Część II – Język T-SQL

język proceduralny serwera MS SQL Server

Spis treści

I. Transact SQL (T-SQL) – podstawowe konstrukcje 4
1. Wiadomości ogólne 4
2. Blok anonimowy 4
3. Odczytanie wyników działania instrukcji 5
4. Komentarze 5
5. Zmienne 6
6. Zmienne systemowe 7
7. Podstawienie wartości na zmienną. Instrukcja SELECT i SET w T-SQL 7
8. Możliwość odczytu nieprawidłowych danych 8
9. Instrukcje sterujące w T-SQL 9
a) Instrukcja warunkowa IF 9
b) Instrukcja IF EXISTS 11
c) Instrukcja pętli WHILE 12
10. Kursory 13
a) Koncepcja kursora 13
b) Składnia kursora 14
c) Scroll cursor 17
II. Transact SQL (T-SQL) – procedury i wyzwalacze 18
1. Procedury składowane 18
a) Koncepcja procedur składowanych 18
b) Składnia procedury 18
2. Zwracanie wyników przez procedury T-SQL 20
a) Result Set 20
b) Parametr OUTPUT 21
c) RETURN 22
3. Podsumowanie procedur 22
4. Wyzwalacze (triggery) 23
a) Koncepcja wyzwalaczy 23
b) Wyzwalacze uruchamiane poleceniami DML 23
c) Składnia wyzwalacza 24
d) Działanie wyzwalaczy 24
e) Kilka wyzwalaczy na jednej tabeli 25
f) Sekwencyjne uruchamianie wyzwalaczy 25
g) Kontrola zmian dokonywanych przez wyzwalacz 25
h) Wyzwalacze INSTEAD OF 27

i) Podsumowanie wyzwalaczy 28
5. Obsługa błędów 29
a) Instrukcja `RAISERROR` 29
a) Instrukcja THROW 31
b) Podsumowanie obsługi błędów 32
III. Transact SQL (T-SQL) – konstrukcje (nieco bardziej) zaawansowane 33
1. Skrótowy zapis operacji algebraicznych 33
2. Funkcje “rankingowe” w MS SQL Server 34
a) Funkcja NTILE 34
b) Funkcja ROW_NUMBER 34
c) Funkcja RANK 36
d) Funkcja DENSE_RANK 36
3. Tabele tymczasowe w MS SQL Server 37
4. Zmienne tabelaryczne T-SQL 38
5. Alternatywa dla podzapytania w UPDATE i DELETE 40
6. Dostęp do zmienianych danych w T-SQL 41
7. CASE (przypomnienie) 43
8. Skorelowany UPDATE (przypomnienie) 44
9. Common Table Expression z rekursją 45
10. MERGE 47
a) Koncepcja MERGE 47
b) Składnia MERGE 48
c) Funkcje poszczególnych klauzul 48
11. Funkcje T-SQL 51
12. Dynamiczny SQL 55

I. Transact SQL (T-SQL) – podstawowe konstrukcje

1. Wiadomości ogólne

Jako pierwszy zostanie omówiony język implementowany przez serwer firmy Microsoft. Język ten został stworzony przez firmę SYBASE i wbudowany do serwerów tej firmy, później prawa do jego wykorzystania kupiła firma Microsoft, istotnie go rozwijając. Obecnie jest wbudowywany w kolejnych wersjach serwera MS SQL i wraz z nimi nadal rozwijany. W dalszej części wykładu będzie zasadniczo omawiana wersja MS SQL Server 2012[1].

T-SQL posiada dość prostą składnię, niemniej umożliwiającą realizację wszystkich niezbędnych zadań po stronie serwera bazy danych. Kod wykonywany jest bezpośrednio na serwerze.

Po stronie serwera bazy danych powinny być wykonywane wszystkie operacje związane ze składowaniem danych, centralną kontrolą spójności i bezpieczeństwa danych. Znaczna część tych zadań może zostać zrealizowana wprost przez polecenia języka SQL. T-SQL pozwala rozszerzyć te zadania, a pojedyncze instrukcje łączyć w większe struktury, które mogą być przechowywane jako obiekty bazy danych:

- procedury składowane
- wyzwalacze

2. Blok anonimowy

Podstawową konstrukcją programistyczną w językach proceduralnych baz danych jest tzw. blok anonimowy. Pojęcie bloku anonimowego jest sformalizowane w języku ORACLE Pl/SQL, w MS SQL Server nazwa ta jest pominięta.

Pod pojęciem bloku anonimowego należy rozumieć szereg instrukcji języka proceduralnego wraz z instrukcjami SQL, które są tworzone bezpośrednio w środowiskach developerskich (Management Studio, Azure Data Studio Sqldeveloper, SQL+, DataGrip), a następnie wykonywane przez interpreter serwera bazy danych, ale nie są zapisywane jako obiekty bazy danych (procedury, funkcje, wyzwalacze).

Tego typu programy mogą być zapisywane w postaci skryptów – plików tekstowych zawierających kod możliwy do ponownego (wielokrotnego) użycia, ale przechowywanych poza bazą danych (pliki .sql lub .txt).

Zgodnie z wymogami standardu języka SQL, wszystkie słowa kluczowe, nazwy obiektów i zmiennych są Not Case Sensitive – mogą być pisane z dowolnym użyciem małych i dużych liter. Dotyczy to zarówno składni SQL jak i T-SQL.

Konstrukcja bloku anonimowego w T-SQL jest praktycznie niesformalizowana, bardzo prosta (może nawet za prosta – pozwalająca na nieporządne pisanie kodu). Instrukcje mogą być pisane w jednej linii, linie mogą być łamane w dowolnym miejscu bez użycia jakiegokolwiek

[1] Zmiany i nowe rozwiązania pojawiające się w kolejnych wersjach MS SQL Server mogą zostać pominięte na tym etapie poznawania tematu.

znacznika. Jednak zdecydowanie zaleca się wprowadzanie pewnego porządku – umieszczanie kolejnych linii jedna pod drugą, kończenie poleceń średnikami (choć MS SQL Server tego nie wymaga), stosowanie wcięć.

W T-SQL każdy blok powinien być kończony słowem **GO**. W praktyce słowo **GO** jest na ogół pomijane. Management studio uzupełnia nim (niejawnie) każdy blok, jeśli zostało ono pominięte. Jawne wprowadzanie **GO** staje się istotne, gdy chcemy uruchomić kilka kolejnych, niezależnych od siebie programów.

Każda zadeklarowana lokalnie zmienna „żyje" do wystąpienia najbliższego słowa **GO**.

3. Odczytanie wyników działania instrukcji

Wynik działania instrukcji SELECT jest wypisywany na pulpit aplikacji MS Management Studio w postaci zestawu odczytanych rekordów. Jest to tzw. Result set.

W przypadku poprawnego wykonania instrukcji DML wyświetlany jest komunikat o liczbie wierszy biorących udział w operacji np. *5 row(s) affected*. Komunikat ten jest też przekazywany do aplikacji, która uruchomiła polecenie. Ten mechanizm może zostać wyłączony poleceniem `SET NOCOUNT ON` (i ponownie włączony `SET NOCOUNT OFF`).

Poleceniem wypisania komunikatu jest `PRINT`, np.

```
PRINT 'Ala ma ' + Cast(2+2 As Varchar) + ' koty.';
```

Komunikat pojawia się na zakładce *Messages*.

Jeżeli zastąpimy polecenie PRINT przez SELECT

```
SELECT 'Ala ma ' + Cast(2+2 As Varchar) + ' koty.';
```

Wynik pojawi się na zakładce *Results* i będzie traktowany jako wynik instrukcji **SELECT**.

4. Komentarze

Część kodu nie przeznaczona do wykonywania (wyłączona), nazywana jest komentarzem. W trakcie wykonywania programu będzie ona ignorowana przez interpreter. W obu środowiskach (ORACLE i MS SQL) komentarz oznaczany jest jednakowo.

Komentarz blokowy, dowolna liczba linii, pomiędzy nawiasami:

```
/* te wiersze są zakomentowane
i nie będą brane pod uwagę w trakcie
wykonywania programu*/
```

Komentarz jednoliniowy – do końca linii:

```
-- a to są zakomentowane dwa  pojedyncze wiersze,
-- one też zostaną pominięte przy wykonywaniu obliczeń

SELECT * FROM emp; -- można też komentować część wiersza
```

5. Zmienne

Zmienna jest jednym z podstawowych elementów (konstrukcji) niezbędnych przy programowaniu kodu. Jej rolą jest przechowywanie wartości (danych) określonego typu. Deklaracja zmiennej to przekazanie do serwera bazy danych informacji o konieczności zarezerwowania w pamięci RAM obszaru o rozmiarze odpowiadającym typowi deklarowanej zmiennej. Nazwa zmiennej to jednocześnie jej identyfikator (musi być unikalna w obszarze swojego działania), jak też pośrednio wskaźnik do miejsca jej przechowywania w pamięci.

W języku T-SQL użycie każdej zmiennej musi zostać poprzedzone jej deklaracją, zawierającą nazwę zmiennej (unikalną w obszarze działania zmiennej) oraz jej typ, zgodny z typami implementowanymi przez serwer. Nazwa każdej zmiennej musi rozpoczynać się znakiem @. W celu zadeklarowania zmiennej używane jest polecenie **DECLARE** a po nim nazwa zmiennej i jej typ:

```
DECLARE @Ile Int;
```

Typy zmiennych – takie same jak w przypadku poleceń języka SQL w poleceniach tworzenia tabel – zostaną wymienione na końcu wykładu. Można w uproszczeniu przyjąć, że mamy do czynienia z typami napisowymi (`Char`, `Varchar`), liczbami stałoprzecinkowymi (`Inetger`), zmiennoprzecinkowymi (`Decimal`, `Money`), daty i czasu (`Date`, `Time`, `Datetime`).

W pojedynczej instrukcji **DECLARE** można zadeklarować wiele zmiennych:

```
DECLARE @nazwisko Varchar(30), @imie Varchar(20),
        @Data_urodzenia Date;
```

Instrukcja **DECLARE** może pojawiać się wielokrotnie, w dowolnym miejscu kodu (bloku). Jedynym wymogiem jest, aby deklaracja zmiennej poprzedzała jej użycie (odwołanie się do niej). Ta cecha z jednej strony ułatwia pisanie kodu (a nawet może sprzyjać jego optymalizacji), z drugiej pozwala na wprowadzanie pewnego chaosu w kodzie.

Wraz z deklaracją zmiennej można nadać jej wartość początkować (zainicjalizować zmienną).

```
DECLARE @Cena Money = 1500
        ,@Kupujacy Varchar(30) = 'PJWSTK'
        ,@Data_sprzedazy Date = Getdate();
```

W powyższym przykładzie wszystkim zmiennym w trakcie ich deklarowania zostaną nadane wartości początkowe, które w dalszym procesie wykonywania obliczeń mogą zostać zmienione.

Wartość podstawiana na zmienną w trakcie jej deklaracji może być wynikiem wykonania instrukcji **SELECT**

```
DECLARE @Ile Int = (SELECT COUNT(1) FROM EMP);
```

Wartości zmiennym nie muszą być nadawane wraz z ich deklaracją. W T-SQL wszystkie niezainicjalizowane zmienne mają wartość NULL, niezależnie od typu danych.

6. Zmienne systemowe

Zmienne implementowane przez MS SQL Server (używana jest też nazwa „funkcje statyczne") służą do przekazania (odczytania z nich) istotnych informacji przechowywanych lub wyliczanych przez serwer bazy danych. Ich nazwy zaczynają się od znaku @@. Nie należy ich deklarować, służą wyłącznie do odczytu (są *Read Only*).

Kilka najczęściej używanych:

`@@Version` - zwraca informację o wersji używanego serwera bazy danych,

`@@Identity` – ostatnio wygenerowana wartość w autonumerowanej kolumnie,

`@@Error` – numer ostatniego błędu,

`@@Rowcount` – liczba rekordów, na których operowała ostatnia instrukcja SQL; w przypadku **`SELECT`** – liczba zwracanych przez instrukcję rekordów,

`@@Fetch_status` – zwraca informację o podstawieniu rekordu w ostatniej instrukcji **`FETCH`** kursora (0 sukces, rekord podstawiony, != 0 porażka, rekord niepodstawiony). Pełne wyjaśnienie pojawi się przy omawianiu kursorów.

Wyświetlenie (odczyt) wartości zmiennych systemowych obywa się przy użyciu instrukcji **`PRINT`** lub **`SELECT`** – tak jak w przypadku każdej zmiennej.

MS SQL Server implementuje wiele zmiennych systemowych. Pełną informację można znaleźć w dokumentacji serwera.

7. Podstawienie wartości na zmienną. Instrukcja SELECT i SET w T-SQL

Instrukcja **`SELECT`** w bloku T-SQL ma rozszerzone funkcje w stosunku do „czystego" SQL. Może zostać użyta do podstawienia wartości na zmienną lub kilka zmiennych. Nie jest wtedy używana klauzula **`FROM`** ani żadna inna klauzula SQL.

```
DECLARE @Imie Varchar(20), @Nazwisko Varchar(50),
        @DataRekrutacji Date;
SELECT  @Imie = 'Jan', @Nazwisko = 'Kowalski';
```

Instrukcją pozwalającą podstawić wartość na zmienną jest także instrukcja **`SET`**. Jednak pozwala ona na podstawienie tylko jednej wartości.

```
SET @DataRekrutacji = GETDATE();
```

Instrukcja **`SELECT`** może służyć do podstawienia na zmienne wartości odczytanych z bazy danych, zgodnie ze wszystkimi zasadami działania tej instrukcji.

```
SELECT  @ename = ename
        ,@job = job
        ,@Hiredate = hiredate
FROM    emp
WHERE   empno = 7788;

SELECT  @ename ,@job ,@Hiredate;
```

Jeżeli w tabeli istnieje rekord spełniający warunek w klauzuli **WHERE** (a tak jest w pokazanym przykładzie), na zmienne zostaną podstawione wartości odczytane z tego rekordu. Kolejne polecenie **SELECT** spowoduje wypisanie odczytanych wartości na pulpit.

W przypadku takiego jak na powyższym przykładzie użycia instrukcji **SELECT**, powinna ona zwracać jeden wiersz. Jeśli jednak zwróci więcej niż jeden wiersz, na zmienne zostaną podstawione wartości z ostatniego odczytanego wiersza!

Przykład II.6.1

Zadeklaruj zmienną, a następnie na tą zmienną przypisz liczbę pracowników (rekordów) z tabeli Emp. Wypisz komunikat z informacją o liczbie odczytanych rekordów.

```
DECLARE @Ilu Int, @Info Varchar(30);
SELECT  @Ilu = COUNT(1)
FROM    EMP;
Print   'W tabeli EMP zapisanych jest ' + Cast(@Ilu AS Varchar) +
        ' pracowników';
```

Pracując z MS SQL Server należy pamiętać o konieczności dokonywania jawnej konwersji danych na typ napisowy przy konkatenacji (funkcje `Cast` i `Convert`).

Przykład II.6.2

Usuń dział o podanej nazwie, ustawiając wartość Deptno pracowników tego działu na NULL. Wypisz komunikat informujący o liczbie zmodyfikowanych rekordów.

```
DECLARE @Deptno Int, @Info Varchar(50), @Ile Int;
SELECT  @Deptno = Deptno
FROM    DEPT
WHERE   DNAME = 'OPERATIONS';
UPDATE  EMP SET DEPTNO = NULL WHERE DEPTNO = @Deptno;
SET @Ile = @@ROWCOUNT;
DELETE FROM DEPT WHERE DEPTNO = @Deptno;
Print   'Usunięto Departament OPERATIONS i zmodyfikowano
        wartość Deptno w rekordach ' + Cast(@Ile AS Varchar)       +
        ' pracowników';
```

8. Możliwość odczytu nieprawidłowych danych

Istnieją przypadki, w których prawidłowe składniowo użycie podstawienia na zmienne, może spowodować odczytanie nieprawidłowych wartości.

