

# Systemy Baz Danych

## Wykład VI

*Transact SQL (T-SQL) język proceduralny  
serwera MS SQL Server- cz. 1*

# Wstęp

Tematem tego wykładu, a także kilku kolejnych, będą języki proceduralne serwerów relacyjnych baz danych - MS SQL Server i ORACLE. Zostanie omówiona składnia obu tych języków oraz przedstawione dostępne konstrukcje programistyczne.

Po opanowaniu materiału prezentowanego w tym cyklu wykładów, student powinien umieć samodzielnie tworzyć procedury składowane i wyzwalacze.

Pierwszy wykład zawiera omówienie podstawowej składni, deklaracji zmiennych, podstawowych konstrukcji programistycznych, jakimi są instrukcje warunkowe i pętle, a także kursorów - konstrukcji charakterystycznej dla pracy z zestawami rekordów (*Result sets*).

Wykład jest przeznaczony dla studentów i słuchaczy przedmiotów:

**Systemy Baz Danych** – studia inżynierskie na Wydziale Informatyki PJWSTK,

**Komunikacja z Bazami Danych** – studia podyplomowe na Wydziale Informatyki  
PJWSTK

# Język proceduralny

*Język proceduralny (ang. procedural language) – język programowania umożliwiający tworzenie oprogramowania w postaci programu głównego oraz wielu procedur (lub podprogramów), z których każda realizuje określoną funkcję i może być wywoływana wielokrotnie przez program główny.* (Cyt. Wikipedia).

Zapoznając się z językiem SQL poznawaliśmy sposoby wykonywania pojedynczych instrukcji tego języka. Jednak w praktyce na ogół istnieje konieczność jednoczesnego wykonania większej liczby instrukcji, przekazania wyników pomiędzy tymi instrukcjami, sprawdzenia warunków ich wykonania i uzależnienia od uzyskanych wyników dalszego postępowania z danymi. Składnia „czystego” SQL nie umożliwia wykonywania tak zaawansowanych operacji.

Serwery baz danych oferują rozszerzenie języka SQL o konstrukcje typowe dla języków programowania. Rozszerzenie to jest zdefiniowane w standardzie języka SQL już od wersji z roku 1999 i nosi nazwę SQL/PSM. Producenci serwerów baz danych implementują jednak własne rozwiązania i tak, Microsoft SQL Server oferuje język Transact-SQL, a ORACLE posiada swój PL/SQL.

# Transact SQL

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. Obecnie jest wbudowywany w kolejnych wersjach serwera MS SQL i wraz z nim rozwijany. W dalszej części wykładu będzie zasadniczo omawiana wersja MS SQL Server 2012.

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

# 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, Sqldeveloper, SQL+), 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.

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.

# Blok anonimowy

Konstrukcja bloku anonimowego w T-SQL jest 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 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 byćkończony 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 wystąpienia słowa **GO**.

# Odczytanie wyników działania instrukcji

Wynik działania instrukcji **SELECT** jest wypisywany na pulpit aplikacji MS Management Studio w postaci zestawu odczytanych rekordów (*Result set*).

W przypadku poprawego wykonania instrukcji DML wyświetlany jest komunikat o liczbie zmodyfikowanych wierszy np. (5 row(s) affected). Komunikat ten jest też przekazywany do aplikacji. Może zostać wyłączony polecienniem SET NOCOUNT ON (i ponownie włączony SET NOCOUNT OFF).

Poleceniem wypisania wyniku operacji lub komunikatu jest **PRINT**, np.

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

Identyczny efekt da zastąpienie słowa **PRINT** przez **SELECT**. Ale **PRINT** pozwala tylko wypisać wartości na pulpit, podczas gdy **SELECT** umożliwia też inne operacje.

