

# SQL Server 2005

*Łukasz Łysik*  
*llysik@gmail.com*

*21 października 2008*

# Zakres prezentacji

- SQL Server Management Studio
- Transakcje
- Lock, deadlocks
- Procedury CLR
- Triggery
- Service Broker
- SQL Server Profiler

# SQL Server Management Studio

The screenshot displays the Microsoft SQL Server Management Studio interface. The main window shows a query editor with the following SQL code:

```
USE AdventureWorks
GO

SELECT * FROM HumanResources.EmployeeAddress
```

The Object Explorer on the left shows the server hierarchy for LUKECOMP\SQL2K5 (SQL Serv). The 'Tables' folder is expanded, showing various tables including 'Person.Address'. The 'Results' pane at the bottom displays the output of the query, which is a table with the following data:

	EmployeeID	AddressID	rowguid	ModifiedDate
1	1	61	77253AEF-8883-4E76-97AA-7B7DAC21A2CD	2004-10-13
2	2	234	7FA5FF71-E97B-457D-B7C4-88FB6566DC40	2004-10-13
3	3	224	3C915B31-7C05-4A05-9859-0DF663677240	2004-10-13
4	4	11387	3DC70CC4-3AE8-424F-8B1F-481C5478E941	2004-10-13
5	5	1	0A4C095C-000E-425E-A103-2E7E7304D0F1	2004-10-13

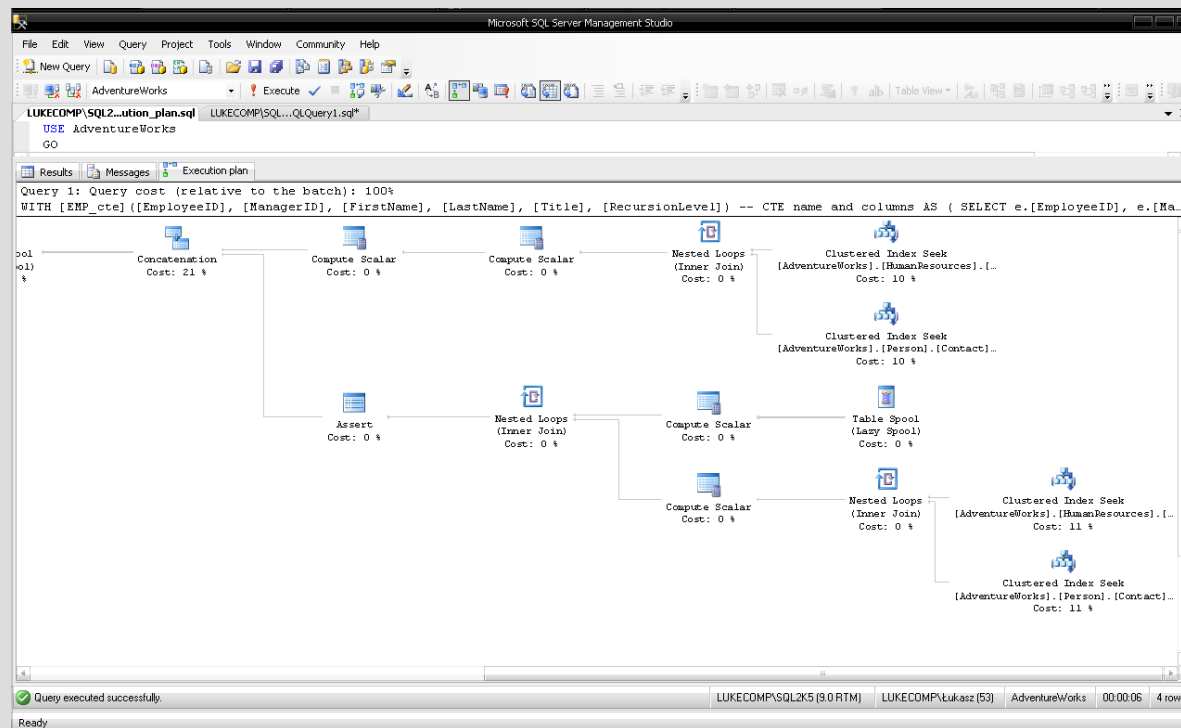
The status bar at the bottom indicates the server is LUKECOMP\SQL2K5 (9.0 RTM), the user is LUKECOMP\Lukasz (51), the database is AdventureWorks, and the session has been running for 00:00:00.