Przykład II.7.1

```
SELECT @ename = ename
       ,@job = job
       ,@Hiredate = hiredate
FROM   emp;

SELECT @ename ,@job ,@Hiredate;
```

W tabeli *Emp* jest 14 rekordów a każda ze zmiennych może przyjąć wartości tylko jednego rekordu. Mimo to polecenie jest wykonywane „prawidłowo” (nie jest zgłaszany błąd), a na zmiennych pojawiają się wartości z ostatniego odczytanego rekordu.

Jeżeli zapytanie nic nie zwróci (wskazany rekord nie zostanie odnaleziony), podstawienie nie zostanie zrealizowane i na zmiennych pozostaną dotychczasowe wartości. Będzie to NULL, jeśli zmienna dotychczas nie była używana, albo wartości odczytane w poprzednim podstawieniu. Prezentuj to kolejny przykład.

Przykład II.7.2

```
SELECT  @ename = ename
        ,@job = job
        ,@Hiredate = hiredate
FROM    emp
WHERE   empno = 7788;

SELECT  @ename ,@job ,@Hiredate;

SELECT  @ename = ename
        ,@job = job
        ,@Hiredate = hiredate
FROM    emp
WHERE   empno = 9999;

SELECT  @ename ,@job ,@Hiredate;
```

Wiersz identyfikowany wartością *empno* = 9999 w tabeli *Emp* nie istnieje, a pomimo tego na zmiennych pozostają wartości odczytane z poprzedniego polecenia SELECT dotyczącego wiersza istniejącego (*empno* = 7788).

W tym przypadkach SQL Server również nie podnosi wyjątku (!), co może być przyczyną istotnych błędów!

9. Instrukcje sterujące w T-SQL

W dotychczasowych rozważaniach i przykładach zakładaliśmy wykonanie wszystkich instrukcji programu po kolei – od pierwszej do ostatniej. Ale taka sytuacja nie zawsze jest możliwa. W bardzo wielu przypadkach zachodzi konieczność podejmowania decyzji w zależności od wyników bieżących obliczeń, powtarzania wielokrotnego tych samych operacji, reagowania na błędy.

Do realizacji tych zadań stosowane są instrukcje sterujące. Ich zestaw w T-SQL jest dość ubogi, sprowadza się do instrukcji warunkowej **IF** i instrukcji pętli **WHILE**.

a) Instrukcja warunkowa IF

Instrukcja warunkowa **IF** jest podstawową instrukcją w większości języków programowania. Pozwala przetestować warunek logiczny i uzależnić od wyniku tego testu przebieg dalszych obliczeń. Składnia tej instrukcji w T-SQL wygląda następująco:

```
IF warunek
     Ciąg instrukcji 1
[ELSE
     Ciąg instrukcji 2]
```

Jeżeli warunek określony po **IF** będzie spełniony (przyjmie wartość **TRUE**) zostanie wykonany `Ciąg instrukcji 1`. Jeśli nie (przyjmie wartość **FALSE** lub **NULL**) zostanie wykonany `Ciąg instrukcji 2`.

Instrukcja **IF** nie musi zawierać klauzuli **ELSE**. W takim przypadku, przy spełnionym warunku, wykonana się Ciąg instrukcji 2, a w przypadku jego niespełnienia program będzie wykonywany dalej, począwszy od pierwszej instrukcji po całym bloku **IF**.

Warunek logiczny zdefiniowany po słowie **IF** może być złożony z wielu warunków elementarnych, połączonych operatorami. Dopuszczalne są wszystkie operatory, które mogły być użyte przy budowaniu warunków w klauzuli **WHERE** instrukcji **SELECT**. Możemy zatem używać operatorów logicznych (**NOT**, **AND**, **OR**), nawiasów, algebraicznych operatorów porównań oraz innych znanych operatorów (**LIKE**, **IN**, **BETWEEN**).

Ciągi instrukcji powinny zostać umieszczone pomiędzy słowami **BEGIN** i **END**. Słowa te mogą zostać pominięte, jeśli wykonywana ma być pojedyncza instrukcja. Jednak dobrym zwyczajem jest nieopuszczanie tych słów, nawet wtedy, kiedy jest to dopuszczalne.

Przykład II.8.1

Sprawdź, czy planowana podwyżka o 10% zarobków wszystkich pracowników (tabela Emp) nie przekracza dysponowanego budżetu 35000. W instrukcji warunkowej uzależnimy dalsze działanie od wyniku tego testu.

```
DECLARE @Info Varchar(50), @Budzet Money;
SELECT  @Budzet = SUM(sal)*1.1
FROM    EMP;

IF @Budzet < 35000
    BEGIN
        UPDATE EMP SET SAL = SAL * 1.1;
        SET @Info = 'Dokonano podwyżki o 10%';
    END;
ELSE
    SET @Info = 'Nie dokonano podwyżki, żeby nie przekroczyć budżetu.';
PRINT @Info;
```

Jak widać na przykładzie *II.8.1*, dwie instrukcje występujące w bloku **IF** zostały ujęte w klamry **BEGIN** i **END**, natomiast pojedyncza instrukcja w bloku **ELSE** już tego nie wymagała, zatem słowa te zostały pominięte. Jeśli warunek jest spełniony, wykonywane są dwie instrukcje (**UPDATE** i **SET**), jeśli nie – tylko jedna (**SET**).

W T-SQL istnieje możliwość umieszczania zapytań **SELECT** wewnątrz warunku logicznego. Pozwala to skrócić kod, zrezygnować z użycia części zmiennych. Przykład *II.8.2* pokazuje kod z przykładu *II.8.1* tak właśnie zmodyfikowany.

Przykład II.8.2

*Modyfikacja przykładu II.8.1 – umieszczenie instrukcji **SELECT** w obszarze deklaracji warunku **IF**.*

```
DECLARE @Info Varchar(50);
IF (SELECT SUM(sal)*1.1 FROM EMP)< 35000
    BEGIN
```

```
       UPDATE EMP SET SAL = SAL * 1.1;
       SET @Info = 'Dokonano podwyżki o 10%';
    END;
ELSE
       SET @Info = 'Nie dokonano podwyżki, żeby nie przekroczyć budżetu.';
PRINT @Info;
```

W obu przykładach proszę zwrócić uwagę na stosowanie wcięć, mających na celu poprawę czytelności kodu, a także na miejsca umieszczenia średników kończących pojedyncze instrukcje.

Niektóre języki programowania oferują konstrukcję ELSEIF, pozwalająca na zdefiniowanie w jednym bloku instrukcji warunkowej więcej niż jednego warunku logicznego, a tym samym rozdzielenia sterowania programem na więcej niż dwie ścieżki.

Konstrukcja taka nie jest dostępna w T-SQL, ale może zostać zastąpiona kilkoma blokami zagnieżdżonymi **IF** … **ELSE**. Pokazuje to kolejny przykład.

Przykład II.8.3

Kolejna modyfikacja przykładu II.8.1. Tym razem sprawdzamy sumę płac po planowanych podwyżkach i w zależności od wyniku podnosimy płacę albo wszystkim pracownikom, albo tylko najmniej zarabiającym, albo nikomu.

```
DECLARE @Info Varchar(50), @Budzet Money, @Ilu Int;
SELECT  @Budzet = SUM(sal) FROM EMP;
IF @Budzet*1.1 < 30000
   BEGIN
       UPDATE EMP SET SAL = SAL * 1.1;
       SET @Info = 'Wszystkim podniesiono place o 10%';
   END;
ELSE
    IF @Budzet BETWEEN 28000 AND 30000
       BEGIN
          UPDATE EMP SET SAL = SAL * 1.1 WHERE SAL < 1200;
          SET @Info = 'Podniesiono płace' + CAST(@@Rowcount AS Varchar) +
                      ' najmniej zarabiającym pracownikom.'
       END;
    ELSE
       SET @Info = 'Nie podwyższono płac, bo nie ma z czego.';
PRINT @Info;
```

b) Instrukcja IF EXISTS

W T-SQL została zaimplementowana bardzo wygodna instrukcja, będąca modyfikacją instrukcji warunkowej, testująca czy polecenia **SELECT** zwraca jakąkolwiek wartość, czyli czy istnieje choć jeden wiersz zwracany przez tę instrukcję.

IF [**NOT**] **EXISTS** (*dowolna instrukcja* **SELECT**)

Jeżeli instrukcja **SELECT** odnajduje choć jeden wiersz i zwraca cokolwiek (nawet **NULL**), cała instrukcja zwraca **TRUE**.

Przykład II.8.4

Sprawdź, czy istnieją pracownicy nie przypisani do żadnego departamentu.

```
IF EXISTS (SELECT 1 FROM EMP WHERE Deptno IS NULL)
PRINT 'Istnieje co najmniej jeden pracownik nie przydzielony do działu.'
```

Instrukcja ta jest bardzo wydajna, gdyż na ogół nie wymaga odczytywania danych z dysku, albo wymaga odczytu do „pierwszego trafienia".

Jeżeli użyjemy zaprzeczenia **IF NOT EXISTS** instrukcja zwróci **TRUE**, jeżeli nie istnieje choć jeden wiersz poszukiwany poleceniem **SELECT**.

W „czystym" SQL **IF EXISTS** może wystąpić tylko wewnątrz instrukcji **SELECT**. W przedstawionej powyżej składni może zostać użyta **tylko w T-SQL**.

UWAGA 1.

IF EXISTS sprawdza istnienie rekordu (co najmniej jednego) spełniającego warunek zdefiniowany w klauzuli **WHERE**, a nie wartości podanej w klauzuli **SELECT** – stąd w przykładzie użyta jest jedynka (czyli cokolwiek). Zatem polecenie

```
IF EXISTS (SELECT comm FROM EMP WHERE empno = 7788)
```

zwróci TRUE jeżeli istnieje wiersz identyfikowany wartością empno = 7788 bez względu na to, jaka jest w tym wierszu wartość w kolumnie comm. Może być ona dowolna, również NULL.

UWAGA 2.

Sekwencja poleceń

```
DECLARE @empno Int;
IF EXISTS (SELECT 1 FROM emp WHERE ename = 'Pink')
      SELECT @empno = empno FROM emp WHERE ename = 'Pink';
ELSE
      INSERT INTO emp (empno, ename)
      SELECT Isnull(Max(empno), 0) + 1, 'Pink'
      FROM emp;
```

jest "delikatnie mówiąc" pozbawiona sensu, gdyż zmusza niepotrzebnie serwer do dwukrotnego odczytu z tego samego wiersza tabeli i powinna być zastąpiona przez sekwencję

```
DECLARE @empno Int;
SELECT @empno = empno FROM emp WHERE ename = 'Pink';
IF @empno IS NULL
      INSERT INTO emp (empno, ename)
      SELECT Isnull(Max(empno), 0) + 1, 'Pink'
      FROM emp;
```

c) Instrukcja pętli WHILE

Bardzo częstym przypadkiem w programowaniu jest konieczność wielokrotnego wykonania tej samej sekwencji instrukcji (poleceń). Struktury programistyczne przeznaczone do realizacji zadań tego typu nazywane są pętlami. Języki programowania na ogół przewidują dwa rodzaje pętli – pętlę wykonywaną aż do spełnienia założonego warunku i pętlę wykonywaną określoną liczbę razy.

Transact SQL implementuje tylko pierwszy rodzaj pętli. Pętla **WHILE** wykonywana jest tak długo, aż warunek logiczny zdefiniowany w deklaracji pętli przestanie mieć wartość **TRUE** i osiągnie **FALSE** lub **NULL**.

Brak implementacji drugiego typu pętli nie stanowi problemu, gdyż korzystając z konstrukcji pętli **WHILE** jesteśmy w stanie zrealizować każdy rodzaj wielokrotnego powtórzenia tej samej sekwencji kodu.

Pętla **WHILE** posiada następującą składnię:

```
WHILE warunek
   Ciąg instrukcji
```

Podobnie jak w przypadku instrukcji **IF** ciąg instrukcji powinien być poprzedzony słowem **BEGIN** i zakończony słowem **END**. Wymóg ten może zostać pominięty, jeśli w pętli ma być wykonywana tylko jedna instrukcja, tak jak to ma miejsce w przypadku instrukcji warunkowej **IF**.

Pisząc kod zawierający instrukcję pętli, musimy się upewnić, że warunek logiczny zdefiniowany w deklaracji pętli kiedyś przestanie być spełniony. W przeciwnym razie program „się zapętli", czyli instrukcje będą wykonywane w nieskończoność (w praktyce – do przepełnienia buforów pamięci).

Ponieważ w przypadku baz danych najczęstsze zastosowanie pętli wiąże się z konstrukcją kursora, zatem wszystkie przykłady pojawią się w kolejnym rozdziale poświęconym kursorom.

10. Kursory

a) Koncepcja kursora

W dotychczasowych rozważaniach dotyczących podstawienia na zmienne wartości odczytywanych z bazy danych przy użyciu instrukcji **SELECT** zakładaliśmy, że odczytywany jest co najwyżej jeden rekord, a wartości poszczególnych wyrażeń tego rekordu podstawiane są na odpowiednie zmienne. Dodatkowym zastrzeżeniem było, że jeśli instrukcja **SELECT** zwróci więcej niż jeden rekord, czyli zestaw (zbiór) rekordów, wówczas na zmienne zostaną podstawione wartości wyrażeń z ostatniego odczytanego rekordu (w przypadku T-SQL).

Jednakże często zachodzi sytuacja, w której pojedyncza instrukcja **SELECT** zwraca zestaw rekordów wynikowych (Result Set) i każdy z nich wymaga oddzielnej operacji na danych. Przykładem może być konieczność wykonania różnych operacji, zależnych od wartości w kolejnych odczytanych rekordach, lub konieczność dokonania zmian w innych tabelach, też zależnych od wartości w odczytywanych rekordach. W takich sytuacjach rozwiązanie stanowi użycie kursora.

Kursor jest konstrukcją programistyczną, w której rekordy odczytane przez instrukcję **SELECT** z tabel zostają zapisane w buforze pamięci, umożliwiającym dostęp do każdego pojedynczego rekordu i podstawienie na zmienne wartości wyrażeń tego rekordu. W ten sposób uzyskujemy możliwość odczytu i operowania na pojedynczych wartościach wyrażeń we wszystkich odczytanych (i zbuforowanych) rekordach.

Kursor może zostać użyty w dowolnym kodzie T-SQL – w bloku anonimowym, w procedurze składowanej czy w wyzwalaczu.

Należy jednak pamiętać, że kursory nie są cudownym przepisem na rozwiązanie wszystkich problemów programowania serwera baz danych. Ich zasadniczą wadą jest niska wydajność (długi czas realizacji kodu). Stąd wniosek, że należy ich używać tylko tam, gdzie nie ma innego rozwiązania. Jeżeli jakakolwiek operacja może być wykonana przez użycie „czystego" SQL, należy z tego skorzystać, rezygnując z użycia kursora.

Szczególnie często ekwiwalentem użycia kursora jest użycie skorelowanej instrukcji **UPDATE**.

b) Składnia kursora

Aby skorzystać z kursora musimy wykonać kilka kroków, ściśle określonych co do kolejności.

Pierwszym krokiem jest zdefiniowanie i zadeklarowanie kursora. Definicję kursora zawsze stanowi instrukcja **SELECT**, która może być utworzona na dowolnym źródle danych (tabela, widok, związki, tabele tymczasowe, CTE). Rekordy odczytane w wyniku działania tej instrukcji zostaną umieszczone w buforze, co umożliwi dostęp do każdego z tych rekordów oddzielnie. Kursor, tak jak zmienna, identyfikowany jest przez nazwę w instrukcji **DECLARE**:

```
DECLARE nazwa_kursora CURSOR FOR instrukcja_SELECT
```

Sama deklaracja kursora jest tylko informacją, jakie rekordy będą odczytywane z bazy danych oraz definicją nazwy, do której należy się odwołać w celu odczytania tych rekordów.

