modyfikować danych z tabel mutujących, łącznie z samą tabelą wyzwalającą. Rozwiązanie tego problemu polega na tworzeniu pary wyzwalaczy: jednego poziomu wierszy i drugiego poziomu instrukcji.

Wyzwalacze można dezaktywować i aktywować:

```
ALTER TRIGGER nazwa_wyzwalacza {DISABLE | ENABLE};
```

Możliwe jest wyłączenie lub włączenie wszystkich wyzwalaczy danej tabeli:

```
ALTER TABLE nazwa_tabeli {DISABLE | ENABLE} ALL TRIGGERS;
```

Całkowite usunięcie wyzwalacza realizuje natomiast instrukcja `DROP`:

```
DROP TRIGGER nazwa_wyzwalacza;
```

W poniższym przykładzie najpierw tworzona jest tabela audytu `audyt`. Wyzwalacz *RejestrujOperacje* poziomu wiersza zapisuje do tabeli *audyt* informacje odnośnie wykonanych na tabeli `ksiazki` operacji `INSERT`, `UPDATE` oraz `DELETE`. Nałożony jest też warunek `WHEN` na wiersze. Wyzwalacze *Rejestruj* i *Rejestruj_2* są wyzwalaczami poziomu instrukcji i działają tylko w przypadku `UPDATE`. Klauzula `FOLLOWS` w wyzwalaczu *Rejestruj* określa, że ma się on wykonać po wyzwalaczu *Rejestruj_2*.

```
CREATE SEQUENCE audyt_id_seq INCREMENT BY 1 START WITH 1;


CREATE TABLE audyt (
  id_audytu    NUMBER(4,0)   PRIMARY KEY,
  kod_zmiany   NUMBER(2,0)   NOT NULL,
  opis         VARCHAR2(200) NOT NULL
);


CREATE OR REPLACE TRIGGER RejestrujOperacje
  AFTER INSERT OR UPDATE OR DELETE ON ksiazki
  FOR EACH ROW
  WHEN (OLD.nr_katalogowy >= 9877
    OR OLD.nr_katalogowy IS NULL)
DECLARE
  z_NrKat ksiazki.nr_katalogowy%TYPE;
  z_KodOperacji NUMBER(1,0);
  z_Opis audyt.opis%TYPE;
BEGIN
  IF INSERTING THEN
    z_KodOperacji := 1;
```

```
    z_Opis := 'Wstawiono ' || :NEW.tytul;
    z_NrKat := :NEW.nr_katalogowy;
  ELSIF UPDATING THEN
    z_KodOperacji := 2;
    z_Opis := 'Zmieniono ' || :OLD.tytul ||
      ' na ' || :NEW.tytul;
    z_NrKat := :OLD.nr_katalogowy;
  --ELSIF DELETING THEN
  ELSE
    z_KodOperacji := 3;
    z_Opis := 'Usunięto ' || :OLD.tytul;
    z_NrKat := :OLD.nr_katalogowy;
  END IF;

  z_Opis := z_Opis || ' (nr pozycji: ' || z_NrKat || ').';

  INSERT INTO audyt(id_audytu, kod_zmiany, opis)
    VALUES (audyt_id_seq.nextval, z_KodOperacji, z_Opis);

EXCEPTION
  WHEN OTHERS THEN
  RAISE_APPLICATION_ERROR (-20041, 'Wystąpił błąd.');
END RejestrujOperacje;
/


CREATE OR REPLACE TRIGGER Rejestruj_2
  AFTER UPDATE OF tytul ON ksiazki
BEGIN
  INSERT INTO audyt (id_audytu, kod_zmiany, opis)
    VALUES (audyt_id_seq.nextval, 2,
      'Zmieniono tytuł - wyzwalacz Rejestruj_2.');
END;
/


CREATE OR REPLACE TRIGGER Rejestruj
  AFTER UPDATE OF tytul ON ksiazki
  FOLLOWS Rejestruj_2
BEGIN
  INSERT INTO audyt (id_audytu, kod_zmiany, opis)
    VALUES (audyt_id_seq.nextval, 2, 'Zmieniono tytuł.');
END;
/
```

Oto kod, który wymusza wykonanie wyzwalaczy:

```
BEGIN
  INSERT INTO ksiazki VALUES (9876, 3, 'Czarodziejski
    kapelusz', 'Juliusz Brett', 2007, 52.00);
  UPDATE ksiazki
    SET tytul = 'Czarodziejski gadający kapelusz'
    WHERE nr_katalogowy = 9876;
  DELETE FROM ksiazki WHERE nr_katalogowy = 9876;
  COMMIT;
END;
/
BEGIN
  INSERT INTO ksiazki VALUES (9877, 3, 'Czarodziejski
    kapelusz 2', 'Juliusz Brett', 2007, 52.00);
  UPDATE ksiazki
    SET tytul = 'Czarodziejski gadający kapelusz 2'
    WHERE nr_katalogowy = 9877;
  DELETE FROM ksiazki WHERE nr_katalogowy = 9877;
  COMMIT;
END;
/
BEGIN
  INSERT INTO ksiazki VALUES (10000, 3, 'Czarodziejski
    kapelusz 3', 'Juliusz Brett', 2007, 52.00);
  UPDATE ksiazki
    SET tytul = 'Czarodziejski gadający kapelusz 3'
    WHERE nr_katalogowy = 10000;
  DELETE FROM ksiazki WHERE nr_katalogowy = 10000;
  COMMIT;
END;
/
```

Zawartość tabeli `audyt` po wykonaniu powyższych trzech bloków kodu jest następująca:

```
SELECT id_audytu AS ID, kod_zmiany AS KD, opis AS Opis
  FROM audyt ORDER BY id_audytu;
```

```
ID  KD  Opis
-------------------------------------------------------------
24  1   Wstawiono Czarodziejski kapelusz (nr pozycji: 9876).
25  2   Zmieniono tytuł - wyzwalacz Rejestruj_2.
26  2   Zmieniono tytuł.
27  1   Wstawiono Czarodziejski kapelusz 2
```

```
          (nr pozycji: 9877).
28  2   Zmieniono Czarodziejski kapelusz 2 na Czarodziejski
          gadający kapelusz 2 (nr pozycji: 9877).
29  2   Zmieniono tytuł - wyzwalacz Rejestruj_2.
30  2   Zmieniono tytuł.
31  3   Usunięto Czarodziejski gadający kapelusz 2
          (nr pozycji: 9877).
32  1   Wstawiono Czarodziejski kapelusz 3
          (nr pozycji: 10000).
33  2   Zmieniono Czarodziejski kapelusz 3 na Czarodziejski
          gadający kapelusz 3 (nr pozycji: 10000).
34  2   Zmieniono tytuł - wyzwalacz Rejestruj_2.
35  2   Zmieniono tytuł.
36  3   Usunięto Czarodziejski gadający kapelusz 3
          (nr pozycji: 10000).
```

Wyzwalacze w języku PL/pgSQL tworzy się w dwóch krokach: najpierw należy utworzyć funkcję, która będzie wywoływana przez wyzwalacz, a następnie zdefiniować wyzwalacz wywołujący utworzoną wcześniej funkcję. Składnia instrukcji definiującej wyzwalacz jest następująca:

```
CREATE TRIGGER nazwa_wyzwalacza
  { BEFORE | AFTER } { zdarzenie_wyzwalające [ OR ... ] }
  ON tabela [ FOR [ EACH ] { ROW | STATEMENT } ]
  EXECUTE PROCEDURE nazwa_funkcji ( argumenty );
```