# SQL Server Management Studio

- Przydatne elementy:
  - Object Explorer – zawiera hierarchiczne drzewo z wszystkimi obiektami w bazie danych
  - Query Analyzer – główne okno edycji
  - Result – okno w którym wyświetlane są wyniki zapytań
- Przydatne skróty klawiszowe:
  - F5 – wykonanie kodu (lub zaznaczonego fragmentu)
  - F8 – przywołanie okna Object Explorer
  - Zaznaczenie nazwy procedury, tabeli oraz wciśnięcie Alt + F1 wyświetla informacje o procedurze (nazwy i typy parametrów), tabeli (nazwy kolumn, indeksów, kluczy, itp).
  - Ctrl + r – zamyka okno *Result*.

# SSMS – Execution Plan

- Przy optymalizacji zapytań przydaje się funkcja *Execution Plan*. Włączenie funkcji następuje poprzez wybranie ikony *Include Actual Execution Plan*.
- Wyświetlane zostaje plan wykonania podzapytań w postaci drzewa, razem z procentem czasu wykonania całego zapytania (przykład w pliku *01\_prez\_execution\_plan.sql*).



# Transakcje 1

- Transakcja jest to zbiór operacji, które stanowią pewną całość i powinny być wykonane wszystkie lub żadna z nich.
- Oznacza to, że gdy np. mamy 2 operacje zawarte w transakcji i pierwsza zakończy się sukcesem a druga porażką, pierwsza z nich zostaje anulowana.
- Transakcje można stosować np. w bankach w sytuacji przelania pieniędzy z konta na konto. W jednej tabeli zmniejszamy ilość pieniędzy a w drugiej zwiększamy. W przypadku błędu w którejś z tych operacji albo na obu kontach jest mniej pieniędzy, albo na obu za dużo. Zamknięcie obu operacji w transakcji rozwiązuje problem.

# Transakcje 2 (przykład)

- Do zaprezentowania transakcji utworzymy przykładową tabelę. Należy zwrócić uwagę, że kolumny *liczba\_1* oraz *liczba\_2* nie mogą mieć wartości *NULL*.

```
-- tworzymy tabele
CREATE TABLE sumy
(
  [id] int PRIMARY KEY IDENTITY(1,1) NOT NULL,
  [liczba_1] int NOT NULL,
  [liczba_2] int NOT NULL,
  [suma] int NULL
)
```

# Transakcje 3 (przykład c.d.)

```
-- 1. Rozpoczecie transakcji
BEGIN TRAN

-- 2. Sprawdzenie: tabela pusta
SELECT * FROM sumy

-- 3. Poprawne wstawienie
INSERT INTO sumy(liczba_1,liczba_2) VALUES (12,34)

-- 4. Sprawdzenie: tabela zawiera jeden wiersz
SELECT * FROM sumy

-- 5. Niepoprawne wstawienie (kolumna 'liczba_2' nie moze byc NULL)
INSERT INTO sumy(liczba_1) VALUES(4)

-- 6. Anulowanie transakcji gdy wystapil blad
IF @@ERROR > 0 ROLLBACK TRAN

-- 7. Sprawdzenie: tabela ponownie pusta
SELECT * FROM sumy

-- 8. Zatwierdzenie transakcji gdy nie ma bledu
COMMIT TRAN
```



# Transakcje 4 (przykład c.d.)

- Punkt 1 – rozpoczęcie transakcji
- Punkt 2 – sprawdzenie: tabela powinna być pusta (nic do niej nie wstawiliśmy po utworzeniu)
- Punkt 3 – wstawiamy poprawny rekord
- Punkt 4 – sprawdzenie: rekord faktycznie znajduje się w tabeli
- Punkt 5 – wstawiamy niepoprawny rekord. Kolumna *liczba\_2* ma atrybut *NOT NULL*, dlatego wstawienie wartości tylko do kolumny *liczba\_1* powoduje błąd.
- Punkt 6 – sprawdzamy, czy wystąpił błąd. Anulowanie transakcji (ROLLBACK)
- Punkt 7 – sprawdzenie: anulowanie transakcji spowodowało cofnięcie wszystkich operacji w jej wnętrzu. Tabela jest pusta.
- Punkt 8 – gdyby błąd nie wystąpił nastąpiłoby zatwierdzenie transakcji.