Odczytanie rekordów przez instrukcję **SELECT** zdefiniowaną w deklaracji kursora następuje po dyrektywie otwarcia kursora

```
OPEN nazwa_kursora
```

„Otwarcie" kursora powoduje odczytanie rekordów, przeniesienie ich do bufora pamięci (w MS SQL Server jest to tabela tymczasowa przechowywana w bazie tempdb), oraz założenie blokad na danych, które zostały odczytane przez kursor, celem uniemożliwienia dokonania zmian tych rekordów przez inną transakcję w trakcie działania kursora. Liczbę rekordów odczytanych z bazy przez ostatnio otwarty kursor przechowuje zmienna systemowa `@@Cursor_rows`.

Od tego momentu uzyskujemy dostęp do wartości wyrażeń poszczególnych rekordów. Instrukcja która odwołuje się do kolejnego wiersza i dokonuje podstawienia odczytanych wartości na wcześniej zadeklarowane zmienne ma postać

```
FETCH NEXT FROM nazwa_kursora INTO zmienne
```

Liczba zmiennych musi odpowiadać liczbie odczytywanych z rekordu wartości, a ich typy muszą być zgodne z typami odczytywanych wartości.

Zwykle kursor odczytuje więcej niż jeden wiersz. Żeby odczytać dane ze wszystkich dostarczonych do bufora rekordów należy użyć pętli w celu przejścia po wszystkich odczytanych rekordach. Pętla musi być wykonywana tak długo, jak istnieją jeszcze nieodczytane wiersze. Informacja o tym przekazywana jest przez zmienną systemową

@@Fetch_status. Zmienna ta przyjmuje wartość 0 jeśli został podstawiony kolejny wiersz, a jeśli nie, przyjmuje wartość -1. Zatem pętla

```
WHILE @@Fetch_status = 0
```

będzie wykonywana do momentu odczytania ostatniego wiersza. Wtedy zmienna @@Fetch_status przyjmie wartość – 1 i pętla się zakończy.

Po zakończeniu podstawiania na zmienne wartości wyrażeń wierszy instrukcją **FETCH** należy wykonać polecenie

```
CLOSE nazwa_kursora
```

które zwolni założone na danych blokady, oraz usunie z tabeli tymczasowej przechowywane tam wartości, umożliwiając tym samym ich dalszy odczyt przez inne procesy. Jednak definicja struktury kursora nie zostaje usunięta i można z niej po raz kolejny skorzystać, wykonując ponownie instrukcję **OPEN**.

Instrukcja **FETCH** jak i instrukcja **CLOSE** muszą odnosić się do otwartego kursora, gdyż w przeciwnym przypadku zostanie wygenerowany błąd:

```
Cursor is not open
```

Analogicznie instrukcja OPEN odnosząca się do już otwartego kursora, wywoła błąd (niekrytyczny):

```
The cursor is already open.
```

Instrukcją usuwająca strukturę kursora jest

```
DEALLOCATE nazwa_kursora
```

Usuwa ona odwołania kursora do tabel, i zwalnia zasoby pamięci dotychczas rezerwowane przez kursor.

Przykład II.9.1

*Pierwszy (trywialny) przykład to wypisanie z tabeli EMP na ekran nazwisk i pensji pracowników zarabiających powyżej 2000. Trywialność tego przykładu wynika z faktu, iż do odczytania danych można użyć tylko instrukcji **SELECT**, która definiuje kursor.*

```
DECLARE Test CURSOR FOR   SELECT Ename, SAL
                          FROM EMP
                          WHERE sal > 2000;
DECLARE @ename Varchar(20), @sal Money;
OPEN   Test;
FETCH NEXT FROM Test INTO @ename, @sal;
WHILE  @@FETCH_STATUS = 0
   BEGIN
      PRINT 'Pracownik ' + @ename + ' zarabia ' + Cast(@sal AS Varchar);
      FETCH NEXT FROM Test INTO @ename, @sal;
   END;
CLOSE Test;
DEALLOCATE Test;
```

Należy zwrócić uwagę na dwukrotne wystąpienie instrukcji **FETCH** w składni kursora. Pierwsze jej wystąpienie, przed otwarciem pętli, jest konieczne aby ustawić wartość zmiennej @@FETCH_STATUS . Bez jej ustawienia pętla nie została by zainicjowana – interpreter pominął by

instrukcje jej wnętrza. Drugie wystąpienie instrukcji **FETCH** , wewnątrz pętli, powinno się znajdować na jej końcu, bezpośrednio przed **END**, aby pobrać kolejny rekord do kolejnej iteracji, ewentualnie zmienić wartość zmiennej @@FETCH_STATUS jeśli pobranie nie powiedzie się, czyli jeżeli już zostały odczytane wszystkie rekordy z bufora i osiągnięty został kres zbioru rekordów.

Niewłaściwe umieszczenie drugiej instrukcji (lub jej brak) może spowodować nieprawidłowe działanie całego kursora, lub wejście w nieskończoną pętlę.

Przykład II.9.2

W kolejnym przykładzie kursor zostanie wykorzystany w celu odczytania z tabeli Emp płac wszystkich pracowników. Pracownicy zarabiający poniżej 1000 będą mieli podniesione płace o 100, pracownicy zarabiający powyżej 3000 – obniżone o 100. Po każdej takiej operacji zostanie wypisany komunikat z informacją o zmianie płacy.

```
DECLARE Test1 CURSOR FOR  SELECT ename, empno, sal
                          FROM   emp;
DECLARE @ename Varchar(15), @empno Int, @sal Money;
OPEN Test1
FETCH NEXT FROM Test1 INTO @ename, @empno, @sal;
WHILE @@Fetch_status = 0
BEGIN
   IF @sal > 3000
   BEGIN
      SET @sal = @sal - 100;
      UPDATE emp SET sal = @sal WHERE empno = @empno;
      PRINT @ename+ ' zarabia po obniżce ' + Cast(@sal As Varchar);
   END;
   IF @sal < 1000
   BEGIN
      SET @sal = @sal + 100;
      UPDATE emp SET sal = @sal WHERE empno = @empno;
      PRINT @ename + ' zarabia po podwyżce ' + Cast(@sal As Varchar);
   END;
FETCH NEXT FROM Test1 INTO @ename, @empno, @sal;
END;
CLOSE Test1;
DEALLOCATE Test1;
```

Cała operacja mogła by zostać zrealizowana przez dwie instrukcje **UPDATE** z warunkami **WHERE**, ale użycie kursora pozwala na wypisanie komunikatu ekranowego po każdej wykonanej operacji **UPDATE**.

Należy zwrócić uwagę na konieczność użycia klauzuli **WHERE** w instrukcji **UPDATE**, w celu dokonania aktualizacji właściwego rekordu w tabeli bazy danych. **UPDATE** działa w sposób typowy – brak klauzuli **WHERE** oznacza modyfikację wszystkich rekordów tabeli. Pomimo tego, że kursor dostarczył konkretny rekord, **UPDATE** „nie wie”, że modyfikacja ma dotyczyć tego właśnie rekordu.

Istnieje alternatywna składnia, rozwiązująca powyższy problem:

```
UPDATE emp SET sal = @sal WHERE CURRENT OF Test1;
```

Powyższy przykład nie został napisany optymalnie – kursor odczytuje z bazy rekordy, a dopiero po ich odczytaniu sprawdza warunki określone w założeniu, co powoduje niepotrzebne

obciążenie serwera, zatem spowalnia całą operację. Korekta jest prosta – wystarczy w deklaracji kursora w instrukcji **WHERE** ograniczyć zwracane rekordy:

```
DECLARE Test1 CURSOR FOR  SELECT  ename, empno, sal
                          FROM    emp
                          WHERE   sal NOT BETWEEN 1000 AND 3000;
```

c) Scroll cursor

Przy domyślnej deklaracji kursora odczyt wierszy następuje sekwencyjnie, instrukcja **FETCH NEXT** zawsze odczytuje następny wiersz po bieżącym. Ten sposób działania kursora można zmienić, deklarując

```
DECLARE nazwa_kursora SCROLL CURSOR FOR SELECT …
```

Po takiej deklaracji uzyskujemy dostęp do dowolnego wiersza, operując poleceniami:

```
FETCH NEXT          -- przejdź do następnego wiersza
FETCH PRIOR         -- przejdź do poprzedniego wiersza
FETCH FIRST         -- przejdź do pierwszego wiersza
FETCH LAST          -- przejdź do ostatniego wiersza
FETCH ABSOLUTE(n)   -- przejdź do wiersza o numerze n
FETCH RELATIVE(n)   -- przejdź do wiersza o n wierszy dalej niż bieżący, lub bliżej,
                       jeśli n < 0)
```

Podsumowanie części I

W tej części zostały zaprezentowane podstawowe konstrukcje Transact SQL - rozszerzenia programistycznego języka SQL w środowisku MS SQL Server. Ich użycie pozwala na efektywniejsze wykorzystanie serwera MS SQL w porównaniu z ograniczeniem do wyłącznego posługiwania się „czystą” składnią SQL. Stanowi też podstawę do rozpoczęcia nauki wykorzystania bardziej zawansowanych struktur, jakimi są procedury składowane i wyzwalacze, omawiane w części II.

II. Transact SQL (T-SQL) – procedury i wyzwalacze

1. Procedury składowane

a) Koncepcja procedur składowanych

Procedury składowane (stored procedures) to zwyczajowa nazwa procedur zapisanych w języku proceduralnym serwera bazy danych i przechowywanych w bazie, jako jej obiekty. Procedury mogą zawierać wszystkie prawidłowe, dotychczas omówione konstrukcje języka T-SQL, a także wszystkie prawidłowe instrukcje właściwego dialektu języka SQL. Tym samym procedury tworzą bardzo wygodną, elastyczną i bezpieczną w użyciu klasę obiektów, poprzez które można wykonywać większość powtarzalnych operacji na bazie danych.

Zasadnicze zalety użycia procedur składowanych to:

- uporządkowanie i centralizacja operacji na bazie danych, co pozwala na lepszą kontrolę nad operacjami wykonywanymi na bazie, gwarantując ich powtarzalność,
- wprowadzenie reguł bezpieczeństwa – aplikacja kliencka ma prawo uruchomić procedurę, a nie wykonywać dowolne polecenia bezpośrednio na tabelach, a jedyna komunikacja z serwerem polega na przekazaniu danych wsadowych i (ewentualnie) odebraniu wyników procedury,
- zmniejszenie liczby interakcji z bazą danych – jedna procedura może wywoływać wiele operacji na bazie, co radykalnie ogranicza ruch w sieci.

Dotychczas omawiane przykłady były fragmentami kodu, każdorazowo uruchamianymi przez użytkownika. Nie były jednak nigdzie trwale zapisywane – ich kod był odczytywany bezpośrednio z edytora w Management Studio i na bieżąco interpretowany (wykonywany). Mógł być przechowywany poza bazą danych w postaci plików tekstowych.

Ten sam kod może zostać zapisany w formie procedury. Procedura otrzymuje swoją unikalną nazwę (w obszarze bazy danych) i wszelkie odwołania do niej są odwołaniami do jej nazwy. Tak jak każda procedura napisana w dowolnym języku programowania, procedury składowane traktowane są jako zamknięta całość, z którą komunikacja odbywa się za pomocą parametrów procedury.

Procedury przy pierwszym wykonaniu są kompilowane, tworzony jest optymalny plan dostępu do danych.

W T-SQL procedury (a także wyzwalacze) mogą odwoływać się do tabel, które na etapie kompilowania jeszcze nie istnieją. Wewnątrz procedur mogą być również wykonywane operacje DDL.

b) Składnia procedury

Składnia polecenia tworzącego procedurę wygląda następująco:

```
CREATE [OR ALTER] PROCEDURE nazwa_procedury Lista_parametrów
AS
instrukcje_Transact_SQL
```

Po napisaniu kodu procedury należy ją skompilować, czyli uruchomić tworzący ją skrypt tak, jak ma to miejsce przy tworzeniu obiektów innych klas (tabel, widoków).

Przy zmianie procedury (jej ponownej kompilacji) słowo **CREATE** zastępowane jest przez **ALTER** zgodnie ze składnią DDL. Użycie składni **CREATE OR ALTER PROCEDURE** możliwe jest od wersji MS SQL Server 2016.

Parametry procedury, czyli zmienne przeznaczone do komunikacji pomiędzy procedurą a wykorzystującymi ją procesami, deklarowane są w takiej samej konwencji jak inne zmienne, ale bez poprzedzającego deklarację słowa **DECLARE**. Nazwa parametru zaczyna się od znaku @, po nim musi zostać podany typ danych, może zostać podana wartość domyślna. Parametr może zostać określony jako wyjściowy (**OUTPUT**), przekazujący wartości wyliczone w procedurze „na zewnątrz". Domyślnym rodzajem jest parametr wejściowy, przekazujący wartości z „zewnątrz" do procedury. Przy deklaracji parametru tego rodzaju, słowo **INPUT**, jako domyślne jest pomijane.

```
@nazwa_parametru TYP [= wartość_domyślna] [OUTPUT]
```

Jeżeli „lista instrukcji T-SQL" zawiera więcej niż jedną instrukcję, można (ale nie jest to konieczne) listę umieścić pomiędzy słowami **BEGIN** i **END**, tak jak w przypadku instrukcji warunkowej lub pętli.

Uruchomienie, czyli „wywołanie" procedury realizowane jest instrukcją **EXEC** lub **EXECUTE**:

```
EXEC nazwa_procedury [lista_wartości]
```

Lista wartości musi odpowiadać liście zadeklarowanych w procedurze parametrów, co do ich liczby i typów danych.

Parametry zadeklarowane z wartością domyślną, przy wywołaniu procedury mogą zostać pominięte, wówczas procedura zostanie wykonana z tymi wartościami. Jeśli przy wywołaniu procedury pojawią się dla nich wartości inne niż domyślne, to one zostaną użyte do obliczeń.

Jeżeli przy wywołaniu procedury nie nadajemy wartości parametrom, dla których określono wartości domyślne i nie są to parametry deklarowane na końcu listy, wówczas zastępujemy ich wartości słowem **DEFAULT**.

Przykład II.10.1

Utworzymy b. prostą procedurę, odwołującą się do danych z tabeli Emp

```
CREATE PROCEDURE Dept_Job @job Varchar(20) = 'MANAGER' ,@Deptno Int = 10
AS
BEGIN
SELECT Ename, Job, Deptno
FROM   emp
WHERE  job = @job AND Deptno = @Deptno;
END;
```

A następnie trzykrotnie uruchomimy tę procedurę manipulując wartościami parametrów, z którymi będzie uruchamiana.

Pierwsze uruchomienie – z domyślnymi wartościami parametrów:

```
EXEC dept_job
```

wynik

Ename	Job	Deptno
CLARK	MANAGER	10

Kolejne dwa uruchomienia i ich wyniki

```
EXEC dept_job 'CLERK'
```

wynik

Ename	Job	Deptno
MILLER	CLERK	10

```
EXEC dept_job Default, 20
```

wynik

Ename	Job	Deptno
JONES	MANAGER	20

2. Zwracanie wyników przez procedury T-SQL

Procedury nie muszą zwracać żadnych wartości do środowiska, które je wywołało. Ich działanie może sprowadzać się do wykonania pewnych operacji na bazie danych. Mogą też tak jak to miało miejsce w przykładzie II.10.1 przekazywać wartości przez nie wyliczone lub odczytane z bazy. Mogą to też być tylko komunikaty o wykonaniu (lub nie) pewnych operacji. Jeżeli procedura ma zwracać wartości wyliczone lub odczytane, może to robić na trzy sposoby:

- przez **Result Set**
- Przez parametry **OUTPUT**
- Przez dyrektywę **RETURN**

a) Result Set

Result Set to wynik **ostatniej** instrukcji **SELECT** wykonanej w procedurze. Ten wynik jest przekazywany do aplikacji, która procedurę uruchomiła. Jeżeli jest to Management Studio albo Azure Data Studio, wynik pojawia się w postaci tekstu na pulpicie aplikacji. Jeżeli jest to inna aplikacja (np. aplikacja webowa) ten wynik musi zostać przechwycony przez odpowiednio do tego przygotowaną kontrolkę. To zagadnienie nie jest tematem niniejszego cyklu wykładów.