# 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  
SELECT * FROM emp; --polecenie odczytania danych
```

# Zmienne

Zmienna jest jednym z podstawowych elementów niezbędnych przy programowaniu kodu. Rola zmiennej 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. Nazwa każdej zmiennej musi rozpoczynać się znakiem `@`. W celu zadeklarowania zmiennej używana jest instrukcja **DECLARE** a po niej nazwa zmiennej i jej typ.

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 (**Integer**), zmienoprzecinkowymi (**Decimal**, **Money**), daty i czasu (**Date**, **Time**, **Datetime**).

# Deklaracja zmiennych

## Przykład 1

Deklaracja jednej zmiennej:

```
DECLARE @nazwisko Varchar(30);
```

W pojedynczej instrukcji DECLARE można zadeklarować wiele zmiennych:

## Przykład 2

Deklaracja wielu zmiennych w jednej instrukcji:

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

Instrukcja **DECLARE** może pojawiać się wielokrotnie, w dowolnym miejscu kodu. Jedynym wymogiem jest, aby deklaracja zmiennej poprzedzała jej użycie (odwołanie się do niej).

# Deklaracja zmiennych

Wraz z deklaracją zmiennej można nadać jej wartość początkową (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, bez względu na typ, są NULL.

# Zmienne systemowe

Nazwy zmiennych systemowych zaczynają się od podwójnego znaku @. Nie należy ich deklarować, są one *Read Only*. Służą do przekazania (odczytania z nich) istotnych informacji przechowywanych lub wyliczanych przez serwer bazy danych. Kilka najczęściej używanych:

**@@Version** - zwraca информацию 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, -1 porażka, rekord niepodstawiony).

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

# Instrukcja SELECT i SET w T-SQL

Instrukcja **SELECT** w bloku T-SQL ma rozszerzone funkcje w stosunku do „czystego” SQL. Służy do podstawienia wartości na zmienną lub kilka zmiennych. Nie musi być 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 w T-SQL

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

```
SELECT @Imie = Imie  
      , @Nazwisko = Nazwisko  
      , @DataRekrutacji = Data_rekrutacji  
FROM Osoby  
WHERE Osoba_Id = 1234;
```

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

Jeżeli zapytanie nic nie zwróci, 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. Ta właściwość może być przyczyną istotnych błędów!

# Przykłady prostych programów T-SQL

## Przykład 3:

Zadeklarujemy zmienną, a następnie na tą zmienną zapiszemy liczbę pracowników zapisanych w tabeli **Emp**.

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

Należy pamiętać o konieczności dokonywania jawnej konwersji danych na typ napisowy przy konkatenacji (funkcje **Cast** i **Convert**).

# Przykłady prostych programów T-SQL

## Przykład 4:

W tym przykładzie usuniemy dział o podanej nazwie, ustawiając wartość Deptno pracowników tego działu na NULL. W komunikacie wyświetlimy liczbę 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';
```

# Instrukcje sterujące w T-SQL

W dotychczasowych rozważaniach i przykładach zakładaliśmy wykonanie wszystkich instrukcji programu po kolejno – 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**. Zostaną one omówione na kolejnych slajdach.

# 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**.

# Instrukcja warunkowa 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.

# Instrukcja warunkowa IF

## Przykład 5

Sprawdzimy, czy planowana podwyżka zarobków wszystkich pracowników nie przekracza dysponowanego budżetu. 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;
```

# Instrukcja warunkowa IF

Jak widać na przykładzie 5, 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 6 pokazuje kod z przykładu 5 tak właśnie zmodyfikowany.

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

# Instrukcja warunkowa IF

## Przykład 6

Modyfikacja przykładu 5 – 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;
```

# Instrukcja warunkowa IF

Niektóre języki programowania oferują konstrukcję **ELSEIF**, pozwalającą 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 przykład 7.

# Instrukcja warunkowa IF

## Przykład 7

Modyfikacja przykładu 5 – dodanie kolejnego warunku IF

```
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 place' + CAST(@@Rowcount AS
Varchar) + ' najmniej zarabiającym pracownikom.'
END;
ELSE
    SET @Info = 'Nie podwyższono płac, bo nie ma z czego.';
PRINT @Info;
```

# IF EXISTS

W T-SQL została zaimplementowana bardzo wygodna instrukcja, będąca modyfikacją instrukcji warunkowej:

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

Jeżeli instrukcja **SELECT** zwraca **cokolwiek** (**Nawet NULL**), cała instrukcja zwraca **TRUE**.

