

Systemy Baz Danych

Wykład VII

*Transact SQL (T-SQL) język proceduralny
serwera MS SQL Server – cz. 2*

Materiał wykładu VII

Wykład VII stanowi kontynuację wykładu poprzedniego i w całości jest poświęcony językowi Transact SQL, czyli języka proceduralnego MS SQL Server.

W wykładzie poprzednim została omówiona składnia, deklaracja zmiennych, podstawowe konstrukcje programistycznych, jakimi są instrukcje warunkowe i pętle, a także kursory.

W tym wykładzie zostanie omówione użycie poznanych wcześniej konstrukcji do tworzenia obiektów bazy danych jakimi są procedury składowane i wyzwalacze, a także obsługa błędów. Na końcu wykładu zostały przedstawione wybrane typy danych i wbudowane funkcje T-SQL.

Wykład jest przeznaczony dla przedmiotów:

Systemy Baz Danych – studia inżynierskie na Wydziale Informatyki PJWSTK,

Komunikacja z Bazami Danych – studia podyplomowe na Wydziale Informatyki
PJWSTK

Procedury składowane

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 dialekту 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,
- wprowadzenie reguł bezpieczeństwa – aplikacja kliencka ma prawo uruchomić procedurę, a nie wykonywać dowolne polecenia bezpośrednio na tabelach,
- zmniejszenie liczby interakcji z bazą danych – jedna procedura może wywoływać wiele operacji na bazie.

Procedury składowane

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).

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 komplikowane, tworzony jest optymalny plan dostępu do danych.

W T-SQL procedury (a także wyzwalacze) mogą odwoływać się do tabel, które na etapie komplikowania jeszcze nie istnieją.

Procedury składowane - składnia

Składnia polecenia tworzącego procedurę wygląda następująco:

```
CREATE PROCEDURE Nazwa_procedury  
    lista parametrów  
AS  
    instrukcje Transact SQL
```

Przy zmianie procedury słowo **CREATE** zastępowane jest słowem **ALTER**.

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 „zewnętrz” do procedury. Przy deklaracji parametru tego rodzaju, słowo **INPUT**, jako domyślne jest pomijane.

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

Procedury składowane - parametry

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:

```
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 ;
```

Procedury składowane - parametry

Wywołanie procedury z poprzedniego przykładu, wraz z wynikami:

```
EXEC Dept_Job
```

Ename	job	deptno
-----	-----	-----
CLARK	MANAGER	10

```
EXEC Dept_Job 'CLERK'
```

Ename	job	deptno
-----	-----	-----
MILLER	CLERK	10

```
EXEC Dept_Job Default, 20
```

Ename	job	deptno
-----	-----	-----
JONES	MANAGER	20

Procedury składowane - składnia

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.

Procedury T-SQL mogą zwracać wyliczone w nich wartości na trzy sposoby:

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

Procedury składowane

Result set to wynik działania instrukcji **SELECT**, pojawiający się na pulpicie aplikacji MANAGEMENT STUDIO, albo odczytywany wprost przez aplikację kliencką, wywołującą procedurę.

Przykład

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 odczytane z tabeli **EMP** wartości przez Result set, czyli zestaw rekordów będących wynikiem wykonania ostatniej instrukcji **SELECT** w procedurze.

Procedury składowane

Przykład

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
@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.

Procedury składowane

Poprzez procedury mogą być wykonywane operacje DML.

Przykład

Utwórz procedurę dopisującą nowego pracownika do tabeli **EMP**.

```
CREATE PROCEDURE emp_add
    @ename Varchar(20)
    ,@sal Money = 1100
    ,@deptno Int = 10
AS
    INSERT INTO EMP (EMPNO, ENAME, SAL)
    SELECT ISNULL(Max(empno), 0) + 1, @ename ,@sal, @deptno
    FROM EMP;
```

I wywołanie procedury z domyślnymi wartościami parametrów:

```
EXEC emp_add 'Kowalski'
```

Procedury składowane – 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 wewnętrz procedury. Wartości wyliczone w procedurze były przekazywane na zewnątrz w postaci Result set – zestawu rekordów przekazywanych do aplikacji – np. MANAGEMENT STUDIO.

Jak wcześniej wspomniano, wartości wyliczone wewnętrz procedury, mogą zostać przekazane poza procedurę przy użyciu parametrów typu OUTPUT.

Przykład przedstawiono na następnym slajdzie.