# Transakcje 5 (zagnieżdżanie)

- Transakcje można zagnieżdżać (we wnętrzu jednej transakcji można umieścić kolejne).
- Pomocna jest zmienna globalna `@@TRANCOUNT`, która przyjmuje wartość aktualnego stopnia zagnieżdżenia transakcji.
- Przykład:

```
PRINT @@TRANCOUNT      -- 0

BEGIN TRAN
PRINT @@TRANCOUNT      -- 1

BEGIN TRAN
PRINT @@TRANCOUNT      -- 2

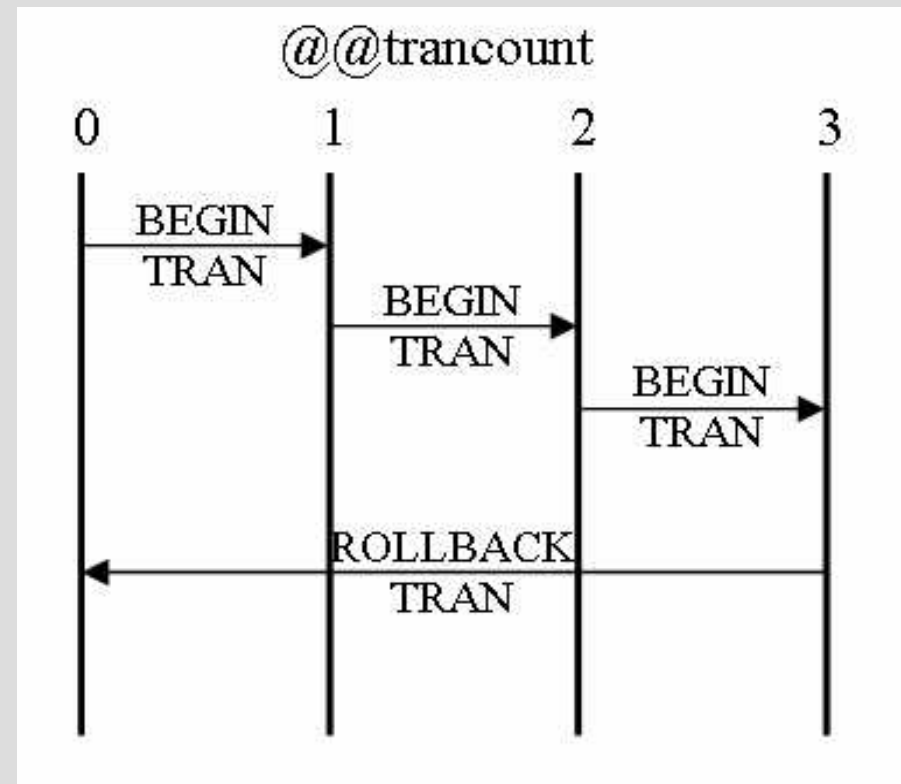
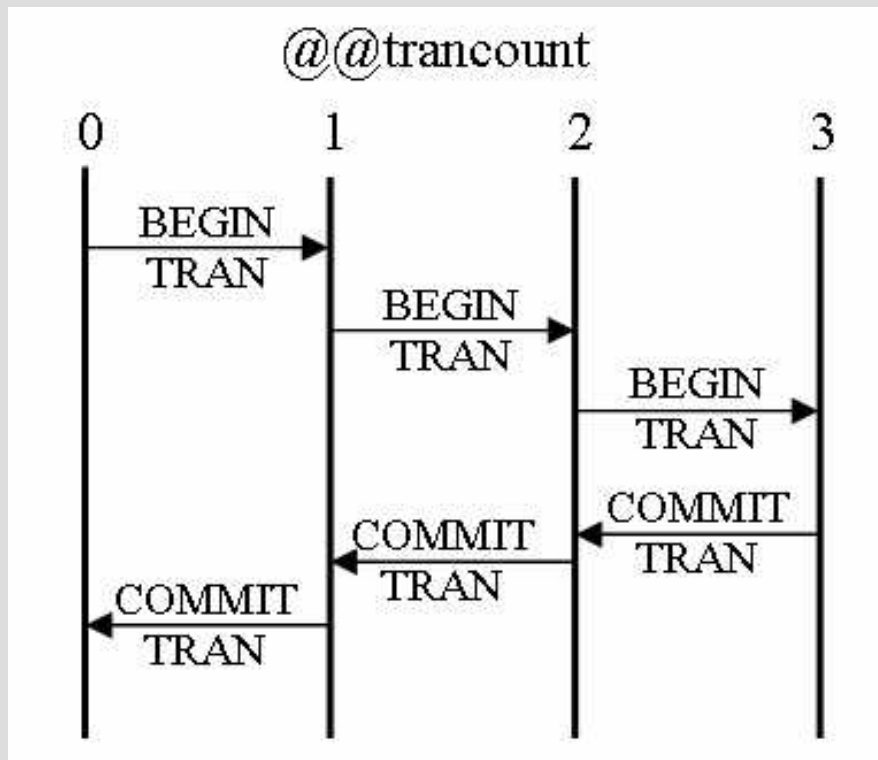
BEGIN TRAN
PRINT @@TRANCOUNT      -- 3

COMMIT TRAN
PRINT @@TRANCOUNT      -- 2

ROLLBACK TRAN
PRINT @@TRANCOUNT      -- 0
```

# Transakcje 6 (zagnieżdżanie c.d.)

- Jak widać z przykładu polecenie COMMIT TRAN powoduje przejście wyżej tylko o jeden poziom. Polecenie ROLLBACK TRAN powoduje przejście od razu do poziomu 0.