Przykład II.10.2

Utwórz procedurę, która wypisze nazwiska, stanowiska i pensję pracowników z tabeli ***EMP****.*

```
CREATE PROCEDURE emp_data
AS
SELECT ename, job, sal
FROM EMP;
```

Ta procedura nie ma parametrów, zwraca wynik poprzez Result Set. Jeżeli zostanie uruchomiona w Management Studio, wynik w postaci trzech kolumn tekstu z nagłówkami pojawi się na pulpicie.

Przykład II.10.3

*Utwórz procedurę, która wypisze nazwiska, stanowiska i pensję pracowników z tabeli **EMP** z departamentu, którego numer podawany jest w parametrze procedury.*

```
CREATE PROCEDURE emp_data_dept @deptno Int = 10
AS
SELECT ename, job, sal
FROM   EMP
WHERE  deptno = @deptno;
```

Ta procedura również zwraca odczytane z tabeli EMP wartości przez Result set. Parametr @deptno posiada wartość domyślną 10. Jeżeli przy wywołaniu procedury wartość parametru zostanie pominięta, zostaną odczytane dane pracowników z departamentu 10. Jeśli zostanie podana inna wartość, dane dotyczyć będą pracowników z tego departamentu, którego numer podany zostanie w parametrze.

b) Parametr OUTPUT

W przykładach z poprzednich slajdów parametry były wykorzystywane w celu przekazania wartości DO procedury, gdzie wartość przekazana przez parametr była używana do wykonania operacji wewnątrz procedury. Wartości wyliczone w procedurze były przekazywane na zewnątrz w postaci Result set – zestawu rekordów przekazywanych do wywołującej ją aplikacji – np. MANAGEMENT STUDIO.

Parametry, jak wcześniej wspomniano, mogą zostać wykorzystane do przekazania wartości wyliczonych (odczytanych) wewnątrz procedury do środowiska (aplikacji) które uruchomiło procedurę. Są to parametry typu OUTPUT.

Przykład II.10.4

Utwórz procedurę, która dla podanej wartości empno zmodyfikuje zarobki wskazanego pracownika o zadany procent (domyślnie 20) i poprzez parametr OUTPUT zwróci nową wartość zarobków.

```
CREATE PROCEDURE Change_sal_txt
@Empno INT
,@Percent INT = 20
,@Info Varchar(50)  OUTPUT
AS
DECLARE @New_sal Money;
BEGIN
       SELECT @New_sal = Sal + Sal * @Percent / 100
       FROM   Emp
       WHERE  Empno = @Empno;
       UPDATE Emp SET Sal = @New_sal
       WHERE  Empno = @Empno;
       SET @Info = 'Pracownik o numerze ' + Cast(@Empno As Varchar) + ' zarabia
                    obecnie ' + Cast(@New_sal As Varchar);
END;
```

Aby wywołać procedurę Change_sal_txt, pobrać i wyświetlić wartość z parametru wyjściowego, należy zadeklarować zmienną, na którą zostanie przekazana wartość przechowywana przez zmienną typu OUTPUT w procedurze.

```
DECLARE @New_sal_out Varchar(50);
EXECUTE Change_sal_txt 7369, Default, @New_sal_out OUTPUT;
```

```
PRINT  @New_sal_out;
```

c) RETURN

Kolejnym sposobem, w jaki procedura może zwracać wartość, jest instrukcja **RETURN**. Instrukcja ta powoduje przerwanie realizacji kodu i natychmiastowe wyjście z procedury (zakończenie jej działania). Poprzez tę instrukcję można przekazać na zewnątrz procedury wyłącznie wartość typu Integer (Int).

Przekazanie wyliczonych wartości dokonywane jest przez nazwę procedury, co upodabnia ten rodzaj procedur do funkcji.

Przykład II.10.5

Utwórz procedurę, która przy użyciu instrukcji **RETURN** *zwróci liczbę pracowników z tabeli emp.*

```
CREATE PROCEDURE Ile_pracownikow
AS
BEGIN
       DECLARE       @ile INT;
       SELECT @ile = COUNT(1) FROM Emp;
       RETURN @ile;
END;
```

i jej uruchomienie

```
DECLARE @Ilu_prac Varchar(30);
EXECUTE @Ilu_prac = Ile_pracownikow;
PRINT   @Ilu_prac;
```

3. Podsumowanie procedur

Procedury składowane są wygodnym i elastycznym narzędziem komunikacji z bazą danych. Jeżeli przyjmiemy, że odrzucamy możliwość bezpośredniego komunikowania się aplikacji klienckich z tabelami bazy danych (co jest rozwiązaniem zdecydowanie niezalecanym), a komunikacja odbywa się albo poprzez widoki (perspektywy), albo poprzez procedury, to niewątpliwie to drugie rozwiązanie jest lepszym wyborem.

Tworząc procedury, należy pamiętać o pewnych zasadach, nie będących wymogami formalnymi, lecz wskazówkami jak powinna być napisana sprawnie działająca procedura.

- Pierwszą instrukcją, umieszczoną zaraz po słowie **AS** powinna być instrukcja SET NOCOUNT ON, wyłączająca wysyłanie komunikatów o wykonaniu instrukcji **SELECT**, **INSERT**, **UPDATE**, **DELETE**, **MERGE**, co wyłączy wykonywanie zbędnych operacji
- w instrukcjach **SELECT** nie powinny pojawiać się funkcje, jeżeli nie operują one na wartościach zwracanych, gdyż wymaga to wyliczenia ich wartości dla każdego wiersza;
- nie należy używać konstrukcji **SELECT** * ….
- należy na jak najwcześniejszym etapie ograniczać dane odczytywane z bazy,
- należy stosować jawne transakcje ujęte w polecenia **BEGIN**/**END TRANSACTION**

- transakcje powinny być jak najkrótsze, aby do minimum ograniczać czas blokowania rekordów i ryzyko wystąpienia zakleszczeń,
- obsługa błędów wewnątrz procedur powinna być realizowana w blokach **TRY**…**CATCH**,
- nie istnieje odgórne ograniczenie na rozmiar kodu,
- dopuszczalne są zagnieżdżenia procedur (wywołanie jednej procedury przez inną) do max. 32 poziomów.

Wśród początkujących programistów baz danych korzystających z Management Studio lub Azure Data Studio pojawia się dość częsty błąd, polegający na tym, że po pierwszej kompilacji procedury zostaje ona uruchomiona poleceniem `EXEC nazwa_procedury`, po czym zostają wprowadzone do procedury poprawki i wykonywana jest kolejna kompilacja, tym razem już z poleceniem `EXEC nazwa_procedury` jako ostatnia linia kodu samej procedury(!). W efekcie, przy kolejnym jej uruchomieniu, procedura sama siebie wywołuje (32 razy!), po czym aplikacja się zawiesza, aż do wyczyszczenia logów. Dobrze, jeżeli admin jest dostępny…

4. Wyzwalacze (triggers)

a) Koncepcja wyzwalaczy

Wyzwalacze są to procedury, tworzące oddzielną klasę obiektów bazy danych. Wyzwalacze są procedurami szczególnymi, gdyż są uruchamiane automatycznie przez SZBD, wskutek zajścia zdarzenia, które zostało przewidziane jako zdarzenie uruchamiające wyzwalacz. Zdarzeniami tymi mogą być operacje DML, DDL a także zdarzenia na poziomie serwera. Wyzwalacze zostały wprowadzone do Standardu SQL:1999.

Koncepcja wyzwalaczy w MS SQL (Transact SQL) i ORACLE (Pl/SQL) znacznie się różni (nie tylko składnią!). Przestrzegam przed próbami automatycznego przenoszenia rozwiązań pomiędzy tymi dwoma środowiskami, gdyż (delikatnie mówiąc) nie przyniesie to oczekiwanego efektu.

Wyzwalacze służą do:

- oprogramowywania więzów spójności,
- oprogramowywania stałych czynności, które muszą być jednakowo zrealizowane dla każdej aplikacji klienckiej korzystającej z bazy danych.

Ponieważ najczęściej wykorzystywane są wyzwalacze uruchamiane poleceniami DML, w tym wykładzie zajmiemy się tym typem wyzwalaczy.

b) Wyzwalacze uruchamiane poleceniami DML

Każdy wyzwalacz musi być powiązany z jedną tabelą lub widokiem bazy danych, ale z każdym z tych obiektów może być powiązanych kilka wyzwalaczy. Na razie skupimy się na wyzwalaczach powiązanych z tabelami, w dalszej części omówiony zostanie wariant łączony z widokami.

Wyzwalacze są uruchamiane (często mówimy „odpalane”) automatycznie przez SZBD PO zajściu odpowiedniego zdarzenia DML dotyczącego tabeli, z którą wyzwalacz jest związany. Zdarzenia uruchamiające wyzwalacz są definiowane w treści wyzwalacza.

c) Składnia wyzwalacza

Składnia polecenia tworzącego wyzwalacz wygląda następująco:

```
CREATE [OR ALTER] TRIGGER nazwa_wyzwalacza
ON nazwa_tabeli
FOR instrukcje_DML
AS
instrukcje_Transact_SQL
```

Przy zmianie wyzwalacza (jego ponownej kompilacji) słowo **CREATE** zastępowane jest przez **ALTER** zgodnie ze składnią DDL. Użycie składni **CREATE OR ALTER TRIGGER** podobnie jek w przypadku procedur możliwe jest od wersji MS SQL Server 2016.

Instrukcjami DML które mogą uruchomić wyzwalacz są:

- **INSERT**
- **UPDATE**
- **DELETE**

Po słowie **FOR** może pojawić się jedna, dwie lub wszystkie trzy instrukcje oddzielone przecinkami.

d) Działanie wyzwalaczy

Wyzwalacze T-SQL uruchamiane są PO zakończeniu wykonywania instrukcji DML która uruchamia wyzwalacz, ale w kontekście transakcji, związanej z tą instrukcją. Zatem tabela, z którą związany jest wyzwalacz, w momencie jego uruchomienia jest już zmieniona przez instrukcję DML, ale transakcja jeszcze się nie zakończyła i zmiany nie są utrwalone. Poprawne zakończenie działania wyzwalacza zakończy transakcję i zmiany zostaną utrwalone. Jeżeli jednak operacja DML się nie powiedzie, albo wyzwalacz nie zakończy się prawidłowo, zmiany zostaną wycofane. Wycofanie zmian może nastąpić także w wyniku polecenia **ROLLBACK** umieszczonego na liście poleceń wyzwalacza[2].

Dostęp do wartości wierszy na których operuje instrukcja DML uruchamiająca wyzwalacz, jest możliwy z poziomu wyzwalacza. Kopie tych wierszy umieszczane są w wirtualnych tabelach o nazwach **INSERTED** i **DELETED.** Tabele te są „tylko do odczytu” (Read Only), a ich „istnienie” kończy się wraz z końcem działania wyzwalacza[3].

W przypadku instrukcji **INSERT** kopie wstawionych wierszy znajdują się w tabeli INSERTED, w przypadku instrukcji **DELETE** kopie wierszy usuniętych znajdują się w tabeli

[2] Omówiony w tym akapicie proces dotyczy domyślnego ustawienia MS SQL Server, w którym każda poprawnie wykonana instrukcja DML traktowana jest jako oddzielna transakcja.
[3] To rozwiązanie jest dostępne dla wszystkich operacji DML, niekoniecznie związanych z wyzwalaczami. To zagadnienie zostanie omówione w dalszej części pracy.

DELETED, w przypadku instrukcji **UPDATE** kopie wierszy w postaci sprzed zmiany umieszczane są w tabeli DELETED, w postaci po zmianie w tabeli INSERTED.

Tabele INSERTED i DELETED są tabelami Read Only (tylko do odczytu), czyli ingerencja w ich zawartość nie jest możliwa. Natomiast z poziomu wyzwalacza dostępne są do odczytu i modyfikacji wszystkie tabele bazy danych, łącznie z tabelą, na której wyzwalacz jest zdefiniowany.

e) Kilka wyzwalaczy na jednej tabeli

W MS SQL Server dla każdego zdarzenia DML na danej tabeli można utworzyć kilka wyzwalaczy, ale nie istnieje ustalona kolejność ich uruchamiania (!!!).

f) Sekwencyjne uruchamianie wyzwalaczy

Jeżeli zostaje uruchomiony wyzwalacz, który dokonuje zmian na kolejnej tabeli, na której również zdefiniowany jest wyzwalacz zostanie on uruchomiony. Jeżeli on z kolei dokona zmian w trzeciej tabeli na której jest wyzwalacz … itd. do 32 zagnieżdżeń. Jeżeli tego typu odwołania zainicjują nieskończoną pętlę wywołań, zostanie przekroczony limit poziomów wywołań i operacje wyzwalaczy zostaną anulowane, a wykonane zmiany wycofane.

Możliwa jest rekursja pośrednia – wyzwalacz TR1 na tabeli T1 modyfikuje tabelę T2, co uruchamia wyzwalacz TR2 na tej tabeli, który modyfikuje tabelę T1.

Rekursja bezpośrednia, w której wyzwalacz TR1 modyfikuje tabelę T1 z którą jest związany, na skutek czego zostaje uruchomiony ponownie wyzwalacz TR1 itd. jest możliwa po zmianie ustawień bazy danych (czego się jednak nie zaleca!).

g) Kontrola zmian dokonywanych przez wyzwalacz

Sprawdzenie, czy kolumna została zmodyfikowana przez instrukcję **UPDATE** lub **INSERT** jest możliwe przez użycie wewnątrz wyzwalacza jednoargumentowej funkcji `UPDATE`(nazwa_kolumny), zwracającej wartość logiczną (**TRUE** lub **FALSE**), zależnie od faktu modyfikacji (lub jej braku) kolumny o nazwie podanej jako argument funkcji. Argumentem funkcji jest nazwa kolumny.

Sprawdzenie, które kolumny zostały zmodyfikowane instrukcją **UPDATE** lub **INSERT** jest możliwe przez użycie wewnątrz wyzwalacza funkcji `COLUMNS_UPDATED`(). Funkcja zwraca maskę bitową (varbinary) złożoną z jednego lub kilku bajtów, pozwalającą zidentyfikować aktualizowane kolumny. Bit odpowiadający zmienianej kolumnie jest ustawiany na 1, a kolejność bitów odpowiada kolejności kolumn w tabeli, na której operuje wyzwalacz – prawy bit pierwszego bajta odpowiada pierwszej kolumnie tabeli.

W przypadku instrukcji **INSERT** funkcja `COLUMNS_UPDATED`() zwraca TRUE dla wszystkich kolumn przyjmując, że bez względu na wstawione wartości wszystkie kolumny zostały zmodyfikowane.

Szczegóły w dokumentacji MS SQL Server.

Przykład II.13.1

Utwórz wyzwalacz, który nie dopuści do usunięcia wierszy z tabeli EMP.

```
CREATE TRIGGER TR1
       ON EMP
       FOR DELETE
       AS
       ROLLBACK;
```

Wyzwalacz jest uruchamiany w tej samej niezatwierdzonej transakcji co instrukcja (w tym przypadku **DELETE**), ale już po jej wykonaniu. Tabela jest już w stanie zmienionym, ale zmiany mogą zostać wycofane, gdyż nie została zakończona i zatwierdzona transakcja. Zakończenie transakcji nastąpi wraz z zakończeniem działania wyzwalacza.