Procedury składowane – parametr OUTPUT

Przykład

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

```
CREATE PROCEDURE Change_sal  
    @Empno INT  
    , @New_sal Money OUTPUT  
    , @Percent INT = 20  
AS  
BEGIN  
    SELECT @New_sal = Sal + Sal * @Percent / 100  
    FROM Emp  
    WHERE Empno = @Empno;  
    UPDATE Emp SET Sal = @New_sal  
    WHERE Empno = @Empno;  
END;
```

Procedury składowane – parametr OUTPUT

Aby wywołać procedurę Change_sal, pobrać i wyświetlić wartość z parametru wyjściowego, należy zadeklarować zmienną, na którą zostanie przekazana wartość przechowywana przez zmienną typu OUTPUT w procedurze. Tutaj ta zmienna, zewnętrzna w stosunku do procedury, została dla odróżnienia od nazwy zmiennej wewnętrz procedury, nazwana @New_sal_out. Jednak nic nie stoi na przeszkodzie, aby w różnych procedurach i blokach używać tych samych nazw zmiennych i parametrów.

```
DECLARE @New_sal_out Money;  
EXECUTE Change_sal 7369, @New_sal_out OUTPUT;  
PRINT @New_sal_out;
```

Procedury składowane – Return

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

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

Ten sposób odwołania prezentuje przykład na następnym slajdzie.

Procedury składowane – Return

Przykład

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;
```

Wywołanie procedury i odczytanie wartości zwracanej przez procedurę wygląda następująco:

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

Procedury składowane – podsumowanie

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, 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**,
- w instrukcjach **SELECT** nie powinny pojawiać się funkcje, gdyż wymaga to wyliczenia ich wartości dla każdego wiersza,
- nie należy używać konstrukcji **SELECT ***

Procedury składowane – podsumowanie

Uwagi dotyczące tworzenia procedur – c.d.

- 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 wewnętrz 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.

Triggery (wyzwalacze)

Wyzwalacze są to procedury, tworzące oddzielną klasę obiektów bazy danych. Wyzwalacze są procedurami szczególnymi. Każdy wyzwalacz musi być powiązany z jedną tabelą bazy danych, ale z każdą tabelą może być powiązanych kilka wyzwalaczy. Wyzwalacze są uruchamiane (często mówimy „odpalane”) automatycznie przez SZBD po zajściu odpowiedniego zdarzenia dotyczącego tabeli, z którą wyzwalacz jest związany. Zdarzenia uruchamiające wyzwalacz są definiowane w treści wyzwalacza.

Wyzwalacze służą do:

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

Wyzwalacze zostały wprowadzone do Standardu SQL:1999.

Sposób definiowania i wykorzystania wyzwalaczy w MS SQL (Transact SQL) i ORACLE (PL/SQL) znacznie się różni! Składnia opisana na dalszych slajdach dotyczy tylko Transact SQL.

Triggery (wyzwalacze)

Składnia polecenia tworzącego wyzwalacz wygląda następująco:

```
CREATE TRIGGER Nazwa_wyzwalacza
ON nazwa_tabeli
FOR instrukcje_DML
AS
instrukcje Transact SQL
```

Instrukcjami DML które mogą uruchomić wyzwalacz są:

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

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

Triggery (wyzwalacze)

Wyzwalacze T-SQL uruchamiane są PO zakończeniu wykonywania instrukcji DML, która uruchamia wyzwalacz. Zatem tabela, z którą związany jest wyzwalacz, w momencie jego uruchomienia jest już zmieniona przez instrukcję DML.

Wiersze, które zostały zmienione przez instrukcję umieszczane są w tabelach tymczasowych o nazwach **INSERTED** i **DELETED**. W przypadku instrukcji **INSERT** nowe wiersze znajdują się w tabeli **INSERTED**, w przypadku instrukcji **DELETE** usunięte wiersze znajdują się w tabeli **DELETED**, w przypadku instrukcji **UPDATE** wiersze 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 dostępna do modyfikacji jest ta tabela, na której wyzwalacz jest zdefiniowany.

Wyzwalacze mogą być wyłączone i włączane instrukcją:

```
ENABLE / DISABLE TRIGGER nazwa_wyzwalacza ON nazwa_tabeli;
```

Triggery (wyzwalacze)

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

Wyzwalacze są wykonywane w ramach transakcji, w której została uruchomiona instrukcja która zainicjowała działanie wyzwalacza. Wewnątrz wyzwalacza może zostać umieszczona instrukcja **ROLLBACK**, która wycofa całą transakcję.

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 wyzwalacz zostanie anulowany.

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!).

Triggery (wyzwalacze)

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 bajtu 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.

Triggery (wyzwalacze)

Przykład

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 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.

Triggery (wyzwalacze)

Przykład

Utwórz wyzwalacz, który nie dopuści do przypisania pracownikowi płacy niższej niż 100. 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;
```

Triggery (wyzwalacze)

Wyzwalacz z poprzedniego 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 pokazane jest na slajdzie następnym.