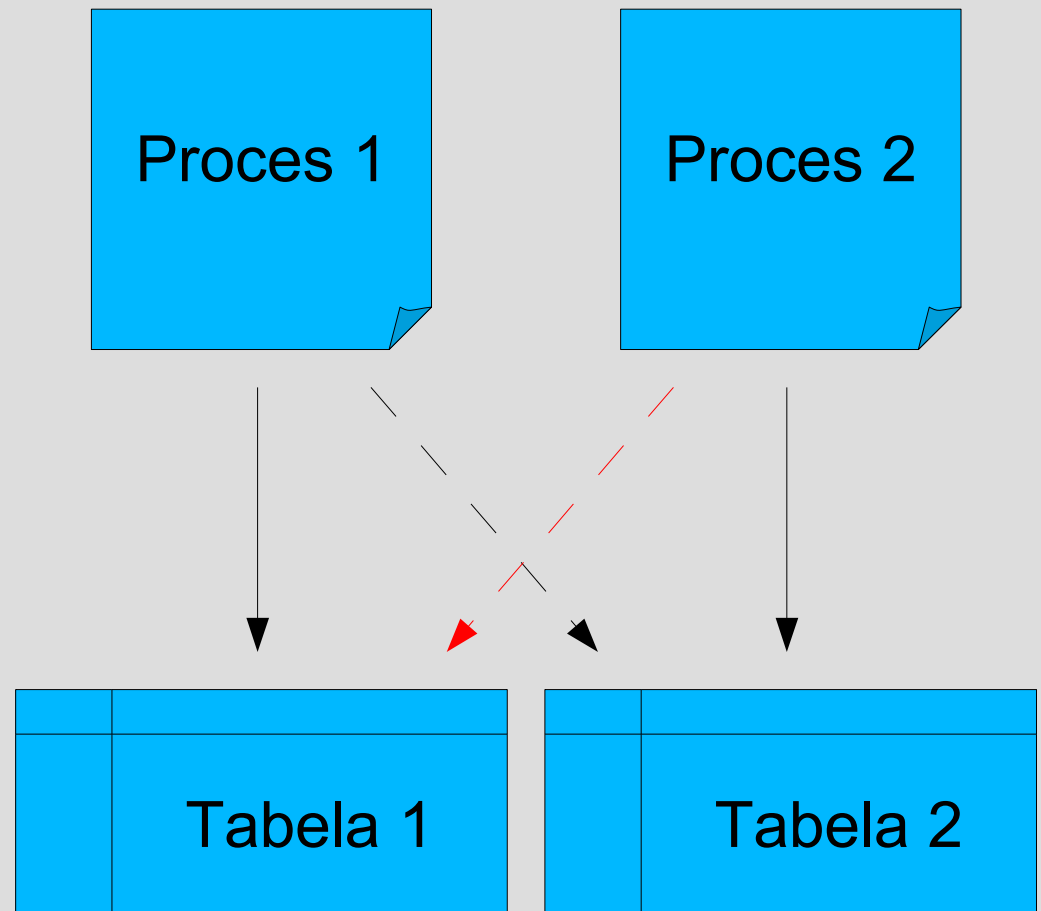


# Lock, deadlock

- Przy pomocy mechanizmu transakcji bardzo łatwo zobaczyć jak powstaje **lock** oraz **deadlock**.

# Lock, deadlock 2 (przykład)

- Proces 1 rozpoczyna transakcję oraz wykonuje zapytanie na tabeli 1 – tabela jest blokowana dla tego procesu.
- Proces 2 rozpoczyna transakcję oraz wykonuje zapytanie na tabeli 2 – tabela jest blokowana dla tego procesu.
- Proces 1 wykonuje zapytanie na tabeli 2 – występuje **lock**. Proces 1 czeka, aż proces 2 zakończy transakcję.
- Proces 2 wykonuje zapytanie na tabeli 1. Proces 2 czeka, aż proces 1 zakończy transakcję – występuje **deadlock**.