## Przykład:

*Sprawdź, czy istnieją pracownicy nie przypisani do żadnego departamentu.*

```
IF EXISTS (SELECT 1 FROM EMP WHERE Deptno IS NULL)
    PRINT 'Istnieje 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”.

# 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 ogólny przewidują dwa rodzaje pętli – pętlę wykonywaną aż do spełnienia założonego warunku i pętlę wykonywaną założoną 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 inny rodzaj pętli.

# Instrukcja pętli WHILE

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.

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).

# Kursory

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.

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 zależnie od wartości w odczytywanych rekordach.

W takich sytuacjach rozwiązanie stanowi użycie *kursora*.

# Kursory

Kursor jest buforem pamięci, w którym zapisywane są rekordy odczytane przez instrukcję **SELECT**, umożliwiającym dostęp do każdego pojedynczego rekordu i podstawienie na zmienne wartości wyrażeń tego rekordu. W ten sposób uzyskujemy możliwość operowania na pojedynczych wartościach wyrażeń we wszystkich odczytanych (i zbuforowanych) rekordach.

Kursor jest strukturą programistyczną, która może być użyta w dowolnym kodzie T-SQL – w bloku anonimowym, w procedurze składowanej czy w wyzwalaczu.

Należy jednak pamiętać, że kursory nie są cudownym przepisem na rozwiązywanie 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**.

# Kursory - składnia

Aby skorzystać z kursora musimy wykonać kilka kroków, ścisłe 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 oddziennie. 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.

# Kursory – konstrukcja

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. 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.

# Kursory – konstrukcja

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 **@@Fetch\_status** przyjmie wartość -1 i pętla się zakończy.

# Kursory – konstrukcja

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 sama struktura kursoła nie zostaje usunięta i można z niej po raz kolejny skorzystać, wykonując ponownie instrukcję **OPEN**. Instrukcja **CLOSE** musi odnosić się do otwartego kursoła, gdyż w przeciwnym przypadku zostanie wygenerowany błąd: **Cursor is not open**

Instrukcją usuwającą strukturę kursoła jest

**DEALLOCATE**

Usuwa ona odwołania kursoła do tabel, i zwalnia zasoby pamięci dotychczas rezerwowane przez kursoł.

# Kursory – przykład 1

Pierwszy (trywialny) przykład to wypisanie z tabeli EMP na ekran nazwisk i pensji pracowników zarabiających powyżej 2000.

```
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;
```

# Kursory – przykład 1

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 może spowodować niewłaściwe działanie całego kurSORA, lub wejście w nieskończoną pętlę.

Trywialność powyższego przykładu wynika z faktu, iż do odczytania danych można użyć tylko instrukcji **SELECT**, która definiuje kurSOR.

## Kursory – przykład 2

W kolejnym przykładzie kurSOR zostanie wykorzystany w celu odczytania 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.

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**.

## Kursory – przykład 2

```
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;
```

## Kursory – przykład 2

```
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;
```

## Kursory – przykład 2

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, rozwiązuająca 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;
```

# 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** – następny wiersz
- **FETCH PRIOR** – poprzedni wiersz
- **FETCH FIRST** – pierwszy wiersz
- **FETCH LAST** – ostatni wiersz
- **FETCH ABSOLUTE (n)** – wiersz o numerze n
- **FETCH RELATIVE (n)** – wiersz o n wierszy dalej niż bieżący, lub bliżej,  
jeśli  $n < 0$ )

# Wybrane typy zmiennych MS SQL Server 2012

## Typy daty i czasu

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

## Typy napisowe

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

# Wybrane typy zmiennych MS SQL Server 2012

## Typy numeryczne dokładne

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

## Typy numeryczne przybliżone

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