Przykład II.13.2

Utwórz wyzwalacz, który nie dopuści do przypisania pracownikowi płacy niższej niż 100 (kolumna sal w tabeli Emp). W przypadku próby naruszenia tej reguły, wyzwalacz wycofa zmiany i zgłosi błąd.

```
CREATE TRIGGER TR2 ON EMP FOR INSERT, UPDATE
AS
BEGIN
DECLARE @Sal Money;
SELECT @Sal = SAL FROM Inserted;
IF @Sal < 100
       BEGIN
              ROLLBACK;
       Raiserror ('Niedopuszczalna wartość SAL!', 1, 2);
END;
END;
```

Wyzwalacz z powyższego przykładu można zastąpić warunkiem CHECK zdefiniowanym dla kolumny Sal w tabeli EMP. Takie rozwiązanie jest lepsze, niż użycie wyzwalacza.

Wyzwalacz w T-SQL jest uruchamiany dla całej instrukcji DML. Instrukcja DML może operować na jednym wierszu (taka sytuacja jest prosta), lub na wielu (szczególnie instrukcje **UPDATE** i **DELETE**). W tym przypadku tabele INSERTED i/lub DELETED będą również zawierały wiele wierszy. Aby w takiej sytuacji dotrzeć do każdego zmienianego wiersza, wewnątrz wyzwalacza trzeba będzie użyć kursora do przejrzenia wszystkich wierszy w tabelach tymczasowych wyzwalacza.

Przykład z poprzedniego slajdu będzie działał poprawnie tylko wtedy, jeśli instrukcja która go uruchomiła operowała na jednym wierszu.

Nieco lepsze, choć nie doskonałe rozwiązanie tego problemu przedstawione jest w przykładzie następnym.

Przykład II.13.3

```
CREATE TRIGGER TR3 ON EMP FOR INSERT, UPDATE
AS
BEGIN
IF EXISTS (SELECT 1 FROM inserted WHERE Sal < 100)
BEGIN
       ROLLBACK;
```

```
        Raiserror ('Niedopuszczalna wartość SAL!', 1, 2);
END;

END;
```

To rozwiązanie jest niezależne od liczby wierszy, na których operuje instrukcja, jednak w przypadku naruszenia reguły, wycofana zostanie cała instrukcja (wszystkie zmienione rekordy), a nie tylko rekordy regułę naruszające. W takim przypadku, jeśli chcemy indywidualnie traktować każdy zmieniany wiersz, trzeba użyć kursora. Ponadto warto rozdzielić wyzwalacz na dwa niezależne – jeden obsługujący **INSERT**, drugi **UPDATE**.

Przykład II.13.4

```
CREATE TRIGGER TR1 ON EMP FOR INSERT
AS
IF EXISTS (SELECT 1 FROM inserted WHERE Sal < 100)
BEGIN
DECLARE TR_cursor CURSOR FOR SELECT empno FROM INSERTED
        WHERE Sal < 100;
DECLARE @empno INT;
OPEN TR_cursor
FETCH NEXT FROM  TR_cursor INTO @empno;
WHILE @@FETCH_STATUS = 0
BEGIN
        DELETE FROM EMP WHERE EMPNO = @empno;
        FETCH NEXT FROM  TR_cursor INTO @empno;
END;
CLOSE TR_cursor;
DEALLOCATE TR_cursor;
END;
```

h) Wyzwalacze INSTEAD OF

Istnieje jeszcze jeden rodzaj wyzwalaczy, który w założeniu miał pozwalać na zrealizowanie postulatu Codd'a pełnego dostępu do danych poprzez perspektywy. Jak wiadomo istnieją silne ograniczenia dotyczące możliwości operowania danymi poprzez perspektywy, co zostało omówione w jednym z poprzednich wykładów.

Wyzwalacz **INSTEAD OF** definiowany na perspektywie (widoku), pozwala na wykonanie operacji **INSERT**, **UPDATE**, **DELETE** przez perspektywę niekoniecznie spełniającą te restrykcyjne wymogi. Realizowane jest to przy użyciu kodu języka proceduralnego i instrukcji SQL, umieszczonych w treści wyzwalacza, a operujących indywidualnie na tabelach, które wchodzą w skład definicji perspektywy.

MS SQL Server po uruchomieniu wyzwalacza **INSTEAD OF** pomija wykonanie instrukcji DML, która wyzwalacz uruchomiła i wykonuje wyłącznie kod zawarty w wyzwalaczu. Ponadto ten serwer pozwala na tworzenie wyzwalaczy **INSTEAD OF** zarówno powiązanych z widokami, jak też z tabelami (czego nie dopuszcza serwer DB ORACLE).

Składnia tworząca taki wyzwalacz wygląda następująco:

```
CREATE TRIGGER IO_TR1 ON nazwa_perspektywy_lub_tabeli
INSTEAD OF INSERT|UPDATE|DELETE
AS
BEGIN
  Instrukcje SQL i T-SQL
```

```
    (...)
END;
```

Przykład II.13.5

Tworzymy widok na tabelach EMP i DEPT

```
CREATE VIEW EmpDept_View
AS
SELECT ename, SAL, dname
FROM   EMP
       INNER JOIN DEPT ON EMP.DEPTNO = DEPT.DEPTNO;
```

Aby zapewnić możliwość dopisania przez ten widok nowego rekordu pracownika w tabeli EMP, a w przypadku braku departamentu o podanej w instrukcji INSERT nazwie departamentu, także rekordu w tabeli DEPT, tworzymy wyzwalacz INSTEAD OF na powyższym widoku.
Dla uproszczenia zakładamy, że instrukcja INSERT będzie operowała na jednym rekordzie. Takie założenie pozwala uniknąć konieczności użycia kursora.

```
CREATE TRIGGER IO_T1 ON EmpDept_View
INSTEAD OF INSERT
AS
BEGIN
DECLARE @Deptno Int, @Empno Int;
IF NOT EXISTS (SELECT      1
       FROM   DEPT D, inserted I
       WHERE  D.DNAME = i.dname)
BEGIN
       SELECT @Deptno = Isnull(MAX(deptno), 0) + 10 FROM DEPT;
       INSERT INTO DEPT (DEPTNO, DNAME)
       SELECT @Deptno, dname FROM inserted;
END;
SELECT @Empno = ISNULL(Max(empno), 0) + 1 FROM EMP;
INSERT INTO EMP      (EMPNO, ENAME, SAL, DEPTNO)
SELECT @Empno, ENAME, SAL, @Deptno
FROM inserted;
END;
```

Działanie wyzwalacza sprawdzamy wykonując instrukcję:

```
INSERT INTO EmpDept_View (ename, SAL, dname)
VALUES ('Kowalski', 1200, 'Marketing');
```

Efektem jej wykonania jest utworzenie dwóch rekordów:

w tabeli EMP:

```
7935 Kowalski NULL  NULL NULL 1200 NULL 50 NULL
```

i w tabeli DEPT:

```
50 Marketing NULL
```

i) Podsumowanie wyzwalaczy

Wyzwalacze to bardzo silne narzędzie w ręku programisty baz danych, może okazać się za silne dla mniej doświadczonych deweloperów. Ich zasadniczą cechą jest jednakowe traktowanie każdej operacji DML, bez względu na to, jaki proces ją wywołał. Stanowi to jeden z gwarantów spójności danych w bazie.

Ale ta sama cecha stanowi też ich wadę – być może różne procesy powinny być odmiennie obsługiwane. Jeszcze większą wadę wyzwalaczy stanowi realna możliwość utraty kontroli nad ich działaniem, a zwłaszcza nad korelacjami wynikłymi z operacji na powiązanych tabelach, wyposażonych w wyzwalacze. Obie te cechy wskazują na konieczność zachowania bardzo dużej ostrożności przy posługiwaniu się wyzwalaczami.

Jedną z możliwości zmniejszenia ryzyka jest wyłączanie z poziomu kodu wyzwalacza innych wyzwalaczy, które mogą powodować interakcję. Ale jeszcze lepszym rozwiązaniem jest ograniczenie do niezbędnego minimum użycia wyzwalaczy na rzecz procedur.

5. Obsługa błędów

MS SQL Server implementuje dwie instrukcje podnoszące błąd: **RAISERROR** oraz **THROW**. Działanie tych instrukcji różni się między sobą.

Obsługa błędów może odbywać się w deklarowanych bezpośrednio po sobie blokach:

BEGIN TRY ... END TRY i **BEGIN CATCH ... END CATCH**

Zasadnicze instrukcje procedury umieszczane są w bloku **TRY** i wykonywane, dopóki nie zostanie podniesiony błąd. W przypadku wystąpienia błędu sterowanie może zostać przekazane do bloku **CATCH** i w takim przypadku wykonywane są dalej instrukcje zawarte w tym bloku. Błąd może zostać podniesiony poprzez uruchomienie zarówno instrukcji **RAISERROR** jak i **THROW**, a także przez SZBD.

Bloków **TRY** i **CATCH** może być w procedurze kilka, mogą być w sobie zagnieżdżane.

a) Instrukcja RAISERROR

Poniżej omawiam instrukcję w jej uproszczonej wersji. Opis wersji pełnej, pozwalającej na bardziej elastyczne użycie tej instrukcji, znajdzie czytelnik w dokumentacji serwera.

Składnia w wersji pełnej:

```
RAISERROR ( { msg_id | msg_str | @local_variable }
    { ,severity ,state }
    [ ,argument [ ,...n ] ] )
    [ WITH option [ ,...n ] ]
```

Składnia w wersji uproszczonej:

```
RAISERROR (message|zmienna_lokalna, severity, state)
```

Gdzie:

`Message` - dowolny tekst (komunikat błędu) lub zmienna lokalna typu Char lub Varchar
`Severity` - liczba z przedziału 0-25 (przy czym „zwykły” użytkownik może definiować wartości z przedziału 0-18, użytkownik przypisany do roli sysadmin 19 - 25)
`State` - liczba z przedziału 1-127 oznaczająca numer błędu

Błąd podniesiony w bloku **TRY** poleceniem **RAISERROR** z wartością Severity >= 11 powoduje zakończenie przetwarzania w tym bloku i przekazanie sterowania do skojarzonego bloku **CATCH**. Błędy podniesione z wartością Severity 10 lub mniejszą zwracają komunikat

(message) w tym samym bloku **TRY**, po czym kontynuowane jest wykonywanie poleceń w bloku.

Błąd podniesiony poleceniem **RAISERROR** poza blokiem **TRY** nie przerywa realizacji poleceń.

Błąd podniesiony z wartością Severity 20 – 25 jest traktowany jako „fatal error" i skutkuje przerwaniem sesji klienta, której ten błąd dotyczy oraz zapisaniem komunikatu do logu.

Wartości wyspecyfikowane w poleceniu **RAISERROR** po jego uruchomieniu w bloku **TRY** są dostępne w bloku **CATCH** poprzez funkcje systemowe ERROR_LINE, ERROR_MESSAGE, ERROR_NUMBER, ERROR_PROCEDURE, ERROR_SEVERITY, ERROR_STATE i zmienną systemową @@ERROR.

- ERROR_NUMBER - numer błędu z systemowej listy sys.messages, podobne do zmiennej @@ERROR
- ERROR_MESSAGE - pełny tekst komunikatu o błędzie zawierający wszystkie parametry, np. nazwa obiektu;
- ERROR_LINE - numer linii w której pojawił się błąd;
- ERROR_SEVERITY – wartość parametru Severity wywołanego błędu;
- ERROR_PROCEDURE - jeżeli błąd został wygenerowany wewnątrz procedury będzie to jej nazwa,
- ERROR_STATE – zwraca wartość State z którą błąd był wywołany.

W przypadku uruchomienia poza blokiem **TRY** funkcje te zwracają NULL, a zwraca numer błędu 0 dla Severity 0 – 10 i numer błędu 5000 dla Severity > 10.

Przykład II.14.1

Podniesienie błędów o różnych Severity poza blokiem TRY

```
DECLARE @err_info Varchar(10);
SET @err_info = 'Poziom 1';
RAISERROR (@err_info, 1, 1);
    SELECT error_message(), @@Error;
SET @err_info = 'Poziom 2';
RAISERROR (@err_info, 12, 2);
    SELECT Error_message(), @@Error;
Print 'Tekst kontrolny';
```

Przykład II.14.2

Podniesienie błędów o różnych Severity w bloku TRY

```
BEGIN TRY
DECLARE @err_info Varchar(10);
SET @err_info = 'Poziom 1';
RAISERROR (@err_info, 1, 1);
    Print 'Tekst kontrolny 1';
SET @err_info = 'Poziom 2';
RAISERROR (@err_info, 12, 2);
    Print 'Tekst kontrolny 2';
END TRY
BEGIN CATCH
SELECT Error_message() Komunikat,
       Error_number() Nr,
       @@Error Nr2,
       Error_state() State;
END CATCH;
```

a) Instrukcja THROW

Składnia instrukcji **THROW** jest zbliżona do składni **RAISERROR** w wersji uproszczonej:

```
THROW (error_number|zmienna_lokalna,
       message|zmienna_lokalna,
       state|zmienna_lokalna);
```

Gdzie:

`error_number` – liczba typu Int nie mniejsza niż 50000 reprezentująca numer błędu,
`message` – nVarchar(2048) – opis błędu
`state` – liczba Tinyint wskazująca na stan błędu

Zasadniczą różnicą jest to, że **THROW** może zostać użyte tylko w bloku **TRY** lub **CATCH**. Użycie w bloku **TRY** działa identycznie jak **RAISERROR** z parametrem Severity = 16, czyli zawsze przekieruje sterowanie do bloku **CATCH**. Użycie **THROW** bez parametrów bloku **CATCH** pozwala na powtórne wypisanie informacji o błędzie, który został podniesiony.

```
BEGIN TRY
    -- Wywołanie błędu dzielenia przez zero.
    SELECT 1/0;
END TRY
BEGIN CATCH
       THROW
END CATCH;
```

Po dalsze szczegóły zagadnienia obsługi błędów oraz działania poleceń **RAISERROR** i **THROW** odsyłam czytelnika do dokumentacji serwera.

b) Podsumowanie obsługi błędów

Przedstawione zostały podstawowe narzędzia służące do obsługi błędów w kodzie T-SQL. Dalsze szczegóły znajdują się w dokumentacji serwera. Ich wykorzystanie wymaga pewnej praktyki, ale należy przyjąć jako regułę przy pisaniu kodu, że przewiduje się możliwości wystąpienia błędów i gdy wystąpią zostają przechwycone i odpowiednio opisane. Zapobiega to sytuacjom, w których użytkownik niespodziewanie otrzymuje niezrozumiały (z jego punktu widzenia) komunikat systemowy o wystąpieniu błędu, co na ogół kończy się lekką paniką użytkownika i rutynowym stwierdzeniem „system nie działa”.

Podsumowanie części II

W części II zostały omówione trzy zagadnienia – tworzenie procedur składowanych, wyzwalaczy oraz obsługa błędów. Korzystanie z tych rozwiązań pozwala na efektywne i bezpieczne wykorzystanie bazy danych, w tym przypadku zarządzanej przez MS SQL Server.

III.Transact SQL (T-SQL) – konstrukcje (nieco bardziej) zaawansowane

1. Skrótowy zapis operacji algebraicznych

Możliwe jest użycie skróconej formy zapisu operacji algebraicznych. Poniżej przedstawiam przykłady, bez komentarza zakładając ich oczywistość.

```
DECLARE @Ile Int = 10;
SET @Ile += 5;
   PRINT @Ile;   --15
SET @Ile -= 5;
   PRINT @Ile;   --10
SET @Ile *= 5;
   PRINT @Ile;   --50
SET @Ile /= 2;
   PRINT @Ile;   --25

-- i jeszcze modulo, czyli reszta z dzielenia (często się przydaje)

SELECT @Ile %= 7
   PRINT @Ile    --4
go

-- i jeszcze raz modulo, z użyciem dodatkowej zmiennej

DECLARE @Ile Int = 30, @div Int = 7;
SELECT @Ile %=@div;
   PRINT @Ile;
```

2. Funkcje "rankingowe" w MS SQL Server

Funkcje „rankingowe" stanowią uzupełnienie funkcji agregujących, pozwalające na bardziej precyzyjne i wygodne operowanie danymi połączonymi w grupy rekordów.

a) Funkcja NTILE

```
NTILE (n) OVER ([ <partition_by_clause> ] <order_by_clause>)
```

Funkcja dzieli zwracane przez instrukcję **SELECT** rekordy na n równych ilościowo grup. Jeżeli zestaw rekordów nie dzieli się bez reszty, tworzone są grupy dwóch rozmiarów, różniące się jednym rekordem.