Język PL/pgSQL nie pozwala na tworzenie wyzwalaczy typu `INSTEAD OF`. Wyzwalacz nie może zawierać ani warunku `WHEN` ani klauzuli `REFERENCING`. Wyzwalacze mogą być definiowane dla instrukcji DML: `INSERT`, `UPDATE` i `DELETE`, na poziomie wiersza lub instrukcji. Mogą działać przed lub po wykonaniu danej operacji.
Funkcja skojarzona z wyzwalaczem może zawierać sekcję deklaracji, wykonawczą oraz obsługi wyjątków. Nie może przyjmować argumentów i musi zwracać specjalny typ `trigger`. Funkcja ta musi zwrócić wartość `NULL` albo rekord mający strukturę tabeli, dla której został odpalony wyzwalacz.
Wewnątrz funkcji związanej z wyzwalaczem dostępnych jest kilka zmiennych specjalnych. Wśród nich znajdują się `NEW` i `OLD`, których działanie jest takie samo jak w PL/SQL. Inne to na przykład `TG_NAME`, zawierająca nazwę uruchomionego wyzwalacza, czy `TG_WHEN` określająca czy wyzwalacz został zdefiniowany jako `BEFORE`, czy `AFTER`. Odpowiednikiem `INSERTING`, `UPDATING` i `DELETING` z PL/SQL jest tu zmienna `TG_OP`, która przyjmuje jedną z wartości: `INSERT`, `UPDATE` lub `DELETE`.
Pomimo, że funkcja skojarzona z wyzwalaczem nie może przyjmować parametrów, to sam wyzwalacz może je przyjmować. Parametrów tych nie deklaruje się, lecz po prostu wywołuje funkcję skojarzoną z wyzwalaczem tak,

jakby przyjmowała parametry. Są one wówczas dostępne wewnątrz funkcji poprzez zmienną `TG_ARGV[]`.
Instrukcja `CREATE TRIGGER` w języku PL/pgSQL nie może zawierać słowa kluczowego `OR REPLACE`. Nie jest również możliwe zbudowanie wyzwalacza dla zmian zachodzących dla konkretnej kolumny tabeli, czyli np.: `UPDATE OF tytul ON ksiazki`. Podczas odwoływania się do zmiennych `OLD` i `NEW` nie stosuje się poprzedzającego dwukropka, tak jak miało to miejsce w PL/SQL.
Jeżeli użytkownik zdefiniuje kilka wyzwalaczy tego samego typu dla danego zdarzenia, będą one wywoływane w porządku alfabetycznym.
W języku PL/pgSQL, wewnątrz funkcji skojarzonej z wyzwalaczem można zawrzeć instrukcje DML odwołujące się do tabeli dla której uruchamiany jest wyzwalacz.

Oto przykład tworzenia wyzwalacza w języku PL/pgSQL. W szczególności pokazano tu w jaki sposób korzystać ze zmiennych `TG_OP` i `TG_ARGV[]`:

```
CREATE OR REPLACE FUNCTION trg_ksiazki()
RETURNS trigger AS $$
DECLARE
  z_NrKat ksiazki.nr_katalogowy%TYPE;
  z_KodOperacji INTEGER;
  z_Opis audyt.opis%TYPE;
  z_Param_1 INTEGER;
  z_Param_2 VARCHAR(50);
BEGIN
  IF (TG_OP = 'INSERT') THEN
    z_KodOperacji := 1;
    z_Opis := 'Wstawiono ' || NEW.tytul;
    z_NrKat := NEW.nr_katalogowy;
  ELSIF (TG_OP = 'UPDATE') THEN
    z_KodOperacji := 2;
    z_Opis := 'Zmieniono ' || OLD.tytul
      || ' na ' || NEW.tytul;
    z_NrKat := OLD.nr_katalogowy;
  ELSIF (TG_OP = 'DELETE') THEN
    z_KodOperacji := 3;
    z_Opis := 'Usunieto ' || OLD.tytul;
    z_NrKat := OLD.nr_katalogowy;
  END IF;

  z_Opis := z_Opis ||
    ' (nr pozycji: ' || z_NrKat || ').';

  z_Param_1 := TG_ARGV[0];
```

```
    z_Param_2 := TG_ARGV[1];

    INSERT INTO audyt
      (id_audytu, kod_zmiany, opis, param_1, param_2)
      VALUES (nextval('audyt_id_seq'), z_KodOperacji,
        z_Opis, z_Param_1, z_Param_2);

    RETURN NULL;
  END;
  $$ LANGUAGE plpgsql;


  CREATE TRIGGER RejestrujOperacje
    AFTER INSERT OR UPDATE OR DELETE ON ksiazki
    FOR EACH ROW
    EXECUTE PROCEDURE trg_ksiazki(100, 'Kod = 100');
```

Wyzwalacz można dezaktywować i ponownie aktywować za pomocą instrukcji `ALTER TABLE`, np.:

```
ALTER TABLE ksiazki DISABLE TRIGGER RejestrujOperacje;
```

Możliwa jest również zmiana nazwy wyzwalacza:

```
ALTER TRIGGER nazwa_wyzwalacza ON tabela
  RENAME TO nowa_nazwa;
```

Podczas usuwania wyzwalacza, w języku PL/pgSQL konieczne jest podanie nazwy tabeli, której dotyczy wyzwalacz. Na przykład:

```
DROP TRIGGER RejestrujOperacje ON ksiazki;
```

Język SQL PL umożliwa zarówno tworzenie wyzwalaczy `BEFORE` i `AFTER`, jak i `INSTEAD OF`. Składnia tworzenia wyzwalaczy w SQL PL jest następująca:

```
CREATE TRIGGER [schemat.]nazwa_wyzwalacza
  {[NO CASCADE] BEFORE | AFTER | INSTEAD OF}
    zdarzenie_wyzwalające
  [klauzula REFERENCING]
  {FOR EACH ROW | FOR EACH STATEMENT}
  [WHEN warunek_wyzwalacza]
ciało_wyzwalacza;
```

Tak jak w PL/pgSQL nie istnieje słowo kluczowe `OR REPLACE` w instrukcji tworzącej wyzwalacz.
Zdarzenie wyzwalające to jedna lub kilka (połączonych spójnikiem `OR`) operacji DML w połączeniu z nazwą tabeli (lub widoku dla wyzwalaczy typu

INSTEAD OF).
Dla operacji UPDATE możliwe jest zdefiniowanie nazwy kolumny (UPDATE OF nazwa_kolumny).
Klauzula REFERENCING działa podobnie jak w PL/SQL, z tą różnicą, że oprócz zmiennych OLD i NEW, pozwala również przemianować OLD TABLE oraz NEW TABLE. OLD TABLE i NEW TABLE są to tabele tymczasowe, zawierające kompletne zbiory danych, na których działa wyzwalacz. Są dostępne jedynie w wyzwalaczach typu AFTER oraz INSTEAD OF.
Zmienne związane OLD i NEW, tak jak w PL/SQL i PL/pgSQL nie mogą być używane w wyzwalaczach typu FOR EACH STATEMENT. Opcja FOR EACH STATEMENT w SQL PL może być używana jedynie dla wyzwalaczy AFTER, zaś warunek WHEN nie może wystąpić w wyzwalaczu typu INSTEAD OF.
Wyzwalacze INSTEAD OF, tak jak w PL/SQL, definiuje się dla widoków i może istnieć tylko jeden wyzwalacz tego typu dla poszczególnej operacji dla danego widoku.
W ciele wyzwalaczy typu BEFORE SQL PL nie dopuszcza stosowania instrukcji INSERT, UPDATE i DELETE. Instrukcje te można stosować jedynie w wyzwalaczach typu AFTER.
W ciele wyzwalacza można, podobnie jak w PL/SQL, wywoływać inne procedury instrukcją CALL. W ciele wywoływanej procedury nie mogą być użyte instrukcje nieobsługiwane w danym typie wyzwalacza. Procedura ta nie może również zawierać instrukcji kontroli transakcji (COMMIT i ROLLBACK).
SQL PL dopuszcza tworzenie wielu wyzwalaczy danego typu dla danej tabeli. Kolejność ich wywoływania jest wówczas zgodna z kolejnością, w której zostały utworzone.
W ciele wyzwalacza możliwe jest odwołanie się do tabeli, dla której wyzwalacz został zdefiniowany.
Język SQL PL nie oferuje funkcjonalności takiej jak predykaty INSERTING, UPDATING i DELETING w PL/SQL czy zmienna TG_OP w PL/pgSQL.
W wyzwalaczach SQL PL zubożona jest również obsługa błędów (na przykład umieszczenie w ciele wyzwalacza instrukcji DECLARE SQLSTATE zakończy się błędem).