Triggery (wyzwalacze)

Przykład

```
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 idywidualnie traktować każdy zmieniany wiersz, trzeba użyć kursora. Ponadto warto rozdzielić wyzwalacz na dwa niezależne – jeden obsługujący **INSERT**, drugi **UPDATE**.

Triggery (wyzwalacze)

Przykład

```
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;
```

Triggery (wyzwalacze) INSTEAD OF

Istnieje jeszcze jeden rodzaj wyzwalaczy, który pozwala zrealizować postulat Codd'a pełnego dostępu do danych poprzez perspektywy. Jak wiadomo istnieją silne ograniczenia dotyczące możliwości operowania danymi poprzez perspektywy. Omówione zostały 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 T-SQL i instrukcji SQL, zdefiniowanych w treści wyzwalacza, a operujących indywidualnie na tabelach, które wchodzą w skład definicji perspektywy.

Składnia tworząca taki wyzwalacz wygląda następująco:

```
CREATE TRIGGER IO_TR1 ON nazwa_perspektywy  
INSTEAD OF INSERT|UPDATE|DELETE  
AS  
BEGIN  
    Instrukcje SQL i T-SQL  
    (...)  
END;
```

Triggery (wyzwalacze) INSTEAD OF

Przykład

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.

Triggery (wyzwalacze) INSTEAD OF

```
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;
```

Triggery (wyzwalacze) INSTEAD OF

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
```

Triggery (wyzwalacze) - podsumowanie

Wyzwalacze to bardzo silne narzędzie w ręku programisty baz danych – może nawet za silne. 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.

Obsługa błędów – instrukcja Raiserror

Instrukcją wywołującą (podnoszącą) błąd jest Raiserror.

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

Gdzie:

Message - dowolny tekst (komunikat błędu) lub zmienna tekstowa

Severity - liczba z przedziału 0-25 (przy czym użytkownik może używać wartości z przedziału 0-18, sysadmin 19 - 25)

State - liczba z przedziału 1-127

Wartości parametrów **Severity** i **State** są przekazywane do aplikacji klienta, dzięki czemu błędy mogą być rozróżniane i obsługiwane w różny sposób.

Umieszczona w kodzie procedury instrukcja zwraca komunikat o błędzie (zadeklarowany w parametrze **Message**) oraz **Severity** i **State**. Dla wartości **Severity** > 10 następuje przerwanie realizacji kodu procedury i (ewentualnie) przekazanie sterowania do bloku **CATCH**.

Obsługa błędów – bloki TRY i CATCH

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 o wartości **Severity > 10** sterowanie zostaje przekazane do bloku **CATCH** i wykonywane są dalej instrukcje zawarte w tym bloku.

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

Przekazanie sterowania do bloku **CATCH** może nastąpić zarówno po podniesieniu błędu instrukcją **RAISERROR**, jak też po podniesieniu przez system bazy danych błędu o **Severity > 10**.

Obsługa błędów – bloki TRY i CATCH

Przykład

Poniższy przykład pokazuje, jak przebiega sterowanie wykonaniem procedur, w sytuacji podniesienia błędu z różnymi wartościami **Severity**. Przykład powinien być wykonany gdyż dopiero w trakcie jego realizacji odpowiednie komunikaty informują o przekazaniu (lub nie) sterowania z bloku **TRY** do bloku **CATCH** w zależności od wartości **Severity** związanej z podnoszonym błędem.

```
CREATE PROCEDURE Error_test @Level Int
AS
DECLARE @StrInfo Varchar(30)
BEGIN TRY
IF @Level = 1
BEGIN
Set @StrInfo = 'Błąd niekrytyczny'
Raiserror (@StrInfo, 1, 10)
Print 'Przeszliśmy błąd o numerze 10'
END
```

Obsługa błędów – bloki TRY i CATCH

Przykład – cd

```
IF @Level = 2
BEGIN
    Set @StrInfo = 'Błąd krytyczny,';
    Raiserror (@StrInfo, 11, 20);
    Print 'Czy przeszliśmy błąd o numerze 20 ?,';
END
END TRY
BEGIN CATCH
DECLARE @ErrMsg NVARCHAR(100), @ErrSev INT, @ErrSt INT;
SELECT @ErrMsg = ERROR_MESSAGE()
    ,@ErrSev = ERROR_SEVERITY()
    ,@ErrSt = ERROR_STATE();
Print @ErrMsg + ' Severity = ' + Cast(@ErrSev As Varchar(3)) +
    ' State = ' + Cast(@ErrSt As Varchar(3));
END CATCH;
```

Wybrane typy zmiennych MS SQL Server 2012

Typy daty i czasu

Typ	Format	Zakres
Time	hh:mm:ss [.nnn]	0.00 – 23.59.59.9999
Date	YYYY-MM-DD	0001 – 01 – 01 do 9999-12-31
Datetime	YYYY-MM-DD hh:mm:ss	1753-01-01 do 9999-12-31