Przykład III.2.1

Podziel pracowników z tabeli EMP na pięć grup wynikających z sortowania po wysokości płacy

```
SELECT NTILE(5) OVER(ORDER BY sal) NrGrupy,
                                   job,
                                   ename,
                                   sal
FROM emp;
```

Przykład III.2.2

Podziel studentów (tabele OSOBA i STUDENT) każdego rocznika na dwie grupy:

```
SELECT NTILE(2) OVER(PARTITION BY YEAR(DataRekrutacji) ORDER BY nazwisko) "Nr grupy",
                YEAR(DataRekrutacji) Rocznik,
                Imie,
                Nazwisko
FROM Osoba O JOIN Student s ON O.IdOsoba = s.idosoba;
```

b) Funkcja ROW_NUMBER

Numeruje wynikowe wiersze, zaczynając od 1, zgodnie warunkiem sortowania. Jeżeli została użyta klauzula PARTITION BY, numeracja odbywa się w obrębie każdej partycji.

```
ROW_NUMBER ( ) OVER ([ PARTITION BY value_expression , ...
                       [n]] order_by_clause)
```

Przykład III.2.2

Uszereguj pracowników z tabeli EMP według wysokości płacy.

```
SELECT ROW_NUMBER() OVER(
       ORDER BY sal) Pozycja,
       job,
       ename,
       sal
FROM   emp;
```

Przykład III.2.3

Uszereguj pracowników z tabeli EMP według wysokości płacy w obrębie każdego stanowiska.

```
SELECT ROW_NUMBER() OVER(PARTITION BY job
       ORDER BY sal) Pozycja,
       job,
       ename,
       sal
FROM   emp;
```

Funkcja pozwala wykonać obliczenia funkcji agregujących, przypisując ich wyniki do każdego (!) rekordu należącego do grupy, bez konieczności użyci dodatkowych konstrukcji takich jak CTE lub widok.

Przykład III.2.4

W wierszu każdego z pracowników z tabeli EMP wypisz jego nazwisko, płacę, stanowisko oraz sumę płac, średnią, minimalną i maksymalną płacę na tym stanowisku.

```
SELECT  ename,job, sal,
Sum(sal) Over (Partition by job) Sumsal,
Avg(sal) Over (Partition by job) Avsal,
Min(sal) Over (Partition by job) Minsal,
Max(sal) Over (Partition by job) Maxsal
FROM emp;
```

c) Funkcja RANK

Działa podobnie jak ROW_NUMBER, ale RANK to 1 + liczba rekordów poprzedzających w grupie (o mniejszej wartości).

Przykład III.2.5

Dla każdego pracownika z tabeli EMP określ jego pozycję zarobkową na jego stanowisku. Pracownicy o takich samych zarobkach znajdują się na tej samej pozycji.

```
SELECT RANK() OVER(PARTITION BY job
       ORDER BY sal) Pozycja,
       job,
       ename,
       sal
FROM   emp;
```

d) Funkcja DENSE_RANK

Działa tak jak RANK, ale zachowuje ciągłość numeracji.

Przykład III.2.6

Wykonaj polecenie z poprzedniego przykładu, tym razem używając funkcji DENSE_RANK. W obu wynikach porównaj numerację dla grupy SALESMAN.

```
SELECT DENSE_RANK() OVER(PARTITION BY job
       ORDER BY sal) Pozycja,
       job,
       ename,
       sal
FROM   emp;
```

3. Tabele tymczasowe w MS SQL Server

Tabele tymczasowe to struktury fizycznie przechowujące dane, ale istniejące poza schematem bazy danych, tworzone „ad hoc” w miarę potrzeb programistycznych. Tabele tymczasowe przechowywane są w bazie systemowej **tembdb** i są widoczne w Management Studio w oknie Object Explorer w gałęzi tej bazy.

Tabele tymczasowe tworzymy wówczas, gdy musimy wielokrotnie odwołać się do wyniku odczytanych danych, zwłaszcza ze złożonych (silnie obciążających serwer) zapytań SQL, albo musimy przechować pośrednie wyniki w procedurze. Mogą też być narzędziem do przenoszenia danych pomiędzy procedurami.

Tabele tymczasowe **nie mogą zawierać kluczy obcych** (więzów referencyjnych FOREIGN KEY). W tabelach tymczasowych mogą być tworzone indeksy oraz więzy integralności, którym można nadawać nazwy. Na tabelach tymczasowych dopuszczalne są wszystkie operacje DDL i DML takie, jakie są możliwe dla „zwyczajnych” tabel.

Składnia polecenia **CREATE TABLE** tworzącego tabelę tymczasową jest identyczna jak dla „normalnych” tabel. Nazwa tabeli tymczasowej musi zaczynać się od znaku # (tabela tymczasowa lokalna) lub ## (tabela tymczasowa globalna).

CRAETE TABLE *#nazwa_tabeli_tymczasowej_lokalnej (...)*

Tabele **tymczasowe lokalne** widoczne są wyłącznie w obrębie sesji (połączenia z bazą danych), w trakcie której zostały utworzone i usuwane są automatycznie po zakończeniu tej sesji.

CRAETE TABLE *##nazwa_tabeli_tymczasowej_globalnej (...)*

Tabele **tymczasowe globalne** widoczne są dla wszystkich sesji (połączeń z serwerem bazy danych) i usuwane są automatycznie, gdy nastąpi zamknięcie sesji w trakcie której tabela została utworzona i żadna inna sesja już z niej nie korzysta.

Można także utworzyć tabelę tymczasową poleceniem

SELECT *lista_kolumn* **INTO** *#tabela* **FROM** *tabela*;

Tabela tymczasowa **LOKALNA**, utworzona wewnątrz procedury jest usuwana po zakończeniu działania tej procedury. Tabela **GLOBALNA** zachowa się zgodnie z powyższymi regułami.

Każda tabela tymczasowa może zostać usunięta poleceniem **DROP TABLE**.

Wszystkie operacje SQL na tabelach tymczasowych wykonywane są tak samo jak na tabelach „zwykłych”.

Pomimo, że SZBD samodzielnie zarządza usuwaniem tabel tymczasowych, sugeruję jawne ich usuwanie zarówno w procedurach, jak też poza nimi.

4. Zmienne tabelaryczne T-SQL

Zmienne tabelaryczne to rozwiązanie podobne do tabel tymczasowych. Są to zmienne, które mogą przechowywać tablice – zawierają nazwane kolumny o określonych typach danych, mogą posiadać więzy integralności:

- **PRIMARY KEY**,
- **UNIQUE**,
- **NULL**,
- **CHECK**.

Zmienne tabelaryczne powoływane są tak jak wszystkie zamienne po słowie DECLARE, ich nazwa musi zaczynać się od znaku @ a po nazwie musi pojawić się słowo TABLE oraz deklaracja struktury tabeli - nazwy kolumn i ich typy danych.

Zmienne tabelaryczne **nie mogą posiadać kluczy obcych** (więzów referencyjnych FOREIGN KEY). Więzom integralności zmiennych tablicowych **nie można nadawać nazw**.

Przykład III.4.1

Zadeklaruj zmienną tabelaryczną, podstaw na nią numery pracowników, nazwiska, stanowiska, nazwy departamentów pracowników z działu 20, następnie odczytaj te dane ze zmiennej.

```
DECLARE @emp20 TABLE (
                    Nr_prac       Int,
                    Nazwisko      Varchar(20),
                    Stanowisko    Varchar(10),
                    Nazwa_dzialu  Varchar(10)
                    );

INSERT INTO @emp20
SELECT empno,
       ename,
       job,
       dname
FROM EMP e JOIN DEPT d ON e.deptno = d.DEPTNO
WHERE e.DEPTNO = 20;

SELECT * FROM @emp20;
```

Czas życia zmiennej tabelarycznej ograniczony jest, tak jak ma to miejsce w przypadku zmiennych, czyli do realizacji bloku, w którym zmienna tabelaryczna została zadeklarowana, albo do czasu wykonywania procedury, wewnątrz której zmienna jest używana.

Zmienne tabelaryczne mogą być częścią wszystkich instrukcji DML (**SELECT**, **INSERT**, **UPDATE**, **DELETE**).

Dla zmiennych **tablicowych nie można użyć instrukcji**

SELECT lista_kolumn **INTO** @tabela …;

tworzącej „w locie” nową tabelę o strukturze resultset’a, która był dopuszczalna w przypadku tabel tymczasowych. Wynika to stąd, że zmienna tabelaryczna musi mieć określoną strukturę w momencie jej utworzenia i ta struktura nie może być zmieniana.

Generalnie – używajmy zmiennych tabelarycznych! Zwłaszcza wtedy, kiedy musimy wielokrotnie sięgnąć do danych odczytanych z tabel (czyli z dysku). Taki odczyt jest kosztowny, a zmienna tabelaryczna przechowuje odczytane dane w pamięci RAM.

5. Alternatywa dla podzapytania w UPDATE i DELETE

Instrukcje **UPDATE** i **DELETE** mogą zostać skonstruowane z użyciem powiązań z innymi tabelami. Nie oznacza to jednak, że rekordy usuwane są ze związku tabel. Operacja wykonywana jest zawsze **na jednej tabeli**.

Przykład III.5.1

Podnieś płace pracowników zatrudnionych w DALLAS o 123.

```
UPDATE e
SET SAL += 123
FROM EMP e JOIN DEPT d ON e.DEPTNO = d.DEPTNO
WHERE LOC = 'DALLAS';
```

Przykład III.5.2

Usuń pracowników zatrudnionych w DALLAS.

```
DELETE e
FROM EMP e JOIN DEPT d ON e.DEPTNO = d.DEPTNO
WHERE LOC = 'DALLAS';
```

6. Dostęp do zmienianych danych w T-SQL

T-SQL oferuje możliwość przechwycenia zmienianych danych, w sposób analogiczny do tego, jak ma to miejsce w wyzwalaczach T-SQL. Dane zmieniane są kopiowane do wirtualnych tabel INSERTED i DELETED, skąd mogą zostać odczytane i zapisane albo w zadeklarowanej w tym celu zmiennej tabelarycznej, albo we wcześniej utworzonej tabeli tymczasowej.

Poniższe przykłady pokazują użycie tego rozwiązania dla **INSERT** i **UPDATE**.

Przykład III.6.1

Prosty przykład przechwycenia danych wprowadzanych do tabeli instrukcją INSERT.

```
DECLARE @Out TABLE (
                    empno Int,
                    Ename Varchar(20),
                    job Varchar(20),
                    sal Money,
                    Hiredate Date
                    );

DECLARE @empno Int;

INSERT INTO emp (empno, ename, job, sal, hiredate)
       OUTPUT INSERTED.empno, INSERTED.Ename, INSERTED.job,
              INSERTED.sal, INSERTED.hiredate
       INTO @Out     --tu nie może pojawić się średnik !!!
SELECT Isnull(Max(empno), 0) + 1, 'Pink', 'ANALYST', 1200, Getdate() FROM emp;

SELECT @empno = empno FROM @Out;

SELECT * FROM @Out;
SELECT * FROM emp WHERE empno = @empno;
```

Przykład III.6.2

Przechwyć na zmienną dane wprowadzane do tabeli Osoba, w której kolumna klucza głównego IdOsoba jest typu Integer a jej wartości realizowane przez metodę Identity.

```
DECLARE @Out Table (
                   IdOsoba Int,
                   Imie Varchar(30),
                   Nazwisko Varchar(30),
                   DataUrodzenia Date,
                   IdPanstwo Int
                   );
DECLARE @Id Int;

INSERT INTO Osoba (Imie, Nazwisko, DataUrodzenia)
       OUTPUT INSERTED.IdOsoba,
              INSERTED.Imie,
              INSERTED.Nazwisko,
              INSERTED.DataUrodzenia,
              INSERTED.IdPanstwo
       INTO @Out
VALUES ('Benedykt', 'Bączek', '12-12-1998');

SELECT * FROM @Out;

SELECT @Id = IdOsoba FROM @out;
SELECT * FROM Osoba WHERE IdOsoba = @Id;
```

Jak widać z powyższego przykładu, w tabeli INSERTED (a także w DELETED) tworzone są kopie całych zmienianych wierszy, nawet jeżeli polecenia DML nie operują na wszystkich ich polach.

7. CASE (przypomnienie)

Konstrukcja z użyciem klauzuli **CASE** stanowi w SQL pewien ekwiwalent rozszerzenia instrukcji warunkowej, znanej z języków programowania. Musi zostać użyta wewnątrz polecenia SELECT i umożliwia zróżnicowanie danych zwracanych przez tą instrukcję w zależności od zdefiniowanego warunku. Pozwala to uniknąć np. tworzenia struktur wielokrotnie powtarzającej się tej samej instrukcji **SELECT** ze zróżnicowanymi warunkami w klauzuli **WHERE** i łączenia ich przez **UNION**.

W składni podawana jest wartość, która zostanie zwrócona (Simple **CASE**), albo wyrażenie, którego wartość zostanie wyliczona (Searched **CASE**) zależnie od wartości odczytywanych ze źródła danych (tabeli, widoku).

CASE może zostać użyte zarówno w instrukcji **SELECT**, jak też w **UPDATE** oraz instrukcji podstawienia **SET**.

Przykład III.7.1

Przy użyciu CASE przetłumacz nazwy stanowisk (job) z tabeli Emp. Przykład Simple Case.

```
SELECT ename,
       CASE job  WHEN 'CLERK' THEN 'Urzędnik'
                 WHEN 'ANALYST' THEN 'Analityk'
                 WHEN 'MANAGER' THEN 'Menedżer'
                 WHEN 'SALESMAN' THEN 'Sprzedawca'
                 WHEN 'PRESIDENT' THEN 'Prezio'
                 ELSE 'Innych nie ma'
       END AS Info
FROM EMP;
```

Przykład III.7.2

Przy użyciu CASE skomentuj płace z tabeli Emp. Przykład Searched Case.

```
SELECT ename,
       sal,
       Case  WHEN SAL < 1000 THEN 'Cienias'
             WHEN SAL > 4500 THEN 'To chyba sam KING!'
             ELSE 'Średniak'
       END As Comment
FROM EMP;
```

8. Skorelowany UPDATE (przypomnienie)

Dane do modyfikacji tabeli mogą być pobierane z innego źródła danych (tabeli, widoku, wyrażenia CTE). Zachodzi wówczas problem stworzenia odpowiedniego powiązania pomiędzy tabelą modyfikowaną, a źródłem danych. Rozwiązanie programistyczne stanowi użycie kursora, ale jak wiadomo jest ono nieefektywne. W wielu przypadkach można zastąpić użycie kursora przez korelację działającą tak samo, jak to miało miejsce w przypadku zapytań skorelowanych z podzapytaniami w instrukcjach **SELECT**.