Oto przykład wyzwalacza w języku SQL PL:

```
CREATE TRIGGER MAIN.tr_new_frame_test
AFTER INSERT ON MAIN.Frame
REFERENCING NEW AS n
FOR EACH ROW
BEGIN ATOMIC
  DECLARE v_m_id_frm INTEGER;

  SET v_m_id_frm = (SELECT COUNT(*) FROM MAIN.Frame
    WHERE id_frm <> n.id_frm AND iFrameNo = n.iframeno
    AND iDayNight = n.idaynight AND iCamId = n.icamid);
```

```
  IF v_m_id_frm = 0 THEN
    CALL MAIN.UpdateFrameStat
      (n.sobject, n.idaynight, n.fra, n.fdec);
  END IF;
END
@
```

W celu usunięcia wyzwalacza należy się posłużyć instrukcją:

```
DROP TRIGGER [schemat.]nazwa_wyzwalacza;
```

W omawianej wersji języka SQL PL nie można włączyć lub wyłączyć wyzwalacza, tak jak jest to możliwe w pozostałych omawianych tu językach. (Aktywowanie i dezaktywowanie wyzwalaczy jest podobno możliwe w DB2 Universal Database dla serwerów iSeries.)

W języku Transact-SQL można tworzyć wyzwalacze dla instrukcji DML oraz instrukcji DDL, a także wyzwalacze typu `LOGON`.
Wyzwalacze DML standardowo obejmują instrukcje `INSERT`, `UPDATE` oraz `DELETE`. Wyzwalacze DDL tworzy się dla instrukcji `CREATE`, `ALTER` oraz `DROP`. Wyzwalacze `LOGON` reagują na rozpoczęcie sesji przez użytkownika.
T-SQL pozwala na tworzenie dwóch typów procedur wyzwalanych: `AFTER` i `INSTEAD OF`. Nie istnieją tu wyzwalacze typu `BEFORE`.
Tak jak w poprzednio omówionych językach, wyzwalacze w SQL Server 2005 stanowią część transakcji, która spowodowała ich uruchomienie. W przeciwieństwie do omówionych wcześniej języków, w wyzwalaczach T-SQL można używać instrukcji `ROLLBACK TRAN`. Powoduje ona wycofanie całej transakcji, łącznie z operacjami, które wyzwoliły wyzwalacz.
Składnia instrukcji tworzącej wyzwalacz DML jest następująca:

```
CREATE TRIGGER [ schemat. ]nazwa_wyzwalacza
  ON { tabela| widok }
  [ WITH  [ ENCRYPTION ] [ EXECUTE AS Clause ] [ ,...n ] ]
  { FOR | AFTER | INSTEAD OF }
  { [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }
  [ WITH APPEND ]
  [ NOT FOR REPLICATION ]
AS { ciało_wyzwalacza }
```

Wyzwalacze typu `DML AFTER` mogą być tworzone tylko w odniesieniu do tabel trwałych. Można je tworzyć za pomocą słowa kluczowego `FOR` lub `AFTER`. Działają zawsze na poziomie instrukcji (nie na poziomie wiersza).

Jeśli dla danej tabeli zdefiniowano wiele wyzwalaczy tego samego typu, to za pomocą procedury składowanej *sp_settriggerorder* można określić, który z nich zostanie wykonany jako pierwszy, a który jako ostatni. Kolejność wywołania pozostałych wyzwalaczy, pomiędzy pierwszym a ostatnim, pozostaje niezdefiniowana.

Wewnątrz wyzwalaczy DML programista może odwoływać się do dwóch tabel: `inserted` i `deleted`. Są to odpowiedniki zmiennych `OLD` i `NEW`, z tą różnicą, że mają postać tabeli a nie rekordu. Tabele te mają strukturę tabeli, dla której zdefiniowano wyzwalacz.

T-SQL nie udostępnia szczególnej funkcjonalności do sprawdzenia, jaka instrukcja spowodowała uruchomienie wyzwalacza. Podobnie jak w SQL PL nie znajdziemy tu odpowiedników predykatów `INSERTING`, `DELETING` oraz `UPDATING` z PL/SQL czy zmiennej `TG_OP` z PL/pgSQL. W celu zidentyfikowania instrukcji, która uruchomiła wyzwalacz, należy sprawdzić zawartość tabel `inserted` i `deleted`. Jeżeli miała miejsce instrukcja `INSERT` – wiersze będą znajdowały się tylko w tabeli `inserted`, jeśli `DELETE` – tylko w tabeli `deleted`, jeśli `UPDATE` – w obu tabelach.

Definicja wyzwalacza nie zawiera klauzuli `UPDATE OF nazwa_kolumny`. Wyzwalacze reagujące na operacje na konkretnej kolumnie można tworzyć przy zastosowaniu albo predykatu `UPDATE`, albo funkcji `COLUMNS_UPDATED`. `UPDATE(nazwa_kolumny)` zwraca `TRUE`, jeśli kolumna została wymieniona w klauzuli `SET` polecenia `UPDATE`, które uruchomiło wyzwalacz. Funkcja `COLUMNS_UPDATED` zwraca łańcuch bitowy, w którym każdej kolumnie tabeli bazowej wyzwalacza odpowiada jeden bit o wartości 1 jeśli kolumna była modyfikowana i 0 jeśli nie była.

Wyzwalacze DML `INSTEAD OF` mogą być tworzone zarówno dla tabel jak i dla widoków. W tym przypadku również dostępne są tabele `inserted` i `deleted`. Zawierają one jednak dane, które dopiero mają być zmienione, a nie dane, które zostały zmienione. Istnieje możliwość utworzenia tylko jednego wyzwalacza `INSTEAD OF` dla danego rodzaju instrukcji dla danego obiektu.

Wyzwalacze DDL mogą być wyłącznie typu `AFTER`. Można je tworzyć na poziomie serwera lub bazy danych, dla konkretnej instrukcji (np. `CREATE TABLE`) lub dla grupy instrukcji (np. `DDL_DATABASE_LEVEL_EVENTS`). Wewnątrz wyzwalacza DDL informacje o zdarzeniu wywołującym są dostępne poprzez funkcję *eventdata*. Zwraca ona wartość typu `XML` zawierającą zestaw informacji o zdarzeniu. W celu wyłuskania konkretnej informacji (np. rodzaju zdarzenia), należy posłużyć się `XQuery`.
Wyzwalacze DDL tworzy się instrukcją:

```
CREATE TRIGGER nazwa_wyzwalacza
  ON { ALL SERVER | DATABASE }
  [ WITH [ ENCRYPTION ] [ EXECUTE AS Clause ] [ ,...n ] ]
  { FOR | AFTER } { zdarzenie | grupa_zdarzeń } [ ,...n ]
AS { ciało_wyzwalacza }
```

Również wyzwalacze `LOGON` mogą być tylko typu `AFTER`. Tworzy się je instrukcją:

```
CREATE TRIGGER nazwa_wyzwalacza
```

```
  ON ALL SERVER
  [ WITH [ ENCRYPTION ] [ EXECUTE AS Clause ] [ ,...n ] ]
  { FOR | AFTER } LOGON
AS { ciało_wyzwalacza }
```

W definicji wyzwalaczy typu `LOGON` oraz w definicji wyzwalaczy DDL w nazwie wyzwalacza nie stosuje się nazwy schematu. Nazwę schematu można podawać tylko w nazwach wyzwalaczy DML.

Język Transact-SQL nie pozwala na definiowanie warunku `WHEN` wyzwalacza (tak jak PL/pgSQL). W ciele wyzwalacza można wywoływać procedury składowane oraz przeprowadzać operacje na tabeli bazowej wyzwalacza. Definicję wyzwalacza można zmienić za pomocą instrukcji `ALTER`. Oto przykład:

```
ALTER TRIGGER test
ON dbo.ksiazki
AFTER INSERT, UPDATE, DELETE
AS
  DECLARE @licznik_i INT;
  DECLARE @licznik_d INT;
  DECLARE @kod_operacji INT;
  DECLARE @opis VARCHAR(200);
  DECLARE @nrKat INT;

  SET @licznik_i = (SELECT COUNT(*)FROM inserted);
  SET @licznik_d = (SELECT COUNT(*)FROM deleted);
  IF (@licznik_i <> 0 AND @licznik_d = 0)
    -- byl INSERT
    BEGIN
      SET @kod_operacji = 1;
      SET @opis = 'Wstawiono ksiazke';
    END;
  ELSE IF (@licznik_i <> 0 AND @licznik_d <> 0)
    -- byl UPDATE
    BEGIN
      SET @kod_operacji = 2;
      SET @opis = 'Zmieniono ksiazke';
    END;
  ELSE IF (@licznik_i = 0 AND @licznik_d <> 0)
    -- bylo DELETE
    BEGIN
      SET @kod_operacji = 3;
      SET @opis = 'Usunieto ksiazke';
    END;
 ELSE
```

```
    BEGIN
      SET @kod_operacji = 0;
      SET @opis =
        '??? - np. nie znaleziono rekordu lub inne';
    END;

INSERT INTO audyt(kod_zmiany, opis)
  VALUES (@kod_operacji, @opis);
INSERT INTO ksiazki VALUES
  ('Wpis do tabeli bazowej wyzwalacza', 'trigger test');

exec lokalny;1 6
GO
```