# Lock, deadlock 3 (przykład)

- Na potrzeby przykładu utworzone zostaną 2 tabele (kod poniżej).
- Przykład wymaga, aby do tabel miały dostęp 2 osobne procesy. W SQL Server Management Studio każde okno edycji jest osobnym procesem. Dla ułatwienia można otworzyć 2 okna sąsiadujące ze sobą. Uruchamianie wybranego fragmentu kodu z okna edycji można uzyskać przez zaznaczenie tego fragmentu i wciśnięcie klawisza *F5* lub przycisku *Execute*.

```
CREATE TABLE t1
(
    col INT
)

CREATE TABLE t2
(
    col INT
)

INSERT INTO t1 VALUES (1)
INSERT INTO t2 VALUES (2)
```

# Lock, deadlock 4 (przykład)

- Kod wywołujemy fragmentami według numeracji.

```
-- PROCES 1

-- 1. Rozpoczęcie transakcji.
BEGIN TRAN

-- 3. Operacja na tabeli 1
UPDATE t1 SET col = 3

-- 5. Operacja na tabeli 2 - lock
UPDATE t2 SET col = 3
```

```
-- PROCES 2

-- 2. Rozpoczęcie transakcji.
BEGIN TRAN

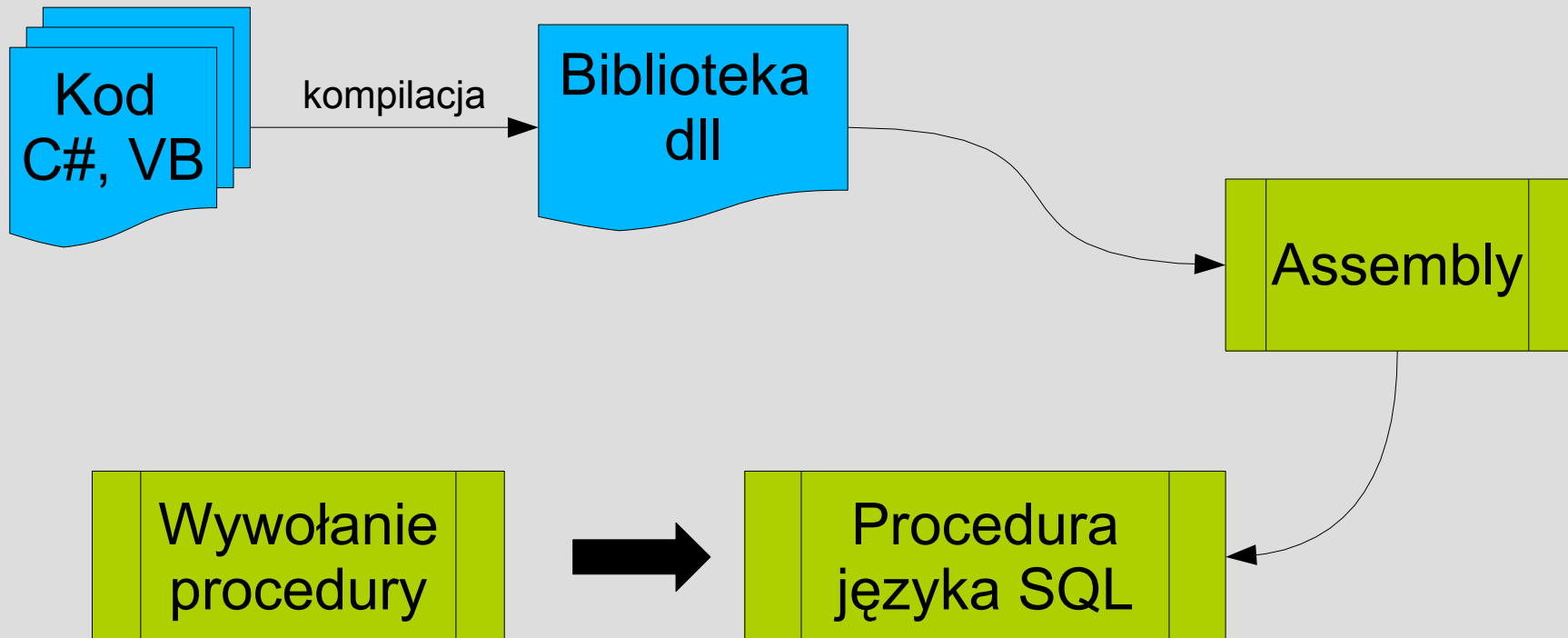
-- 4. Operacja na tabeli 2
UPDATE t2 SET col = 4

-- 6. Operacja na tabeli 1 - deadlock
UPDATE t1 SET col = 4
```

- Gdy powstaje **lock** proces czeka, aż inny proces zakończy działanie i zwolni zasób (tabelę).
- Gdy powstaje **deadlock** zostaje podniesiony wyjątek, który należy obsłużyć. Można zrezygnować z operacji na zajętej tabeli lub spróbować ponownie później.

# CLR – zewnętrzne procedury

- CLR – Common Language Runtime
- Są to procedury zewnętrzne, pisane w językach takich jak C# czy Visual Basic, kompilowane do biblioteki dll, które mogą być wywoływane z poziomu zapytań języka SQL.