Typy napisowe

Typ	Format	Zakres
Char (n)	Znaki ASCII	1 – n znaków; max. 8000
Varchar (n)	Znaki ASCII	1 – n znaków; max. 8000
nChar (n)	Unicode	1 – n znaków; max. 4000
nVarchar (n)	Unicode	1 – n znaków; max. 4000
Varchar (max)	Znaki ASCII	Max 2GB

Wybrane typy zmiennych MS SQL Server 2012

Typy numeryczne dokładne

Typ	Format	Zakres
Int		-2^31 do 2^31
Bigint		-2^63 do 2^63
Numeric	(p[,s])	-2^31 (-2,147,483,648) to 2^31-1
Decimal	(p[,s])	-2^31 (-2,147,483,648) to 2^31-1
Money		-922,337,203,685,477.5808 do 922,337,203,685,477.5807

Typy numeryczne przybliżone

Typ	Format	Zakres
Float		1 - n znaków; max. 8000
Real		- 3.40E + 38 to -1.18E - 38, 0 and 1.18E - 38 to 3.40E + 38

Wybrane funkcje MS SQL Server 2012

Funkcje konwersji typów danych

`CAST (wyrażenie AS Typ_Danych [(długość)])`

`CONVERT (Typ_Danych, wyrażenie [, styl_daty])`

np.:

`CONVERT (Varchar, HIREDATE, 13)` zwraca datę w formacie **17 Dec 1980**

`CONVERT (Varchar, HIREDATE, 13)` zwraca datę w formacie **1980-12-17**

Funkcje daty i czasu

`GETDATE () , CURRENT_TIMESTAMP`

obie funkcje zwracają aktualną datę i czas z dokładnością do milisekundy

`DATEPART (datepart, Data)`

zwraca liczbę odpowiadającą części daty określonej przez symbol datepart

np.:

`DATEPART (mm, '2011-11-12')`

zwraca liczbę **11**

Wybrane funkcje MS SQL Server 2012

`DATEADD (datepart, liczba, Data)` zwraca datę zmienioną o liczbę części daty określonej przez symbol datepart

np.:

`DATEADD (mm, 2 , '2013-11-12')` zwraca datę **2014-01-12**

`DATEDIFF (datepart, startdate, enddate)` zwraca liczbę części daty określonej przez datepart pomiędzy startdate i enddate

np.:

`DATEDIFF (dd, '2013-11-12', '2013-12-21')` zwraca liczbę **11**

`DAY (Data), MONTH (Data), YEAR (Data)` zwracają liczby odpowiadające podanym częściom daty

`ISDATE (wyrażenie)` zwraca **1** jeżeli wyrażenie jest datą, **0** jeśli nią nie jest

`DATENAME (datepart, Data)` zwraca nazwę części daty (tekst)

np.:

`DATENAME (dd, '2013-11-12')` zwraca **November**

Wybrane funkcje MS SQL Server 2012

Funkcje operujące na tekstach

LEN (*wyrażenie_tekstowe*) zwraca liczbę znaków w tekście

LTRIM | **RTRIM** (...) zwraca wyrażenie tekstowe pozbawione spacji z lewej / prawej strony wyrażenia

UPPER | **LOWER** (*wyrażenie_tekstowe*) zwraca wyrażenie zapisane wielkimi / małymi literami

LEFT | **RIGHT** (*wyrażenie_tekstowe*, *wyrażenie_liczbowe*) zwraca określoną przez *wyrażenie_liczbowe* liczbę znaków z lewej | prawej strony wyrażenia tekstowego

SUBSTRING (*wyrażenie_tekstowe*, *start*, *długość*) zwraca część wyrażenia tekstowego, określonej *długości*, poczawszy od znaku *start*

REVERSE (*wyrażenie_tekstowe*) zwraca znaki *wyrażenia_tekstowego* w odwróconej kolejności („czytane od tyłu”)

Wybrane funkcje MS SQL Server 2012

REPLACE (wyrażenie_tekstowe, Tekst1, Tekst2)	zwraca wyrażenie_tekstowe z tekstem Tekst1 zamienionym przez Tekst2
ASCII (wyrażenie_tekstowe)	zwraca kod ASCII pierwszego z lewej znaku wyrażenia
CHAR (wyrażenie_tekstowe)	zwraca znak ASCII odpowiadający wartości wyrażenia

Funkcje systemowe

@@ROWCOUNT	zwraca liczbę rekordów, na których operowała ostatnia instrukcja SQL
@@ERROR	zwraca numer błędu podniesionego w ostatniej instrukcji T-SQL
@@IDENTITY	zwraca wartość wygenerowaną podczas ostatniej operacji INSERT na kolumnie z właściwością IDENTITY

Pełna lista funkcji T-SQL v.2012 znajduje się pod adresem:

<http://technet.microsoft.com/en-us/library/ms174318.aspx>