Wzywalacz można usunąć instrukcją `DROP`. Dla wyzwalaczy DML instrukcja ta przyjmuje postać:

```
DROP TRIGGER [schemat.]nazwa_wyzwalacza [ ,...n ] [;]
```

Dla wyzwalaczy DDL ma postać:

```
DROP TRIGGER nazwa_wyzwalacza [ ,...n ]
  ON { DATABASE | ALL SERVER } [;]
```

zaś w przypadku wyzwalaczy `LOGON`:

```
DROP TRIGGER nazwa_wyzwalacza [ ,...n ]
  ON ALL SERVER [;]
```

Wyzwalacz można również dezaktywować i ponownie aktywować:

```
{ENABLE | DISABLE } TRIGGER
  {[schemat.]nazwa_wyzwalacza [ ,...n ] | ALL}
  ON { nazwa_obiektu | DATABASE | ALL SERVER } [;]
```

lub:

```
ALTER TABLE [schemat.]nazwa_tabeli
  {ENABLE | DISABLE} TRIGGER
  {ALL | nazwa_wyzwalacza [ ,...n ]}[;]
```

Na przykład:

```
ALTER TABLE dbo.ksiazki DISABLE TRIGGER test;
```

Zarówno wyzwalacze typu DML jak i typu DDL mogą być zagnieżdżane do 32 poziomów. Ponadto wyzwalacze typu `AFTER` (DML i DDL) mogą być wywoływane rekurencyjnie, również z ograniczeniem do 32 poziomów. Wyzwalaczy `INSTEAD OF` nie można wywoływać rekurencyjnie.

3.3 Dynamiczny SQL

Dynamiczny język SQL pozwala tworzyć instrukcje SQL w czasie wykonywania podprogramu. Umożliwia budowanie instrukcji odwołujących się do obiektów nieistniejących w chwili kompilacji programu oraz tworzenie kodu dostosowującego się do zmian definicji tabel. Mechanizm ten pozwala konstruować instrukcje w zależności od danych wejściowych wprowadzonych przez użytkownika. Za pomocą dynamicznego SQL można uruchamiać zarówno insrukcje DDL jak i DML.

Język PL/SQL oferuje dwie metody korzystania z dynamicznego SQL:

- NDS (*Native Dynamic SQL*) – wbudowany dynamiczny język SQL,
- DBMS_SQL – wbudowany pakiet Oracle do obsługi dynamiczneg SQL.

NDS jest prostszy w użyciu – ma łatwiejszą składnię, dzięki czemu programy są łatwiejsze do pisania i czytania. Ponadto działa szybciej niż DBMS_SQL. DBMS_SQL pozwala natomiast budować instrukcje SQL, dla których nie jest znana (w czasie kompilacji kodu) liczba i typy parametrów wejściowych i wyjściowych. W przypadku NDS budowana dynamicznie instrukcja SQL musi mieć dokładnie określoną liczbę, nazwy oraz typy parametrów.

Budowa dynamicznej instrukcji SQL za pomocą NDS rozpoczyna się od utworzenia zmiennej typu `VARCHAR2` i zapisania w niej treści instrukcji. Jeżeli w instrukcji występują jakieś wartości (np. w klauzuli `WHERE`), wartości te można zapisać bezpośrednio („na sztywno"), bądź zastosować zmienne powiązane.
Kolejnym krokiem jest wykonanie zbudowanej instrukcji za pomocą polecenia `EXECUTE IMMEDIATE`. Przyjmuje ono jeden argument typu `VARCHAR2`, reprezentujący instrukcję w języku SQL. `EXECUTE IMMEDIATE` udostępnia trzy klauzule: `INTO`, `RETURNING INTO` oraz `USING` w celu powiązania zwracanych wartości ze zmiennymi. Klauzula `INTO` wiąże wartości i zmienne na podstawie kolejności ich wystąpienia. Klauzula `RETURNING INTO` wiąże na podstawie nazwy. Obie działają tylko dla instrukcji zwracających jeden wiersz. Klauzula `INTO` obsługuje tylko tryb `OUT`, podczas, gdy dwie pozostałe tryb `IN OUT`.
W trakcie tworzenia instrukcji (jako ciągu znakowego) należy pominąć kończący ją średnik. Wyjątkiem jest sytuacja, gdy w składanym ciągu umieszcza się instrukcje tworzące anonimowy blok PL/SQL – wówczas należy zastosować średnik po tych instrukcjach.
Oto przykład użycia dynamicznego SQL (NDS) w PL/SQL:

```
CREATE OR REPLACE PROCEDURE przenies_ksiazke
(p_nr_kat IN NUMBER, p_przyjeta IN BOOLEAN) AS
  instrukcja VARCHAR2(200);
  z_tytul  ksiazki.tytul%TYPE;
```

```
  z_autor  ksiazki.autor%TYPE;
  z_tabela VARCHAR2(20);
BEGIN
  SELECT tytul, autor INTO z_tytul, z_autor FROM ksiazki
    WHERE nr_katalogowy = p_nr_kat;
  IF (p_przyjeta = TRUE) THEN
    z_tabela := 'ksiazki_przyjete';
  ELSE
   z_tabela := 'ksiazki_odrzucone';
  END IF;

  instrukcja := 'INSERT INTO ' || z_tabela ||
    ' VALUES (:kol_1, :kol_2, :kol_3)';
  EXECUTE IMMEDIATE instrukcja
    USING p_nr_kat, z_tytul, z_autor;
  COMMIT;
END;
/
```

Wykorzystanie pakietu DBMS_SQL jest podobne – również należy zbudować instrukcję w postaci ciągu znakowego, a następnie zastosować polecenie `EXECUTE IMMEDIATE`. Różnica polega na wykonaniu kilku dodatkowych operacji, niezbędnych podczas korzystania z pakietu DBMS_SQL:

```
CREATE OR REPLACE PROCEDURE przenies_ksiazke_dbms_sql
(p_nr_kat IN NUMBER, p_przyjeta IN BOOLEAN) AS
  nr_kursora INTEGER := dbms_sql.open_cursor;
  stan INTEGER;
  instrukcja VARCHAR2(200);
  z_tytul  ksiazki.tytul%TYPE;
  z_autor  ksiazki.autor%TYPE;
  z_tabela VARCHAR2(20);
BEGIN
  SELECT tytul, autor INTO z_tytul, z_autor FROM ksiazki
    WHERE nr_katalogowy = p_nr_kat;
  IF (p_przyjeta = TRUE) THEN
    z_tabela := 'ksiazki_przyjete';
  ELSE
   z_tabela := 'ksiazki_odrzucone';
  END IF;

  instrukcja := 'INSERT INTO ' || z_tabela ||
    ' VALUES (:kol_1, :kol_2, :kol_3)';
  dbms_sql.parse (nr_kursora, instrukcja, dbms_sql.native);
  dbms_sql.bind_variable (nr_kursora, 'kol_1', p_nr_kat);
```

```
    dbms_sql.bind_variable (nr_kursora, 'kol_2', z_tytul);
    dbms_sql.bind_variable (nr_kursora, 'kol_3', z_autor);
    stan := dbms_sql.execute (nr_kursora);
    dbms_sql.close_cursor(nr_kursora);
    COMMIT;
END;
/
```

Pakiet DBMS_SQL wymaga użycia kursora oraz zadeklarowania zmiennych zawierających numer kursora, oraz wartość zwracaną przez instrukcję `EXECUTE` wywołaną dla kursora o podanym numerze. W celu wykonania złożonej w postaci ciągu znakowego instrukcji należy najpierw otworzyć kursor, następnie wykonać wiązanie instrukcji z numerem kursora oraz wiązanie zmiennych, a dopiero później uruchomić instrukcję `DBMS_SQL.EXECUTE`. Na koniec należy zamknąć używany kursor.