# CLR – przykład

- Utworzymy przykładową bibliotekę dll w języku C#. Będzie zawierała funkcję zliczającą wystąpienie danego znaku w ciągu znaków.
- Otwieramy Microsoft Visual C# 2005 Express Edition (darmowa wersja).
- File -> New Project... -> Class Library. Nadajemy nazwę *CountChars*.
- W pliku *Class1.cs* umieszczamy poniższy fragment kodu, po czym kompilujemy wciskając klawisz *F6* (lub menu: Build -> Build solution).

```
namespace CountChars {  
    public class Prezentacja {  
        public static int CountChars(string inputString, char c) {  
            int result = 0;  
            foreach(char character in inputString) {  
                if(character == c)  
                    result++;  
            }  
            return result;  
        }  
    }  
}
```

# CLR – przykład c.d.

- W wyniku kompilacji powstał plik *CountChars.dll*.
- Przechodzimy do Management Studio i uruchamiamy naszą procedurę.

```
-- 1. Tworzymy assembly
CREATE ASSEMBLY AssCountChars
FROM 'F:\dll\CountChars.dll'
GO

-- 2. Tworzymy procedure
CREATE FUNCTION dbo.clr_CountChars(@inputString NVARCHAR(MAX), @char NCHAR )
RETURNS INT
AS EXTERNAL NAME AssCountChars.[CountChars.Prezentacja].CountChars
GO

-- 3. Test funkcji
DECLARE @retVal INT
EXEC @retVal = clr_CountChars 'aa.a.bbb.ccc.dd.eee', '.'

-- 4. Wynik: 5
PRINT @retVal
```

# CLR – przykład 2

- Za pomocą procedur CLR można wysłać wiadomość SMS z poziomu zapytania SQL.
- Kod C# znajduje się w pliku *PlusSend.cs*. Bibliotekę dll oraz procedurę stworzymy analogicznie do poprzedniego przykładu.
- Wymagane jest ustawienie opcji *TRUSTWORTHY* na *ON*, aby nasza funkcja mogła łączyć się z internetem (względy bezpieczeństwa).
- Wywołanie procedury:

```
ALTER DATABASE prezentacja SET TRUSTWORTHY ON  
EXEC clr_PlusSend 'Lukasz', '504123456', 'Wiadomosc testowa z SQL Servera'
```

# Triggery

- Trigger (wyzwalacz) jest to fragment kodu (procedura) wywoływany, gdy w systemie wystąpi pewne zdarzenie, np. wstawienie wiersza do tabeli.
- W systemie SQL Server 2005 występują 2 rodzaje triggerów:
  - DML – Data Manipulation Language – są to triggery wyzwalane po wystąpieniu akcji INSERT, UPDATE, DELETE.
  - DDL – Data Definition Language – wyzwalane przez rodzinę poleceń CREATE, ALTER, DROP.
- Zdarzenie i wyzwolony trigger występują w jednej transakcji – błąd, który wystąpi we wnętrzu triggera spowoduje cofnięcie zdarzenia.

# Triggery DML

- We wnętrzu triggerów można korzystać z tabel *inserted* oraz *deleted*, które zawierają wstawione lub usunięte wiersze.
  - Dla polecenia INSERT – tylko tabela *inserted*.
  - Dla polecenia DELETE – tylko tabela *deleted*.
  - Dla polecenia UPDATE – tabela *deleted* zawiera starą wersję zmienionego rekordu, tabela *inserted* zawiera nową wersję.
- Przykład rozpoznający wykonane polecenie na podstawie tabel *inserted* oraz *deleted*:

```
CREATE TRIGGER trg_sumy_a_ins_upd_del ON sumy
AFTER INSERT, UPDATE, DELETE
AS
BEGIN
    IF EXISTS(SELECT * FROM inserted)
        IF EXISTS(SELECT * FROM deleted)
            PRINT 'Trigger AFTER UPDATE'
        ELSE
            PRINT 'Trigger AFTER INSERT'
    ELSE
        PRINT 'Trigger AFTER DELETE'
END
```

# Triggery DDL

- Utwórzmy trigger DDL, który zabroni usuwania tabel w bazie.

```
CREATE TRIGGER safety
ON DATABASE
FOR DROP_TABLE
AS
BEGIN
    PRINT 'You must disable Trigger "safety" to drop tables!'
    ROLLBACK
END

-- TEST

-- 1. Tworzymy tabele
CREATE TABLE tmp_table (col1 int)

-- 2. Próba usunięcia tabeli - porażka
DROP table tmp_table

-- 3. Wyłączenie triggera
DISABLE TRIGGER safety ON DATABASE

-- 4. Ponowna próba - sukces
DROP table tmp_table
```