Składnia MS SQL Server dopuszcza dwa warianty skorelowanego **UPDATE** (ORACLE tylko pierwszy):

UPDATE *tabela_modyfikowana*
SET *kolumna* = (**SELECT** *wyrażenie*
FROM *źródło_rekordów*
WHERE *warunek_powiązania*);

albo

UPDATE *tabela_modyfikowana*
SET *kolumna* = *wyrażenie* **FROM** *źródło_rekordów*
WHERE *warunek_powiązania*;

Poniżej przykłady, których celem jest jedynie weryfikacja obu składni, a logika jest pomijalna.

```
UPDATE EMP SET DEPTNO = (SELECT DEPTNO
                         FROM DEPT
                         WHERE EMP.DEPTNO = DEPT.DEPTNO);
```

```
UPDATE EMP SET EMP.DEPTNO = DEPT.DEPTNO
FROM DEPT
WHERE EMP.deptno= DEPT.DEPTNO;
```

```
UPDATE e SET e.DEPTNO = DEPT.DEPTNO
FROM EMP e JOIN DEPT ON e.DEPTNO = DEPT.DEPTNO;
```

Wszystkie trzy wariant polecenia działają poprawnie. Ale sprawdźmy jeszcze polecenie, które aktualizuje dane na podstawie wyniku funkcji agregującej.

```
UPDATE EMP SET SAL = (SELECT AVG(sal)
                      FROM EMP e1
                      WHERE EMP.JOB = e1.JOB);
```

Powyższe polecenie wykonuje się poprawnie. Ale próba wykonania tej samej operacji w składni alternatywnej

```
UPDATE e1 SET SAL = AVG(e2.sal)
FROM EMP e1 INNER JOIN EMP e2 ON e2.JOB = e1.JOB;
```

podnosi błąd:

```
An aggregate may not appear in the set list of an UPDATE statement.
```

Na koniec przykład wykorzystujący **CTE**, **CASE** i skorelowany **UPDATE**.

Przykład III.8.1

*Utwórz tabelę **Siatka_plac** {Job, Placa} i korzystając z wymienionych wyżej konstrukcji wypełnij ją danymi, przypisując każdemu stanowisku inną wartość wynagrodzenia.*

```
SELECT DISTINCT Job
INTO SiatkaPlac
FROM EMP ;

ALTER TABLE SiatkaPlac
ADD Placa Money;

WITH SP (JB, Placa)
AS(
SELECT JOB, CASE Job WHEN 'PRESIDENT' THEN 5000
                     WHEN 'MANAGER' THEN 4000
                     WHEN 'ANALYST' THEN 3000
                     WHEN 'SALESMAN' THEN 2000
                     WHEN 'CLERK' THEN 1000
                     END
FROM SiatkaPlac)

UPDATE SiatkaPlac SET Placa = SP.Placa
FROM SP
WHERE JOB = SP.JB;

SELECT * FROM SiatkaPlac;
```

9. Common Table Expression z rekursją

W poprzednich wykładach konstrukcję Common Table Expression (CTE) wykorzystywaliśmy w roli „dostarczyciela rekordów", zastępując tym użycie widoku lub podzapytania. Teraz przedstawiamy rekurencyjne wykorzystanie tej konstrukcji.

Przykład pierwszy – trywialny.

Przykład III.9.1

Korzystając z CTE wygeneruj kolumnę licz 1- 10.

```
WITH Kolumna (X)
AS
(
   SELECT  0
   UNION   ALL
   SELECT  X + 1
   FROM    Kolumna
   WHERE   X < 10
)

SELECT X FROM Kolumna;
```

Przykład III.9.2

Tym razem CTE w roli generatora określonego rekurencyjnie ciągu Fibonacciego {0, 1, Fn = F(n – 1) + F(n – 2)} w zakresie <0, 144>.

```
WITH Fibonacci (X1, X2)
AS
(
   Select 0,  1
   UNION      ALL
   SELECT X2, X1 + X2
   FROM       Fibonacci
   WHERE      X1 + 1 < 144

)
SELECT X1, X2
FROM Fibonacci;
```

W powyższym rozwiązaniu kolumna X2 jest kolumną pomocniczą, ale wartości jej wyniku mogą być traktowane jako wyrazy ciągu Fibonacciego rozpoczynającego się od 1 a nie od 0.

W kolejnych przykładach zaczniemy operować na danych z tabeli Emp

Przykład III.9.3

Wygeneruj nazwiska pracowników oraz ich szefów z tabeli EMP, podając w wyniku poziom podległości.

```
WITH X (EmpId, EmpName, SzefId, Poziom)
AS
(SELECT empno,
        ename,
        mgr,
        1
FROM EMP
WHERE ENAME = 'KING'
UNION ALL
        SELECT EMP.EMPNO,
        EMP.ENAME,
        EMP.MGR,
        X.poziom + 1
FROM EMP
INNER JOIN X ON X.EmpId = EMP.MGR AND Poziom < 4)

SELECT * FROM X;
```

Przykład III.9.4

Wypisz w czterech kolejnych kolumnach nazwiska pracowników na kolejnych poziomach podległości .

```
WITH X (Id, naz, S_id, Nr)
as
(SELECT empno,
        ename,
        mgr,
        1
FROM emp
WHERE emp.ename = 'KING'
UNION ALL
SELECT  emp.empno,
        emp.ename,
        emp.mgr,
        X.Nr + 1
FROM emp
INNER JOIN X ON X.Id = emp.mgr AND Nr < 4)

SELECT  X1.naz Szef,
        X2.naz "Podlega 1",
        X3.naz "Podlega 2",
        X4.naz "Podlega 3"
FROM X X1  JOIN X X2 ON X1.Id = x2.S_id AND X1.Nr = 1
           LEFT JOIN X X3 ON X2.Id = X3.S_id AND X2.Nr = 2
           LEFT JOIN X X4 ON X3.Id = X4.S_id AND X3.Nr = 3;
```

10. MERGE

a) Koncepcja MERGE

Częstym przypadkiem, z jakim mamy do czynienia w codziennej eksploracji danych w bazach relacyjnych jest zagadnienie jednoczesnej operacji aktualizacji (**UPDATE**) rekordów istniejących w tabeli oraz dopisania (**INSERT**) rekordów, których w tabeli nie ma, z jednoczesną możliwością usunięcia (**DELETE**) rekordów zbędnych.

Prześledźmy następujące zagadnienie na schemacie EMP, DEPT i SALGRADE. Chcemy utworzyć tabelę Budzety_dzialow w której co miesiąc będą zapisywane kwoty przeznaczone na płace pracowników poszczególnych działów. Ale pisząc instrukcję (procedurę) realizującą taką operację musimy założyć nie tylko zmiany w wysokości budżetów działów (wynikające ze zmiany płac lub zmian zatrudnienia), ale także powstawanie nowych działów, ewentualnie konieczność zlikwidowania informacji o działach nie zatrudniających aktualnie pracowników. Istnieje także możliwość, że tabela docelowa jest pusta.

Rozwiązać ten problem można tworząc procedurę, która przy pomocy instrukcji warunkowych wybierze rekordy do aktualizacji, dopisania i usunięcia. Alternatywą dla rozwiązania programistycznego jest użycie instrukcji **MERGE** operującej „czystą” składnią SQL, zatem efektywniejszą w działaniu niż sekwencja operacji DML.

Koncepcja instrukcji **MERGE** opiera się na idei złączenia tabeli aktualizowanej ze źródłem danych (rekordów), porównania odpowiadających sobie rekordów zgodnie z warunkami złączenia (i ewentualnymi warunkami dodatkowymi) a następnie wykonaniu odpowiednich akcji w stosunku do rekordów „dopasowanych” i „niedopasowanych”.

b) Składnia MERGE

Instrukcja **MERGE** składa się z pięciu podstawowych klauzul:

- **MERGE** – definiuje docelową tabele (lub widok) na której zostaną dokonane operacje **INSERT**, **UPDATE** lub **DELETE**,
- **USING** – określa źródło danych łączone z tabelą docelową,
- **ON** – określa warunki złączenia tabeli docelowej ze źródłem danych,
- **WHEN** (**WHEN MATCHED THEN**, **WHEN NOT MATCHED BY TARGET THEN**, **WHEN NOT MATCHED BY SOURCE THEN**) – określa akcje, które zostaną zrealizowane w wyniku złączenia i dodatkowych kryteriów wyszukiwania podanych w klauzuli **WHEN**,
- **OUTPUT** – zwraca wiersze z informacją o każdym wstawionym, zmienionym lub usuniętym wierszu w tabeli docelowej, pozwalając stworzyć swoisty log przeprowadzonych operacji.

Pojęcie „dopasowania" wierszy w tabeli docelowej i źródle danych, wynikające z warunku złączenia określonego w klauzuli ON należy rozumieć następująco:

- Para „dopasowanych" wierszy składa się z jednego **wiersza w źródle** i **jednego wiersza w tabeli docelowej**; dla tych wierszy realizowane są akcje określone przez klauzulę **WHEN MATCHED THEN**,
- Wobec wierszy ze **źródła danych**, które nie mają odpowiedników (zgodnie z warunkiem złączenia) w **tabeli docelowej**, zostaną zrealizowane akcje określone w klauzuli **WHEN NOT MATCHED BY TARGET THEN**,
- Wobec wierszy w **tabeli docelowej**, które nie mają odpowiedników (zgodnie z warunkiem złączenia) **w źródle danych**, zostaną zrealizowane akcje określone w klauzuli **WHEN NOT MATCHED BY SOURCE THEN**.

c) Role poszczególnych klauzul

Klauzula **WHEN MATCHED THEN** - po tej klauzuli może pojawić się instrukcja **UPDATE** albo instrukcja **DELETE**. Instrukcja **MERGE** może zawierać maksymalnie dwie klauzule **WHEN MATCHED**. W takim przypadku pierwsza klauzula musi zostać uzupełniona przez **AND** *warunek*, druga z klauzul zostanie zrealizowana tylko wtedy, kiedy pierwsza zostanie pominięta, jedna musi zawierać **UPDATE** a pozostała **DELETE**.

Klauzula **WHEN NOT MATCHED BY TARGET THEN** może pojawić się w całej instrukcji tylko raz i specyfikuje wiersze które zostaną dopisane do tabeli docelowej.

Klauzula **WHEN NOT MATCHED BY SOURCE THEN** specyfikuje wiersze w tabeli docelowej nie mające odpowiedników w źródle danych. Instrukcja **MERGE** może zawierać maksymalnie dwie klauzule **WHEN NOT MATCHED BY SOURCE**. W takim przypadku pierwsza klauzula musi zostać uzupełniona przez **AND** *warunek*, druga z klauzul zostanie zrealizowana tylko wtedy, kiedy pierwsza zostanie pominięta, jedna musi zawierać **UPDATE** a pozostała **DELETE**.

Instrukcja **MERGE** musi być zakończona średnikiem!!!

Przykład III.10.1

Utwórz tabelę BudzetyDzialow przeznaczoną do rejestrowania sumy płac poszczególnych działów wraz z datą tej rejestracji. Utwórz tabelę tymczasową przechowującą informacje o rezultatach działania instrukcji MERGE, generowanych przez klauzulę OUTPUT.

```
CREATE TABLE BudzetyDzialow (
                              Deptno  Int NULL,
                              Budzet  Int NULL,
                              DataAkt Datetime
                              );

GO
CREATE TABLE #MyTempTable (
                            ExistDept   Int,
                            ExistBudzet Int,
                            ActionTaken Nvarchar(10),
                            NewDept     Int,
                            NewBudzet   Int,
                            DataAkt     Datetime

                            );

GO
MERGE BudzetyDzialow AS Target    -- tabela docelowa

USING (SELECT   deptno,
                Sum(sal)
       FROM     emp
       GROUP BY deptno) AS Source (Dzial, Budzet) -- źródło danych

ON     Target.Deptno = Source.Dzial  -- warunek złączenia

WHEN MATCHED THEN      UPDATE SET Budzet = Source.Budzet,
                                  DataAkt = Getdate()

WHEN NOT MATCHED THEN  INSERT (deptno, Budzet, DataAkt )
                       VALUES (Source.dzial,
                               Source.Budzet,
                               Getdate())

OUTPUT deleted.deptno, -- wypisanie informacji o operacji
       deleted.budzet,
       $action,
       inserted.*
INTO   #MyTempTable;
```

Zawartość tabel po pierwszym uruchomieniu instrukcji:

```
Deptno      Budzet      DataAkt
----------- ----------- -----------------------
NULL        800         2022-11-16 23:04:44.473
10          8750        2022-11-16 23:04:44.473
20          10075       2022-11-16 23:04:44.473
30          9000        2022-11-16 23:04:44.473
```

```
ExistDept   ExistBudzet ActionTaken NewDept    NewBudzet   DataAkt
----------- ----------- ----------- ----------- ----------- -----------------------
NULL        NULL        INSERT      NULL        800         2022-11-16 23:04:44.473
NULL        NULL        INSERT      10          8750        2022-11-16 23:04:44.473
NULL        NULL        INSERT      20          10075       2022-11-16 23:04:44.473
NULL        NULL        INSERT      30          9000        2022-11-16 23:04:44.473
```

Dokonamy teraz zmian w tabelach Emp i Dept:

```
UPDATE EMP
SET SAL = SAL*1.1
WHERE DEPTNO = 10;

UPDATE EMP
SET DEPTNO =  40
WHERE DEPTNO IS NULL;
```

Ponownie uruchomimy polecenie MERGE i ponownie odczytamy zawartość obu tabel:

```
Deptno      Budzet      DataAkt
----------- ----------- -----------------------
NULL        800         2022-11-16 23:04:44.473
10          9625        2022-11-16 23:06:45.507
20          10075       2022-11-16 23:06:45.507
30          9000        2022-11-16 23:06:45.507
40          800         2022-11-16 23:06:45.507
```

```
ExistDept   ExistBudzet ActionTaken NewDept    NewBudzet   DataAkt
----------- ----------- ----------- ----------- ----------- -----------------------
NULL        NULL        INSERT      NULL        800         2022-11-16 23:04:44.473
NULL        NULL        INSERT      10          8750        2022-11-16 23:04:44.473
NULL        NULL        INSERT      20          10075       2022-11-16 23:04:44.473
NULL        NULL        INSERT      30          9000        2022-11-16 23:04:44.473
10          8750        UPDATE      10          9625        2022-11-16 23:06:45.507
20          10075       UPDATE      20          10075       2022-11-16 23:06:45.507
30          9000        UPDATE      30          9000        2022-11-16 23:06:45.507
NULL        NULL        INSERT      40          800         2022-11-16 23:06:45.507
```

Jak widać, zaktualizowany rekord działu o numerze (obecnie) 40 został potraktowany jako nowy rekord i dopisany do tabeli. Rekord poprzedni, opisujący ten dział pozostał niezmieniony. Wynika to ze zdefiniowania warunku złączenia tabel docelowej i źródłowej po wartościach deptno, wśród których występuje NULL, niemożliwy do powiązania z żadnym innym rekordem.