Dynamiczny SQL w PL/SQL pozwala również obsługiwać przetwarzanie masowe, które umożliwia zapisywanie kolekcji do bazy danych i pobieranie ich z niej z wyeliminowaniem przetwarzania na poziomie wierszy. W tym celu można się posłużyć zarówno NDS jak i DBMS_SQL.

Język PL/pgSQL udostępnia instrukcję `EXECUTE` do wykonywania dynamicznego SQL. Tak jak instrukcja `EXECUTE IMMEDIATE` w PL/SQL, tak i instrukcja `EXECUTE` w PL/pgSQL przyjmuje jeden parametr, typu `text` lub `varchar`, zawierający treść instrukcji SQL. Podobnie jak w PL/SQL można rozpocząć od zadeklarowania zmiennej i przypisania jej treści instrukcji przeznaczonej do wykonania. W odróżnieniu od PL/SQL, PL/pgSQL nie pozwala na stosowanie zmiennych powiązanych. Wszystkie potrzebne wartości muszą być bezpośrednio umieszczone w konstruowanym ciągu. Tak skonstruowany ciąg wystarczy przekazać do instrukcji `EXECUTE`. Instrukcję `EXECUTE` można wywołać z klauzulą `INTO` w celu pobrania zwróconych wartości do zmiennych. PL/pgSQL nie oferuje tutaj klauzuli `RETURNING INTO` ani `USING`.

Budując w PL/pgSQL instrukcję SQL w postaci ciągu znakowego należy pamiętać o tym, że podstawiane dynamicznie wartości mogą zawierać znaki apostrofu. Nieoczekiwane pojawienie się takiego znaku w budowanej dynamicznie instrukcji wywoła błąd. Żeby temu zapobiec należy w tworzonej instrukcji stosować funkcje `guote_ident` i `quote_literal`. Funkcji `guote_ident` należy używać w celu ograniczenia ciągów reprezentujących nazwy tabel i kolumn. Funkcji `quote_literal` natomiast do ograniczania wyrażeń reprezentujących wartości w postaci ciągów znakowych.

Poniższe dwa przykłady obrazują zastosowanie dynamicznego SQL w języku PL/pgSQL. W pierwszym dynamiczny SQL zostosowano do utworzenia instrukcji DML, w drugim do utworzenia instrukcji DDL:

```
--Przyklady napisane na PostgreSQL 8.2.5
```

```
CREATE OR REPLACE FUNCTION dynamiczny
  (IN p_nr_kat int, IN p_kolumna VARCHAR(50),
   IN p_wartosc VARCHAR(50))
RETURNS VOID AS
$$
DECLARE
  instrukcja varchar(400);
BEGIN
  instrukcja := 'UPDATE ksiazki SET '
    || quote_ident(p_kolumna)
    || ' = ' || quote_literal(p_wartosc)
    || ' WHERE nr_katalogowy = '
    || quote_literal(p_nr_kat);
  EXECUTE instrukcja;
END;
$$
LANGUAGE 'plpgsql';

CREATE OR REPLACE FUNCTION dynamiczna_tabela
  (IN nazwa_tabeli VARCHAR, IN kol_1 VARCHAR,
   IN kol_2 VARCHAR)
RETURNS VOID AS
$$
DECLARE
  instrukcja text;
BEGIN
  --instrukcja := 'DROP TABLE'
  --  || quote_ident(nazwa_tabeli) ;
  instrukcja := 'CREATE TABLE '
    || quote_ident(nazwa_tabeli) || ' ( '
    || quote_ident(kol_1) || ' integer, '
    || quote_ident(kol_2) || ' VARCHAR(10) '
    || ' ) ';
  EXECUTE instrukcja;

  instrukcja := 'INSERT INTO ' || quote_ident(nazwa_tabeli)
    || ' VALUES (1, ''Pierwszy'' )';
  EXECUTE instrukcja;
END;
$$
LANGUAGE 'plpgsql';
```

Język SQL PL oferuje dwa sposoby na wykonywanie dynamicznego SQL. Pierwszy z nich to instrukcja `EXECUTE IMMEDIATE`, drugi to para instrukcji: `PREPARE` i `EXECUTE`.

Instrukcja `EXECUTE IMMEDIATE` odpowiada łącznemu działaniu instrukcji `PREPARE` i `EXECUTE`. Jest podobna w użyciu do instrukcji `EXECUTE IMMEDIATE` w języku PL/SQL i instrukcji `EXECUTE` w PL/pgSQL. Przyjmuje jeden parametr, typu `VARCHAR`, będący treścią instrukcji SQL. Jej zastosowanie wygląda następująco:

```
CREATE PROCEDURE dynamiczny_ex_imm (OUT stan INT)
SPECIFIC dynamiczny_ex_imm
LANGUAGE SQL
BEGIN
  DECLARE instrukcja VARCHAR(1000);
  SET instrukcja = 'INSERT INTO MAIN.publikacje
    VALUES (2, ''instrukcja'', ''opis'')';
  EXECUTE IMMEDIATE instrukcja;
  GET DIAGNOSTICS stan = ROW_COUNT;
END
@
```

Jeżeli dynamicznie składana instrukcja ma być wywoływana wielokrotnie, wówczas należy zastosować parę instrukcji: `PREPARE` i `EXECUTE`. Podobnie jak poprzednio i w tym przypadku należy rozpocząć od zadeklarowania zmiennej typu `VARCHAR` i przypisania jej treści istrukcji. Ponadto należy zadeklarować zmienną specjalnego typu `STATEMENT`. Instrukcja `PREPARE` przygotowuje instrukcję do wykonania z instrukcji zapisanej jako ciąg znaków w zmiennej `VARCHAR`. Tak powstaje *statement*, który następnie należy przekazać do instrukcji `EXECUTE` w celu wykonania. Można również korzystać z dowiązywania zmiennych za pomocą klauzuli `USING`. Wewnątrz budowanej dynamicznie instrukcji miejsce na dowiązaną zmienną oznacza się znakiem zapytania.

Dynamiczny SQL w SQL PL pozwala zarówno na wykonywanie instrukcji DML jak i DDL.

Poniższy przykład przedstawia wykorzystanie `PREPARE` i `EXECUTE`:

```
CREATE PROCEDURE zmien
  (IN p_nr INT, IN p_tytul VARCHAR(100),
   IN p_autor VARCHAR(100))
SPECIFIC zmien
LANGUAGE SQL
BEGIN
  DECLARE instr VARCHAR(400);
  DECLARE stmt STATEMENT;
```

```
  SET instr = 'UPDATE MAIN.ksiazki
    SET tytul = ?, autor = ? WHERE nr_kat = ?';
  PREPARE stmt FROM instr;
  EXECUTE stmt USING p_tytul, p_autor, p_nr;
END
@
```

Kolejny przykład pokazuje zastosowanie dynamicznego SQL do utworzenia tabeli:

```
CREATE PROCEDURE utworz (IN p_nazwa VARCHAR(10))
SPECIFIC utworz
LANGUAGE SQL
BEGIN
  DECLARE instr VARCHAR(200);
  DECLARE stmt STATEMENT;

  SET instr = 'CREATE TABLE MAIN.'
    || p_nazwa || ' (nr INT, opis VARCHAR(20))';
  PREPARE stmt FROM instr;
  EXECUTE stmt;
END
@
```

Instrukcję `EXECUTE` można również stosować z klauzulą `INTO`, przekazującą wartości do podanych zmiennych. Należy również pamiętać, że nie wszystkie instrukcje SQL mogą być używane w dynamicznym SQL.

Język T-SQL oferuje dwa polecenia pozwalające na wykonywanie dynamicznie konstruowanych instrukcji: `EXEC` (`EXECUTE`) oraz `sp_executesql`.