# Triggery - przykład

- Wykorzystamy utworzoną wcześniej tabelę *sumy*. Zdefiniujemy trigger DML AFTER INSERT, który wstawi do kolumny *suma* sumę liczb z kolumn *liczba\_1* oraz *liczba\_2*.

```
CREATE TRIGGER trg_sumy_a_ins ON sumy
AFTER INSERT
AS
BEGIN
    UPDATE sumy SET suma = liczba_1 + liczba_2 WHERE id IN (SELECT id FROM
Inserted)
END

-- 1. Test: wstawiamy liczby
INSERT INTO sumy(liczba_1, liczba_2) VALUES (4,5)

-- 2. Sprawdzamy: suma = 9.
SELECT * FROM sumy
```

- Ponieważ na tabeli *sumy* będziemy tworzyć kilka triggerów dobrze jest wyłączyć nieużywane w danym momencie triggery poleceniem:

```
DISABLE TRIGGER trg_sumy_a_ins ON sumy
```

# Triggery – przykład c.d.

- Zdarzenie i wywołany przez nią trigger występują w jednej transakcji. Co za tym idzie, proces który wykonuje np. polecenie INSERT na tabeli musi poczekać, aż trigger zakończy wykonywanie działań. Przy bardziej skomplikowanych operacjach może być to problemem. Zmodyfikujemy nieco trigger *trg\_sumy\_a\_ins*, dodając opóźnienie 10 sekund:

```
CREATE TRIGGER trg_sumy_a_ins ON sumy
AFTER INSERT
AS
BEGIN
    WAITFOR DELAY '000:00:10'
    UPDATE sumy SET suma = liczba_1 + liczba_2 WHERE id IN (SELECT id FROM
Inserted)
END

-- 1. Test: wstawiamy liczby - operacja jest dłuższa o 10 sekund
INSERT INTO sumy(liczba_1, liczba_2) VALUES (4,5)

-- 2. Dopiero po upływie długiego czasu sprawdzamy: sumy = 9.
SELECT * FROM sumy
```



# Triggery – przykład c.d.

- Dobrym rozwiązaniem byłby mechanizm, który operacje zawarte w triggerze wywołuje asynchronicznie. Proces, który wykonuje polecenie INSERT, natychmiast kontynuuje swoje dalsze wykonanie. Trigger zostaje wywołany w osobnym wątku, w późniejszym czasie, a w przypadku wielu poleceń INSERT i skomplikowanych operacji istnieje możliwość wykonania triggerów np. w nocy, gdy obciążenie serwera jest mniejsze.
- Takie możliwości daje nam **Service Broker**.

# Service Broker

- Jest to mechanizm asynchronicznej komunikacji pomiędzy procesami na jednym lub nawet na osobnych instancjach systemu SQL Server 2005.
- Opiera się na wysyłaniu komunikatów.
- Do zdefiniowania najprostszego mechanizmu potrzebujemy utworzyć:
  - Typ wiadomości.
  - Utworzyć kontrakt, który mówi jakie typu wiadomości są przesyłane i w którym kierunku.
  - Kolejkę – do której trafiają wysłane wiadomości.
  - Procedurę, nasłuchuje na kolejce i pobiera z niej wiadomości.
  - Serwis – który łączy wszystko.
- Schemat naszego triggera wyglądałby następująco: trigger wysyła komunikat do service brokera, że nastąpiło polecenie INSERT. Service broker pobiera wiadomość z kolejki i przetwarza tabelę, wstawiając sumę liczb do odpowiedniej kolumny.
- Przykład znajduje się w pliku *11\_prez\_serv\_broker.sql*