Problem „niezaktualizowanego" działu, który wyjściowo w kolumnie deptno instrukcji ***SELECT*** *użytej dla zdefiniowania źródła rekordów miał wartość NULL możemy rozwiązać modyfikując instrukcję* ***MERGE****:*

```
MERGE BudzetyDzialow AS Target
USING (SELECT deptno, Sum(sal) FROM      emp GROUP BY  deptno)
              AS Source (Dzial, Budzet)
ON     Target.Deptno = Source.Dzial
WHEN MATCHED THEN UPDATE SET Budzet = Source.Budzet,
              DataAkt = Getdate()
WHEN NOT MATCHED BY TARGET THEN INSERT (deptno, Budzet, DataAkt)
       VALUES (Source.dzial, Source.Budzet,Getdate())
WHEN NOT MATCHED BY SOURCE THEN DELETE
OUTPUT deleted.deptno,     -- wypisanie informacji o operacji
       deleted.budzet,
       $action,
       inserted.*
INTO   #MyTempTable;
```

Po kolejnym uruchomieniu instrukcji MERGE, bez względu na zmiany wykonane na tabeli EMP, rekord niepowiązany z tabeli ***BudzetyDzialow*** *zostanie usunięty, i jej stan będzie wyglądał następująco:*

```
Deptno      Budzet      DataAkt
----------- ----------- -----------------------
10          9625        2022-11-16 23:08:13.230
20          10075       2022-11-16 23:08:13.230
30          9000        2022-11-16 23:08:13.230
40          800         2022-11-16 23:08:13.230
```

Z kolei tabela #MyTempTable będzie zawierała informacje o ostatniej aktualizacji:

```
ExistDept   ExistBudzet ActionTaken NewDept    NewBudzet   DataAkt
----------- ----------- ----------- ----------- ----------- -----------------------
NULL        NULL        INSERT      NULL        800         2022-11-16 23:04:44.473
NULL        NULL        INSERT      10          8750        2022-11-16 23:04:44.473
NULL        NULL        INSERT      20          10075       2022-11-16 23:04:44.473
NULL        NULL        INSERT      30          9000        2022-11-16 23:04:44.473
10          8750        UPDATE      10          9625        2022-11-16 23:06:45.507
20          10075       UPDATE      20          10075       2022-11-16 23:06:45.507
30          9000        UPDATE      30          9000        2022-11-16 23:06:45.507
NULL        NULL        INSERT      40          800         2022-11-16 23:06:45.507
NULL        800         DELETE      NULL        NULL        NULL
10          9625        UPDATE      10          9625        2022-11-16 23:08:13.230
20          10075       UPDATE      20          10075       2022-11-16 23:08:13.230
30          9000        UPDATE      30          9000        2022-11-16 23:08:13.230
40          800         UPDATE      40          800         2022-11-16 23:08:13.230
```

11. Funkcje T-SQL

Funkcje (user defined functions, stored functions) są obiektami programistycznymi przechowywanymi w bazie danych. Ich struktura i rola jest zbliżona do procedur składowanych, zwłaszcza zwracających wynik działania poprzez dyrektywę **RETURN**. W odróżnieniu od procedur, funkcje mogą być użyte bezpośrednio w poleceniach SQL i wyrażeniach T-SQL, bez konieczności wywoływania ich instrukcją **EXECUTE**.

Funkcja w swym kodzie może zawierać wszystkie dotychczas poznane polecenia SQL i T-SQL. Funkcja poprzez odwołanie się do jej nazwy zwraca wynik określony po instrukcji **RETURN**, typu określonego po instrukcji **RETURNS**. Funkcje mogą używać parametrów (argumentów).

Funkcje pozwalają uprościć kod. Warto stosować funkcje dla wielokrotnie powtarzanych obliczeń, w różnych miejscach aplikacji. Nie należy natomiast używać funkcji w składni instrukcji **SELECT** zwracającej więcej niż jeden rekord, jeżeli w każdym rekordzie funkcja będzie przyjmowała tą samą wartość. W takiej sytuacji należy obliczyć wartość funkcji i podstawić ją na zmienną, a następnie w składni polecenia **SELECT** użyć zmiennej.

Przy odwołaniu się do funkcji jej nazwa musi być poprzedzona (kwalifikowana) nazwą schematu (patrz poniższe przykłady). Nazwa funkcji musi być zakończona dwoma nawiasami nawet wówczas, gdy nie używa parametrów, zarówno przy jej definiowaniu jak też przy odwołaniu do niej.

a) Funkcje skalarne

Utworzenie funkcji:

```
CREATE FUNCTION nazwa_funkcji ([Lista_parametrów])
RETURNS Typ_danych
AS
BEGIN
Instrukcje SQL i T-SQL
RETURN (wyrażenie lub zmienna)
END;
```

Przykład III.11.1

Utwórz funkcję zwracającą średnią płacę w dziale, którego numer będzie podawany w parametrze funkcji. Użyj tej funkcji do znalezienia danych pracowników o zarobkach wyższych niż średnia płaca w dziale 20.

```
CREATE FUNCTION srednia_w_dziale (@deptno INT)
RETURNS MONEY
AS
BEGIN
       DECLARE      @srednia MONEY;
       SELECT @srednia = AVG(sal)
       FROM   emp
       WHERE  deptno = @deptno;
       RETURN @srednia;
END;
```

... i wykorzystanie funkcji

```
SELECT ename Nazwisko,
       sal Płaca,
       deptno "Nr działu"
FROM   emp
WHERE  sal > dbo.srednia_w_dziale(20)
ORDER BY ename;
```

Przykład III.11.2

Użyj funkcji z poprzedniego przykładu do znalezienia danych pracowników o zarobkach wyższych niż średnia płaca w działach, w których są zatrudnieni.

```
SELECT ename Nazwisko,
       sal Płaca,
       deptno "Nr działu",
       dbo.srednia_w_dziale(deptno) "Średnia płaca w dziale"
FROM   emp
WHERE  sal > dbo.srednia_w_dziale(deptno)
ORDER BY 3, 1 Desc;
```

Jak widać, w obu przypadkach użycie funkcji wyeliminowało konieczność użycia podzapytania lub konstrukcji CTE.

b) Funkcje tabelaryczne

Funkcje tabelaryczne różnią się od funkcji skalarnych tym, że zwracają nie pojedynczą wartość, lecz tabelę wartości. Są w pewnym stopniu ekwiwalentem perspektyw, uzupełnionym o możliwość użycia parametrów.

Definiując funkcję tabelaryczną, po dyrektywie **RETURNS** nie podajemy typu danych, lecz słowo **TABLE**. Składnia funkcji tabelarycznej powinna być zakończona instrukcją **SELECT** definiującą zwracany przez funkcję zestaw rekordów. Funkcje tabelaryczne mogą być używane w poleceniach SQL tak samo jak tabele lub perspektywy.

Przykład III.11.3

Utwórz funkcję zwracającą dane pracowników działu, którego numer podany jest w parametrze funkcji.

```
CREATE FUNCTIONPpracownicy_dzialu (@deptno INT)
RETURNS TABLE
AS
RETURN (
SELECT empno,
       ename,
       job,
       sal,
FROM   emp
WHERE  deptno = @deptno);
```

...i użycie funkcji w instrukcji SELECT, tak jak tabeli

```
SELECT * FROM pracownicy_dzialu(30);
```

c) Złożone funkcje tabelaryczne

W funkcji tabelarycznej przedstawionej poprzednio można było użyć tylko jednej instrukcji **SELECT**, zwracającej wynik. Struktura wynikowego zestawu rekordów powstawała automatycznie, na podstawie samej instrukcji. Złożone funkcje tabelaryczne pozwalają zdefiniować tę strukturę, czyli jej kolumny i typy danych.

W tego typu funkcji, po słowie **RETURNS** definiowana jest zmienna tabelaryczna (ze zdefiniowaną strukturą), która jest następnie w kodzie wypełniana. Wypełnianie zmiennej tabelarycznej może być realizowane przez więcej niż jedną instrukcję SQL. Całość kończona jest słowem **RETURN**.

Przykład III.11.3

Korzystając z rozwiązania przykładu III.9.4 utwórz funkcję, która będzie zwracała zestawienie (ścieżkę) podległości dla pracownika, którego nazwisko zostanie podane w parametrze funkcji.

```
CREATE Function Podleglosc (@Name Varchar(10))
RETURNS @Hierarchy TABLE
              (
              Szef Varchar(10),
              Podl1 Varchar(10),
              Podl2 Varchar(10),
              Podl3 Varchar(10)
              )
AS
BEGIN
WITH X (Id, naz, S_id, Nr)
as
(SELECT empno, ename, mgr, 1
FROM emp
WHERE emp.ename = 'KING'
UNION ALL
SELECT emp.empno, emp.ename, emp.mgr, X.Nr + 1
FROM emp
INNER JOIN X
ON X.Id = emp.mgr
AND Nr < 4)
INSERT INTO @Hierarchy (Szef, Podl1, Podl2, Podl3)
SELECT X1.naz, X2.naz, X3.naz, X4.naz
FROM X X1
JOIN X X2
ON X1.Id = x2.S_id AND X1.Nr = 1
LEFT JOIN X X3
ON X2.Id = X3.S_id AND X2.Nr = 2
LEFT JOIN X X4
ON X3.Id = X4.S_id AND X3.Nr = 3
WHERE X1.naz = @Name OR X2.naz = @Name OR X3.naz = @Name  OR X4.naz = @Name;
RETURN;
END;
```

... i wywołanie

```
SELECT * FROM Podleglosc('Martin');
```

d) Ograniczenia składniowe dla funkcji

Funkcje

- nie mogą być wykorzystywane do modyfikacji bazy danych
- nie mogą zawierać klauzuli **OUTPUT INTO**
- nie mogą realizować obsługi błędów przy użyciu **TRY**...**CATCH**, **@ERROR**, **RAISERROR**..
- nie mogą używać dynamicznego SQL ani tabel tymczasowych; mogą używać zmiennych tabelarycznych
- niedopuszczalne jest użycie **SET**

12. Dynamiczny SQL

Tworząc procedury T-SQL lub Pl/SQL często spotykamy się z problemami wynikłymi ze „sztywności" poleceń języka SQL. Raz zapisane polecenie nie może być modyfikowane, a przez to tracimy elastyczność kodu. Częściowym rozwiązaniem jest stosowanie parametrów, które mogą być użyte w klauzulach **WHERE** i **HAVING**, w poleceniach **SELECT**, **UPDATE**, **DELETE**. Ale już przy **INSERT** nie jesteśmy w stanie różnicować użytych kolumn, nie mówiąc już o takich zagadnieniach jak użycie w parametrach nazw obiektów (tabel, kolumn)

nazwa tabeli podawana w parametrze, czy też „ad hoc" tworzone instrukcje DML lub DDL. Tych ostatnich w ORACLE w ogóle nie wolno używać w blokach Pl/SQL. Rozwiązanie tego problemu stanowi użycie dynamicznego SQL. Stanowi on narzędzie tyleż mocne, co niebezpieczne. Z jednej strony pozwala na tworzenie bardzo elastycznego kodu, z drugiej, jego niewłaściwe użycie naraża aplikacje na ataki z użyciem SQL Injection. Przedstawiamy tylko zarys problemu. Użycie podstawowych rozwiązań w MS SQL Server nie jest bardzo skomplikowane, jednak pewne zaawansowane konstrukcje mogą nastręczać problemy.

Na początek prosty blok T-SQL, w którym wyszukamy dane pracowników zatrudnionych na stanowisku i w dziale, którego nazwa i numer będą podawane w parametrach:

```
DECLARE @Job Varchar(20) = 'MANAGER';
DECLARE @Deptno INT = 20;

SELECT * FROM EMP WHERE JOB = @Job AND DEPTNO = @Deptno;
```

To rozwiązanie oczywiście działa, ale jego „elastyczność" sprowadza się do wyboru stanowisk i numerów działów. Zawsze musimy odczytać dane z wszystkich kolumn, chyba że inaczej skonstruujemy instrukcję **SELECT**.

a) Uruchomienie dynamicznie budowanego polecenia SQL

Znacznie bardziej elastycznym jest wykorzystanie możliwości uruchomienia polecenia SQL, budowanego dynamicznie.

Przykład III.12.1

Utwórz procedurę służącą do odczytywania dowolnie wybranych danych pracowników z tabeli Emp na podstawie ich stanowiska i numeru działu.

Najpierw rozwiązanie w postaci skryptu:

```
DECLARE @sqlCommand varchar(1000);
DECLARE @columnList varchar(75);
DECLARE @Job varchar(20);
DECLARE @Deptno INT;

SET @columnList = 'empno, ename, sal';
SET @Job = '''MANAGER''';
SET @Deptno = 20;
SET @sqlCommand = 'SELECT ' + @columnList + ' FROM emp WHERE job = ' + @Job
                + ' AND deptno = ' + Cast(@Deptno AS Varchar);

EXEC (@sqlCommand);
```

... a następnie procedury:

```
CREATE PROCEDURE Dynamic_SQL @Columnlist Varchar(75), @Job Varchar(10), @Deptno Int
AS
DECLARE @sqlcommand Varchar(100);
BEGIN
SET @sqlcommand = 'SELECT ' + @Columnlist + ' FROM emp WHERE job = ' + @Job
                + ' AND deptno = ' + CAST(@Deptno As varchar) + ';';
EXEC (@sqlcommand);
END;
```

I jej wywołanie dla różnych wariantów danych:

```
EXEC Dynamic_SQL 'empno, ename, sal', '''Manager''', 20
EXEC Dynamic_SQL 'empno, ename, hiredate, comm', '''Clerk''', 10
```

W tym przypadku poprzez parametry przekazywana jest nie tylko nazwa stanowiska i numer departamentu, dla którego odczytujemy dane pracowników, ale też lista kolumn, z których te dane są odczytywane. W przeciwieństwie do przykładu z początku tego rozdziału, w tym rozwiązaniu poprzez parametry dostarczane są nie tylko wartości odczytywane z tabel, ale również nazwy tabel i kolumn.

Należy zwrócić uwagę na konieczność użycia dodatkowych pojedynczych cudzysłowów wyróżniających wartości zmiennych tekstowych (tutaj słowo *MANAGER*). Konieczność ich użycia wynika ze sposobu przekazania wartości tekstowej do procedury (czyli odróżnienia wartości tekstowych od tekstu definiującego całą operację). Nie jest to konieczne przy przekazywaniu wartości liczbowych.

b) Polecenia SQL uruchamiane procedurą sp_executesql

Wydajniejszą (zgodnie z dokumentacją serwera) metodą uruchomienia dynamicznie tworzonego polecenia SQL jest użycie procedury systemowej **`sp_executesql`**. Poniżej przykład.

Przykład III.12.2

Rozwiąż zadanie z poprzedniego przykładu korzystając z procedury sp_executesql.

```
DECLARE @sqlCommand nVarchar(1000);
DECLARE @columnList Varchar(75);
DECLARE @Job Varchar(75);
DECLARE @deptno INT;
SET @columnList = 'empno, ename, sal';
SET @Job = 'MANAGER';
SET @Deptno = 20;
SET @sqlCommand = 'SELECT ' + @columnList
                  + ' FROM emp WHERE job = @Jb AND deptno = @D_no';
EXECUTE sp_executesql @sqlCommand, N'@Jb nvarchar(75), @D_no Int',
                      @Jb = @Job, @d_no = @deptno;
```

I to samo rozwiązanie, ale w postaci procedury:

```
CREATE PROCEDURE Dynamic_SQL_1 @Columnlist Varchar(75), @Job Varchar(10), @Deptno Int
AS
BEGIN
DECLARE @sqlCommand nVarchar(1000) = 'SELECT ' + @Columnlist +
                                     ' FROM emp WHERE job = @Jb AND deptno = @D_no;';
EXECUTE sp_executesql @sqlCommand, N'@Jb nvarchar(75), @D_no Int',
                                   @Jb = @Job, @d_no = @deptno;
END;
```

I jej wywołanie dla różnych wariantów danych:

```
EXEC Dynamic_SQL_1 'empno, ename, sal', 'Manager', 20
EXEC Dynamic_SQL_1 'empno, ename, hiredate, comm', 'Clerk', 10
```

Procedura systemowa **sp_executesql** posiada trzy parametry. Parametr pierwszy (w naszym przykładzie @sqlCommand) to zmienna lub stała zawierająca polecenie SQL. Jego typem musi być Unicode.

Argument drugi to tekst Unicode zawierający definicję wszystkich parametrów użytych w składni polecenia SQL (@sqlCommand) wraz z typami danych. Jeżeli w poleceniu SQL nie są używane parametry, ten argument nie jest wymagany. Jego wartością domyślną jest NULL.

Wreszcie trzeci argument to ciąg podstawień wartości parametrów użytych w definicji polecenia SQL. Jeżeli w definicji polecenia nie ma parametrów, ten ciąg podstawień także zostaje pominięty. Jak widać przy porównaniu dwóch różnych realizacji dynamicznie tworzonego polecenia SQL, w przypadku użycia procedury **sp_executesql** unikamy konieczności „ubierania" wartości tekstowych parametrów w dodatkowe cudzysłowy, co pozwala na łatwiejsze kontrolowanie ich wartości.

Podsumowanie części III

W tej części zostały omówione rozwiązania dostępne w T-SQL, których użycie może ułatwić rozwiązanie niektórych problemów, ale nie decydują one o możliwości ich rozwiązania. Również w procesie poznawania T-SQL początkowo można je pominąć.