Polecenie `EXEC` przyjmuje jeden parametr: ujęty w nawiasy łańcuch znakowy. W łańcuchu tym nie mogą wystąpić wywołania funkcji ani wyrażenie `CASE`. `EXEC` nie pozwala też na dynamiczne dowiązywanie zmiennych. Wszystkie potrzebne wartości i zmienne muszą zostać zaszyte w konstruowanej instrukcji.

```
ALTER PROCEDURE dyn_exec
  @nr_kat INT, @tytul VARCHAR(50), @autor VARCHAR(50) AS
BEGIN
  DECLARE @instr VARCHAR(200);
  SET @instr = 'UPDATE dbo.ksiazki SET tytul = ''' +
    @tytul +  ''' , autor = ''' + @autor +
    ''' WHERE nr_katalogowy = ' +
    CONVERT(VARCHAR(20), @nr_kat);
  EXEC (@instr);
END
```

`sp_executesql` to wbudowana procedura serwera, przyjmująca zarówno parametry wejściowe i wyjściowe. Składnia jej wywołania to:

```
EXEC sp_executesql
  @stmt = <innstrukcja>,
  @params = <paramery>,
  <przypisanie_parametrów>
```

Zarówno `@stmt` jak i `@params` muszą być typu `ntext`, `nchar` lub `nvarchar`. `sp_executesql` pozwala na dowiązywanie zmiennych, tak jak pokazano to na przykładzie procedury `dyn_sp_exec`:

```
ALTER PROCEDURE dyn_sp_exec @p_nr_kat INT  AS
BEGIN
  DECLARE @instr NVARCHAR(200);
  SET @instr = 'SELECT * FROM dbo.ksiazki
    WHERE nr_katalogowy = @nr_kat;';
  EXEC sp_executesql
    @stmt = @instr,
    @params = N'@nr_kat AS INT',
    @nr_kat = @p_nr_kat;
END
```

Przykładowe wywołanie:

```
EXEC dyn_sp_exec 4
```

Dynamiczny SQL w Transact-SQL, podobnie jak w poprzednio omówionych językach, umożliwia uruchamianie instrukcji DML oraz DDL.

Rozdział 4

Podsumowanie

4.1 Proceduralny SQL w Oracle, PostgreSQL, IBM DB2 i MS SQL Server

Przeprowadzona analiza porównawcza czterech języków programowania proceduralnego: PL/SQL, PL/pgSQL, SQL PL i T-SQL wykazała zarówno wiele podobieństw, jak i różnic pomiędzy tymi językami. Najwięcej elementów wspólnych można wskazać w obszarze podstawowych właściwości omawianych języków. W miarę zagłębiania się w bardziej zaawansowane mechanizmy można dostrzec więcej różnic. Niektóre z nich są mniej, a inne bardziej istotne. Zaobserwować można zarówno drobne różnice w formie słów kluczowych, nazewnictwie i składzie poszczególnych instrukcji, jak i istotne rozbieżności w sposobie działania mechanizmów poszczególnych języków. Poniżej przedstawiono wnioski z przeprowadzonej analizy.

Jedną z fundamentalnych własności każdego języka jest sposób konstruowania bloków kodu, używania zmiennych i typów danych. W językach PL/SQL i PL/pgSQL struktura bloku jest taka sama. Jest to struktura ściśle uporządkowana. Język SQL PL implementuje już inną, lecz wciąż uporządkowaną strukturę bloku. Język T-SQL dopusza natomiast pełną dowolność w występowaniu poszczególnych elementów w ramach bloku.
Również taki sam jest sposób deklaracji i używania zmiennych w PL/SQL i PL/pgSQL.
Język T-SQL jako jedyny natomiast nie pozwala na przypisanie zmiennej wartości domyślnej oraz nie obsługuje mechanizmu zakresu zasięgu zmiennej. Również jako jedyny pozwala na pominięcie znaku końca instrukcji.

W zakresie sterowania wykonaniem programu niemalże identycznie zachowują się języki PL/SQL i PL/pgSQL. Język SQL PL już nieco od nich odbiega. T-SQL zaś oferuje najmniej możliwości na tym polu, liczba dostępnych instrukcji sterujących jest tu zauważalnie mniejsza w stosunku do pozostałych języków.

Więcej różnic można odnotować w przypadku przetwarzania danych.

Prostym przykładem może być instrukcja `SELECT INTO`. Występuje ona w językach PL/SQL, PL/pgSQL i SQL PL, i przez każdy z tych języków jest trochę inaczej obsługiwana. I choć wydaje się należeć do konstrukcji podstawowych, to jednak w języku T-SQL jej nie odnajdziemy.
Poważniejszym przykładem jest przetwarzanie transakcji. We wszystkich systemach istota transakcji jest taka sama. Inna jest już jednak obsługa transakcji z poziomu kodu proceduralnego SQL. Największe możliwości oferują w tym przypadku Oracle i IBM, najmniejsze PostgreSQL. Ponadto Oracle jako jedyny oferuje mechanizm transakcji autonomicznych.
We wszystkich językach można korzystać z kursorów, a sposób ich obsługi jest dość podobny. Nie wszystkie języki oferują jednak specjalne atrybuty lub funkcje do obsługi kursorów. Pewną osobliwością jest konieczność dealokacji kursora po jego zamknięciu w języku T-SQL.

Poważne różnice pomiędzy omawianymi językami występują w zakresie obsługi tabel i typów tablicowych. PL/SQL oferuje najbardziej złożony system tablic, tak zwanych kolekcji, wraz z pakietem funkcji do ich obsługi. Pozwala na zapis tablic do tabel bazy danych. Znacznie okrojony odpowiednik Oracle'owych kolekcji – tablice – występuje w języku PL/pgSQL, który rówież pozwala na posługiwanie się tablicami zarówno w kodzie proceuralnym, jak i ich zapis do kolumn tabel bazy danych. Tablic programistycznych nie udostępniają niestety już ani SQL PL, ani T-SQL. W przypadku tych języków jako zamiennik można wykorzystać tymczasowe tabele bazy danych. Tabele te są dostępne zarówno w systemach DB2 i MS SQL Server, jak i Oracle i PostgreSQL.

Koncepcyjnie podobny, lecz inaczej zaimplementowany w poszczególnych językach, jest mechanizm obsługi błędów. Wyraźne podobieństwa obserwujemy tu między językami PL/SQL i PL/pgSQL. Oba, na przykład, definiują sekcję obsługi wyjątków `EXCEPTION`, jednak PL/SQL, w przeciwieństwie do PL/pgSQL, pozwala na definiowanie błędów użytkownika. W przypadku korzystania z obsługi wyjątków w PL/pgSQL programista musi ograniczyć się do puli błędów predefiniowanych. Inne jest również działanie instrukcji `RAISE` występującej w obu tych językach. `RAISE` z PL/pgSQL odpowiada PL/SQL-owej instrukcji `RAISE_APPLICATION_ERROR` (a nie `RAISE`).
W strukturze bloku języka SQL PL nie istnieje specjalnie wydzielona część przeznaczona do obsługi wyjątków. Język ten wprowadza, oprócz pojęcia wyjątku, pojęcie uchwytu wyjątku. Deklaracje związane z obsługą błędów muszą tu znaleźć się w odpowiednim miejscu wszystkich deklaracji w bloku. T-SQL proponuje natomist strukturę znaną z innych języków programowania: sekwecję pary bloków `TRY` – `CATCH`. W tej strukturze błędy zaistniałe w bloku `TRY` są obsługiwane w bloku `CATCH`.
Zarówno w SQL PL jak i T-SQL użytkownik ma możliwość deklarowania własnych wyjątków.

W zakresie tworzenia funkcji i procedur składowanych jedną z ważniejszych różnic pomiędzy omawianymi językami jest brak możliwości utworze-

nia procedury w języku PL/pgSQL. W języku tym można tworzyć tylko funkcje, podczas gdy w pozostałych językach zarówno funkcje jak i procedury. Języki PL/SQL i PL/pgSQL do tworzenia i modyfikacji programu składowanego udostępniają instrukcję `CREATE OR REPLACE`, dzięki której nie jest konieczne wcześniejsze usuwanie starej wersji danego podprogramu. Język T-SQL oferuje w jej miejsce parę instrukcji: `CREATE` i `ALTER`, zaś w języku SQL PL podprogram należy usunąć przed utworzeniem jego nowej wersji.
Różnice występują również w organizacji kodu podprogramów. Jako że podprogramy są blokami kodu, ich struktura jest zgodna ze strukturą obowiązującą dla bloku w danym języku. Bloki PL/SQL i PL/pgSQL mają taką samą strukturę, lecz w przypadku programów składowanych w języku PL/SQL nie stosuje się słowa `DECLARE` rozpoczynającego sekcję deklaracji.
W kwestii deklaracji parametrów podprogramów T-SQL jako jedyny nie pozwala na jawne zadeklarowanie parametru w trybie `IN`, zaś w przeciwieństwie do PL/SQL i PL/pgSQL, języki SQL PL i T-SQL pozwalają na modyfikację parametrów wejściowych w ciele podprogramu.
Istotnymi cechami języka PL/SQL, odróżniającymi go od pozostałych języków, jest możliwość deklarowania podprogramów lokalnych w sekcji deklaracji programu składowanego oraz możliwość tworzenia pakietów. Natomiast język PL/pgSQL jako jedyny nie dopuszcza stosowania instrukcji kontroli transakcji wewnątrz programu składowanego.
Języki PL SQL i PL/pgSQL pozwalają na przekazywanie do programu nieograniczonej liczby parametrów. Język SQL PL ogranicza możliwą liczbę parametrów funkcji, podczas gdy liczba parametrów procedury jest nieograniczona. Język T-SQL nakłada natomiast ograniczenia zarówno w liczbie parametrów procedury jak i funkcji.
Różnice można odnotować również w zakresie możliwych typów danych przyjmowanych i zwracanych przez podprogramy składowane oraz w samym sposobie wywoływania programów.
Poza wymienionymi istnieje jeszcze wiele innych, omówionych wcześniej różnic w tworzeniu podprogramów w językach PL/SQL, PL/pgSL, SQL i T-SQL. Najwięcej podobieństw można wskazać pomiędzy językami PL/SQL i PL/pgSQL.