# Service Broker - przykład

- Przykład znajduje się w pliku *11\_prez\_serv\_broker.sql*.
- Test:

```
-- TEST

-- 1. Wstawiamy liczby
INSERT INTO Table_1 (Col1, Col2) VALUES (3, 6)

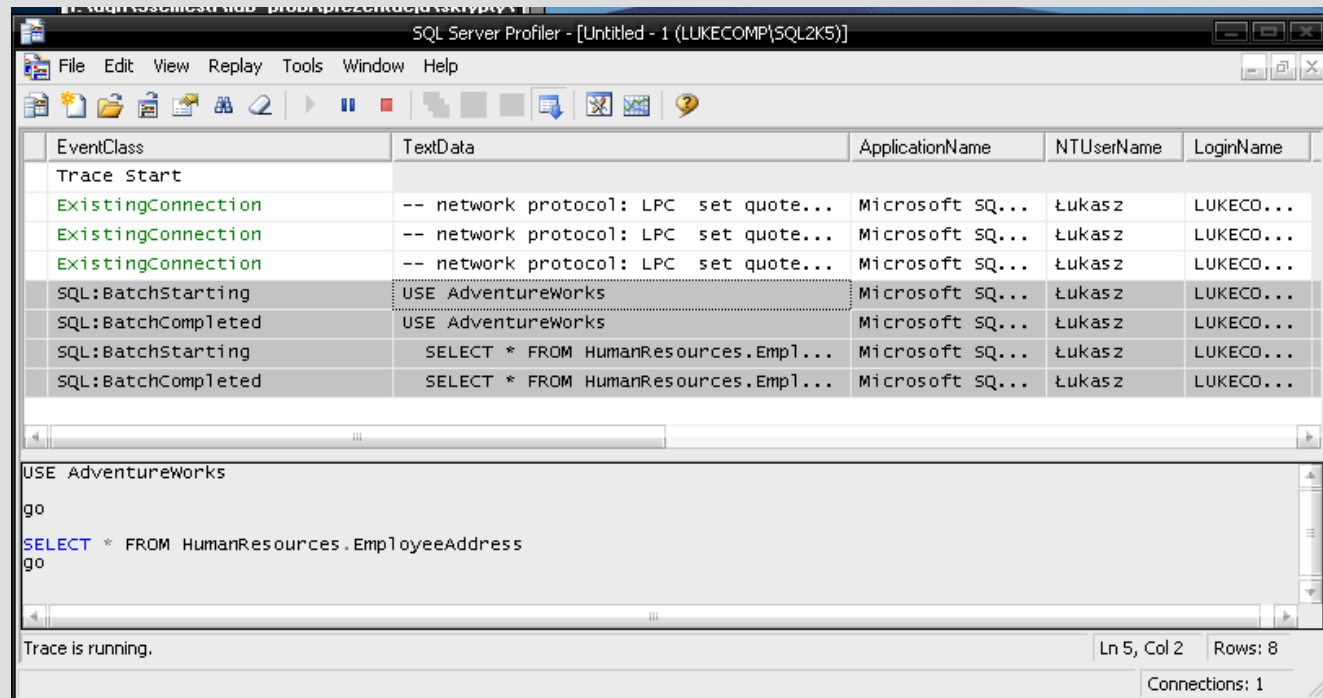
-- 2. Natychmiastowo sprawdzamy: sum = NULL
SELECT * FROM Table_1

-- 3. Oczekujemy 10 sekund i sprawdzamy: sum = 9
SELECT * FROM Table_1
```

- Warto zwrócić uwagę, że mimo iż opóźnienie jest ustawione na 10 sekund polecenie INSERT kończy się natychmiastowo.

# SQL Server Profiler

- Narzędzie, które pozwala podglądać zapytania wykonywane na bazie danych.
- Bardzo przydatne przy debugowaniu aplikacji oraz przy optymalizacji zapytań.
- Dostęp z menu: Tools -> SQL Server Profiler.
- Komendy języka SQL można skopiować z okna Profilera i analizować w SSMS.



The screenshot displays the SQL Server Profiler interface. The main window shows a table of events with columns: EventClass, TextData, ApplicationName, NTUserName, and LoginName. The events include 'Trace start', three 'ExistingConnection' events, and two 'SQL:BatchStarting' and 'SQL:BatchCompleted' events for a query.

EventClass	TextData	ApplicationName	NTUserName	LoginName
Trace start				
ExistingConnection	-- network protocol: LPC set quote...	Microsoft SQ...	Łukasz	LUKECO...
ExistingConnection	-- network protocol: LPC set quote...	Microsoft SQ...	Łukasz	LUKECO...
ExistingConnection	-- network protocol: LPC set quote...	Microsoft SQ...	Łukasz	LUKECO...
SQL:BatchStarting	USE AdventureWorks	Microsoft SQ...	Łukasz	LUKECO...
SQL:BatchCompleted	USE AdventureWorks	Microsoft SQ...	Łukasz	LUKECO...
SQL:BatchStarting	SELECT * FROM HumanResources.Emp1...	Microsoft SQ...	Łukasz	LUKECO...
SQL:BatchCompleted	SELECT * FROM HumanResources.Emp1...	Microsoft SQ...	Łukasz	LUKECO...

Below the table, the SQL text being executed is visible in a code editor:

```
USE AdventureWorks
go
SELECT * FROM HumanResources.EmployeeAddress
go
```

At the bottom of the window, it indicates 'Trace is running.' and shows 'Ln 5, Col 2' and 'Rows: 8'. The 'Connections: 1' indicator is also visible.

# Literatura

- *Microsoft SQL Server 2005. Implementation and Maintenance. Study Guide.* Joseph L. Jordan, Dandy Wayn, Wiley Publishing, Inc., 2006, Indianapolis, Indiana
- *Software Developers Journal* – nr 10/2008
- *Software Developers Journal Extra* – nr 19

# WEB

- <http://www.youtube.com/user/ittechtalk>
- <http://www.simple-talk.com/sql>