Wszystkie z omówionych w pracy języków umożliwiają tworzenie wyzwalaczy. Jednak nie każdy typ wyzwalacza można utworzyć w każdym języku. Największe możliwości oferuje pod tym względem język PL/SQL. W języku PL/pgSQL nie utworzymy wyzwalacza typu `INSTEAD OF`, a w T-SQL nie istnieją wyzwalacze typu `BEFORE`. Język PL/pgSQL w celu zdefiniowania wyzwalacza wymaga utworzenia dwóch elementów: funkcji, która będzie wywoływana przez wyzwalacz, i wyzwalacza właściwego. W pozostałych językach wyzwalacz definiuje się w jednym kroku. Języki PL/pgSQL i T-SQL nie pozwalają na podanie nazwy kolumny tabeli w zdarzeniu uruchamiającym wyzwalacz oraz na zdefiniowanie warunku `WHEN`. PL/SQL, PL/pgSQL oraz SQL PL udostępniają zmienne `OLD` i `NEW`. Ich odpowiednikiem w T-

SQL są dostępne w ciele wyzwalacza tabele `deleted` i `inserted`. Ponadto PL/SQL udostępnia predykaty `INSERTING`, `DELETING` i `UPDATING`, które pozwalają zidentyfikować operację, która uruchomiła wyzwalacz. W ich miejsce PL/pgSQL wprowadza zmienną `TG_OP`. SQL PL nie oferuje podobnego mechanizmu, a w przypadku T-SQL należy się odwołać do tabel `deleted` i `inserted`. Ważną właściwością wyzwalaczy w języku PL/SQL jest brak możliwości operowania w ciele wyzwalacza na tabeli, dla której wyzwalacz został uruchomiony. W pozostałych językach jest to możliwe. Istotną rzeczą jest rówież możliwość wykonania instrukcji wycofania transakcji w wyzwalaczu w języku T-SQL. Zarówno w PL/SQL jak i PL/pgSQL oraz SQL PL wyzwalacz nie może zawierać instrukcji kontroli transakcji.

W zależności od języka różni się również sposób korzystania z dynamicznego SQL. PL/SQL oferuje dwa sposoby: wbudowany dynamiczny język SQL (NDS) i pakiet DBMS_SQL. Utworzoną instrukcję SQL wykonuje się tu za pomocą `EXECUTE IMMEDIATE`. Jej odpowiednikiem w PL/pgSQL jest instrukcja `EXECUTE`. PL/pgSQL, w przeciwieńsiwie do PL/SQL nie pozwala stosować zmiennych powiązanych. SQL PL do wykonywania dynamicznego SQL udostępnia dwie metody: instrukcję `EXECUTE IMMEDIATE`, która jest podobna do `EXECUTE IMMEDIATE` w PL/SQL i `EXECUTE` w PL/pgSQL oraz parę instrukcji `PREPARE` i `EXECUTE`, służącej do wielokrotnego wywoływania raz utworzonej instrukcji SQL. SQL PL, podobnie jak PL/SQL, pozwala na korzystanie ze zmiennych powiązanych. T-SQL również jest wyposażony w dwa sposoby wykonywania dynamicznych zapytań SQL. Pierwszy z nich to instrukcja `EXECUTE`, która nie pozwala na dynamiczne dowiązywanie zmiennych. Drugim sposobem jest procedura składowana `sp_execute`. Procedura ta pozwala na dowiązywanie zmiennych.

Reasumując, z przeprowadzonej analizy wynika, iż największe podobieństwo do siebie wykazują języki PL/SQL i PL/pgSQL. Pomiędzy wszystkimi językami najwięcej zbieżności pojawia się w ramach właściwości podstawowych. W tym obszarze łatwo wykazać znaczne podobieństwo języków PL/SQL i PL/pgSQL. Wraz z zagłębianiem się w tematykę bardziej zaawansowaną rośnie liczba różnic i wyraźniej zarysowują się indywidualne cechy języków. Mimo to, wciąż PL/SQL i PL/pgSQL wydają się posiadać najwięcej wspólnych elementów.

Natomiast w kwestii przydatności i wygody użycia poszczególnych języków należy stwierdzić, że najłatwiej i najwygodniej jest osiągnąć zamierzony efekt w języku PL/SQL. Wynika to z faktu, że język ten oferuje najbogatszą gamę konstrukcji. Jednocześnie narzuca on programiście bardzo uporządkowaną strukturę, co przekłada się na czytelność i łatwość konserwacji kodu. Nieco trudniej jest osiągnąć rezultaty w językach PL/pgSQL i SQL PL. W porównaniu do PL/SQL funkcjonalość tych języków jest trochę mniejsza. Znacznie mniejsza jest natomiast wygoda programowania w oferowanym przez Microsoft Corporation języku T-SQL. Nie tylko udostępnia on najmniej możliwości programistycznych, ale ponadto brak wymogów odnośnie

uporządkowanej struktury w połączeniu z brakiem wydawałoby się błahych rzeczy, takich jak np. brak słowa kluczowego `THEN`, czy kończącego strukturę `END IF`, często prowadzą do powstawania kodu nieczytelnego i trudnego w późniejszym utrzymaniu.
Podsumowując, należy więc uznać, iż najbardziej efektywnym narzędziem pracy programisty, spośród omawianych tu czterech, jest język PL/SQL.

4.2 Pi of the Sky – wnioski z migracji

Instalacja i konfiguracja serwera bazy danych IBM DB2, wykonana przez innego członka zespołu, przebiegła bez większych zakłóceń. W wyniku analizy struktury istniejących w systemie PostgreSQL baz *Pi of the Sky* została zaprojektowana uniwersalna struktura bazy danych, do której mogą być migrowane wszystkie obecne bazy eksperymentu. Utworzenie struktur bazy (tabel, kluczy, indeksów) w systemie DB2 i przeniesienie danych przebiegło pomyślnie. Ważnym zagadnieniem na tym etapie było uzgodnienie typów danych pomiędzy systemami PostgreSQL a DB2. Pewnych problemów przysporzyły duże tabele. Ułatwieniem podczas migracji danych był natomiast, skądinąd niezgodny ze sztuką projektowania i utrzymania baz danych, brak kluczy obcych w pierwotnych strukturach w systemie PostgreSQL. Szczegóły z tej części projektu można odnaleźć w pracy [13]. Dokument ten zawiera diagram struktury bazy danych utworzonej w systemie DB2 oraz opis zespołu skryptów, które powstały w celu wykonania migracji.

Przeniesienie warstwy oprogramowania serwera (funkcje, procedury, wyzwalacze) również przebiegało dość sprawnie. W trakcie przepisywania podprogramów wyszczególnionych na liście w dodatku A okazało się, że liczba programów do przełożenia z języka PL/pgSQL na SQL PL jest znacznie większa niż wynikałoby to z powyższej listy. Powodem tego było nieumieszczenie na liście programów pośrednich, niezbędnych do działania funkcji wymienionych w zestawieniu. Dlatego też nie wszystkie funkcje z listy zostały przepisane. Jednak liczba programów utworzonych pod DB2 znacznie przewyższyła liczbę tych wymienionych na liście.

W trakcie implementowania funkcji, procedur i wyzwalaczy w języku SQL PL można było zaobserwować rozbieżności w mechanizmach tego języka i języka PL/pgSQL. Różnice dotyczyły zarówno podstawowych elementów takich jak typy danych czy struktura bloku, jak i właściwości bardziej zaawansowanych, takich jak obsługa transakcji z poziomu kodu proceduralnego SQL, mechanizmu przechwytywania i obsługi wyjątków, stosowania tablic programistycznych oraz samego sposobu definiowania podprogramów. Problemem okazał się brak instrukcji wyprowadzającej komunikaty na ekran z procedury w języku SQL PL. Funkcje zaimplementowane w ramach projektu *Pi of the Sky* w języku PL/pgSQL często wykorzystywały ten mechanizm do informowania operatora o postępach w wykonywaniu obliczeń na danych

zgromadzonych w bazie. Udało się zaimplementować podobny mechanizm w języku SQL PL. Nie jest on jednak całkowicie spójny z mechanizmem komunikatów w PL/pgSQL. Ze względu na fakt, że DB2 nie umożliwia wysyłania komunikatów na ekran w trakcie wykonywania procedury, a dopiero po jej zakończeniu, niemożliwe jest informowanie użytkownika na bieżąco o postępach w przetwarzaniu danych. Możliwe jest natomiast wyprowadzenie informacji na ekran po zakończeniu wykonywania podprogramu. Informacja jest budowana w trakcie wykonywania procedury, a następnie cała wyświetlana użytkownikowi na ekranie. Taki sposób przekazywania komunikatu sprawia, że niecelowe staje się informowanie operatora o procentowym przyroście wykonanych obliczeń. Po zakończeniu działania procedury informacje te przestają być wartościowe.

Poza problemem z wyświetlaniem komunikatów, który z punktu widzenia całego projektu nie był zbyt istotnym zagadnieniem, nie napotkano innych, które uniemożliwiałyby przeniesienie warstwy oprogramowania z serwera PostgreSQL i języka PL/pgSQL na serwer DB2 i towarzyszący mu język SQL PL.

Podsumowując przebieg migracji baz eksperymentu *Pi of the Sky* z systemu zarządzania bazami danych PostgreSQL do systemu DB2 należy zauważyć, że migrację tę byłoby łatwiej przeprowadzić jeśli systemem docelowym byłby system firmy Oracle. Zarówno serwer Oracle jak i język proceduralny PL/SQL mają więcej cech wspólnych z PostgreSQL i jego językiem PL/pgSQL niż DB2 i związany z nim SQL PL. Podobieństwa systemów PostgreSQL i Oracle wynikają z faktu, iż twórcy PostgreSQL i PL/pgSQL dokładają starań, by budowany przez nich system był zbieżny z mechanizmami działającymi w Oracle. Dzięki temu, choć nie są identyczne, systemy PostgreSQL i Oracle posiadają wiele wspólnych właściwości. W przypadku przenoszenia warstwy oprogramowania z serwera PostgreSQL na serwer Oracle wiele podprogramów napisanych w języku PL/pgSQL można uruchomić po niewielkich poprawkach na serwerze Oracle. Uruchomienie tych samych programów na serwerze DB2 wymaga wykonania już dużo większych zmian w ich kodzie.

Bibliografia

[1] Oracle Corporation: *Oracle Database PL/SQL Language Reference 11g Release 1 (11.1)*. http://download.oracle.com/docs/cd/B28359_01/appdev.111/b28370.pdf oraz http://download.oracle.com/docs/cd/B28359_01/appdev.111/b28370/toc.htm

[2] Oracle Corporation: *Oracle Database SQL Language Reference 11g 11g Release 1 (11.1)*. http://download.oracle.com/docs/cd/B28359_01/server.111/b28286.pdf oraz http://download.oracle.com/docs/cd/B28359_01/server.111/b28286/toc.htm

[3] The PostgreSQL Global Development Group: *PostgreSQL 8.2 Documentation*. http://www.postgresql.org/docs/manuals/

[4] The PostgreSQL Global Development Group: *PostgreSQL 8.3 Documentation*. http://www.postgresql.org/docs/manuals/

[5] International Business Machines Corporation (IBM): Information center. http://publib.boulder.ibm.com/infocenter/dzichelp/v2r2/index.jsp?topic=/com.ibm.qmf.doc.iandm/dsqf2mst361.htm lub http://publib.boulder.ibm.com/infocenter/db2luw/v8/index.jsp?topic=/com.ibm.db2.udb.doc/ad/c0011916.htm

[6] Paul Yip, Drew Bradstock, Hana Curtis, Michael X. Gao, Zamil Janmohamed, Clara Liu, Fraser McArthur: *DB2 SQL Procedural Language for Linux, UNIX, and Windows*. Prentice Hall Professional Technical Reference 2002, ISBN: 0-13-100772-6.

[7] International Business Machines Corporation (IBM): *DB2 Version 9.5 for Linux, UNIX, and Windows SQL Reference Volume 1*. http://publibfp.boulder.ibm.com/epubs/pdf/c2358611.pdf

[8] International Business Machines Corporation (IBM): *DB2 Version 9.5 for Linux, UNIX, and Windows SQL Reference Volume 2*. http://publibfp.boulder.ibm.com/epubs/pdf/c2358621.pdf

[9] International Business Machines Corporation (IBM): *IBM DB2 Universal Database Komunikaty, tom 1*. http://publibfp.boulder.ibm.com/epubs/pdf/c8500611.pdf

[10] International Business Machines Corporation (IBM): *IBM DB2 Universal Database Komunikaty, tom 2*. http://publibfp.boulder.ibm.com/epubs/pdf/c8500621.pdf

[11] Microsoft Corporation: Microsoft Developer Network. http://msdn.microsoft.com/en-us/library/default.aspx

[12] Microsoft Corporation: SQL Server 2005 Books Online.

[13] Michał Ziółkowski: *Migracja do systemu DB2 i optymalizacja bazy danych eksperymentu „Π o the Sky"*. Uniwersytet Warszawski, Wydział Matematyki, Informatyki i Mechaniki. Praca dyplomowa, wrzesień 2008.

Dodatek A

Pi of the Sky: Migracja procedur

Założenia migracji
Autor: Marcin Sokołowski

Celem projektu jest migracja najbardziej potrzebnych procedur składowych napisanych w pg/SQL (dla bazy danych PostgreSQL) na język procedur składowych bazy danych DB2 (pl/SQL).

Pliki SQL z treścią procedur dla bazy danych PostgreSQL są dostępne w pliku proc.tar.gz. Dostępnym przez WWW:
http://grb.fuw.edu.pl/pi/user/msok/src/pg_sql/proc.tar.gz

Nie wszystkie z procedur zawartych w tych plikach wymagają migracji, poniżej załączam listę tych które trzeba zrobić w pierwszej kolejności. Być może trzeba będzie dopisać jeszcze kilka innych.

1. Z pliku create_func.sql, następujące procedury:
   - CreateSuperStar (oraz wszystkie funkcje których ona używa)
   - UpdateCatalogRange
   - FillSuperStarNew
   - ReCalcStarStat
   - ReCalcNight
   - ReCalcNights
   - FixStarLinks
   - CreateStarLink
   - CalcBestField
   - FillObsStat

- FindCloseStars
- FindCloseToStar
- MatchIoToPiStars
- MatchIoToPiStarsFAST
- UpdateLastObsIO
- GetStarName
- GetRejReasonDesc
- CleanBad
- RecalcSuperStar
- IsHotPixel
- ReCalcNobs
- ReCalcSpecifiedStars

2. Z pliku pidb_func.sql:

   - UpdateFieldStat
   - UpdateFrameStat
   - trigger_on_new_frame
   - CalcDistInRad
   - CalcDistRaDEC_ARCSEC
   - CalcDistRADEC
   - GetMeanMagDiff
   - GetRmsMagDiff
   - recalc_night_stat
   - update_night_stat
   - avg_ra_corr24
   - get_cat_err_desc
   - AddHotPixel
   - CheckClouds

3. Z pliku pidb_ast.sql:

   - wszystko, trudność tutaj polega na tym, że funkcje te wykorzystują bibliotekę C libastsql.a, jest ona dostępna pod: http://grb.fuw.edu.pl/pi/user/msok/src/pg_sql/astsql.tar.gz