Laboratorium 9 – MS SQL Server

Temat: Środowisko CLR w MS SQL Server - UDT i UDA
Opracowanie: A.Dydejczyk

W ramach laboratorium przedstawiona zostanie technologia tworzenia CLR UDT (User Defined Type) – prezentacja typu CLR UDT hierarchyid (dostępnego w ramach SQL Server'a) i omówienie realizacji własnego typu CLR UDT oraz omówienie zagadnienie tworzenia agregatów CLR UDA.

Zagadnienia do opracowania w trakcie laboratorium:

1. Ćwiczenie A. CLR UDT – typ hierarchyid w SQL Server
2. Ćwiczenie B. Realizacja CLR UDT w SQL Server
3. Ćwiczenie C. Realizacja CLR UDA w SQL Server
4. Zadania

Ćwiczenie A. CLR UDT – typ hierarchyid w MS SQL Server

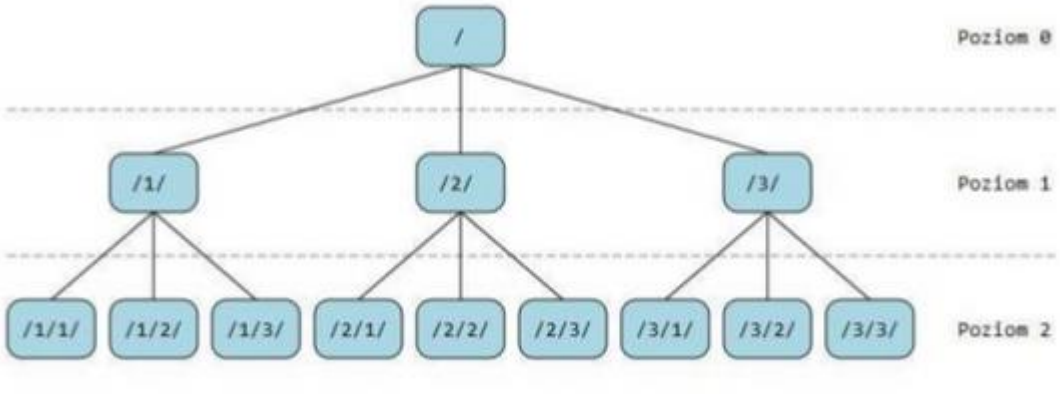
CLR UDT jest zaimplementowany jako klasa lub struktura na platformie .NET Framework. W ramach SQL Server Microsoft wprowadził trzy nowe typy danych CLR UDT: hierarchyid, geography i geometry. Typ hierarchyid zostanie przedstawiony w dalszej części ćwiczenia. Typ danych geometry służy do reprezentowania danych opisujących położenie na płaszczyźnie euklidesowej zgodnie ze standardem OGC (ang. Open Geospatial Consortium). Typ danych geography obsługuje dane przestrzenne uwzględniające krzywiznę Ziemi (elipsoidalne).

Poniżej linki do wspomnianych wyżej zdefiniowanych typów danych CLR UDT w MS SQL Server:

- https://docs.microsoft.com/en-us/sql/t-sql/data-types/hierarchyid-data-type-method-reference?view=sql-server-ver15
- https://docs.microsoft.com/en-us/sql/t-sql/spatial-geography/spatial-types-geography?view=sql-server-ver15
- https://docs.microsoft.com/en-us/sql/t-sql/spatial-geometry/spatial-types-geometry-transact-sql?view=sql-server-ver15

W ramach tego ćwiczenia zostanie zaprezentowany typ HierarchyID. HierarchyID jest specjalnym typem stworzonym realizacji zadań wymagających przetwarzania danych reprezentowanych w postaci grafu odpowiadającego strukturze hierarchicznej. Reprezentowanie danych hierarchicznych za pomocą tego modelu jest realizowane z wykorzystaniem binarnej zmaterializowanej ścieżki. Opracowana struktura charakteryzuje się wysokim stopniem kompresji. Operację na stworzonej tabeli będą zatem dużo efektywniejsze. Kolumna typu HierarchyID nie reprezentuje automatycznie drzewa. Do aplikacji wykorzystującej ten typ danych należy wygenerowanie i przypisanie wartości HierarchyID w taki sposób, aby pożądana relacja między

wierszami była odzwierciedlona w wartościach. Na rysunku 1 przedstawiono realizowaną przez typ HierarchyID organizację danych zgodną z tym typem.



Rys.1 Typ danych HierarchyID

Innym model często stosowanym w relacyjnym bazach danych jest model reprezentowania danych hierarchicznych z wykorzystaniem list sąsiedztwa (ang. adjacency list model). W tym modelu każdy wiersz tabeli zawiera odwołanie do wiersza nadrzędnego. Wzorzec listy sąsiedztwa nazywany również wzorcem samosprzężenia. Jest to tradycyjny wzorzec używany do modelowania danych hierarchicznych. Wzorzec listy sąsiedztwa przechowuje zarówno klucz bieżącego węzła, jak i klucz jego bezpośredniego nadrzędnego w bieżącym wierszu węzła.

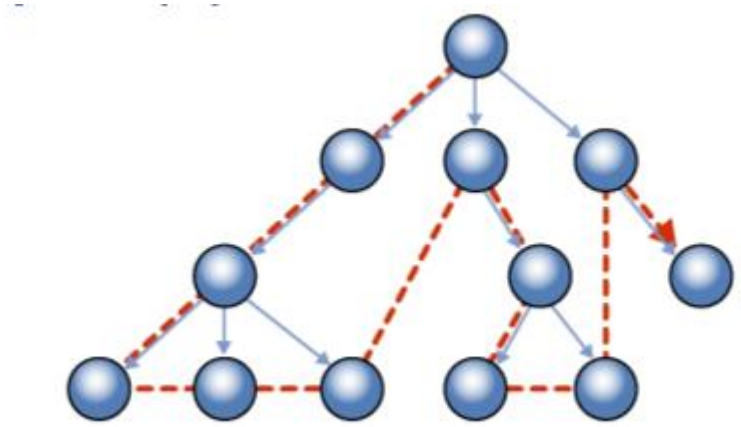
Model zmaterializowanej ścieżki wymaga zapisania hierarchicznej ścieżki z węzła głównego do bieżącego węzła. Ścieżka hierarchiczna jest podobna do współczesnych ścieżek w systemach plików, gdzie każdy folder, czyli katalog, reprezentuje węzeł w ścieżce. Do obsługi typu danych HierarchyID przygotowano odpowiednie metody przedstawione w tabeli 1.

Metoda	Opis
-- statyczne, dostępne z SQL	
hierarchyid::GetRoot()	Pobiera węzeł główny instancji hierarchyid
Hierarchyid::Parse(string)	Konwertuje podany ciąg znaków zapisany w postaci oddzielanej ukośnikami na ścieżkę hierarchyid
-- dynamiczne, dostępne z SQL	
@h.GetLevel()	Pobiera poziom instancji węzła hierarchyid
@h. GetAncestor(@n)	Pobiera n-tego przodka danej instancji węzła hierarchyid.
@h. GetDescendant(@n)	Pobiera n-tego potomka danej instancji węzła hierarchyid
@h. IsDescendantOf(@n)	Zwraca 1, jeśli wskazany węzeł jest potomkiem węzła danej instancji hierarchyid
@h.GetReparentedValue(@m,@n)	Zwraca węzeł przeniesiony ze stary_korzeń do nowy_korzeń.
@h.toString()	Konwertuje instancję hierarchyid na ciąg znaków zawierający nazwy węzłów oddzielane ukośnikami. Zwracana wartość jest typu nvarchar(4000).

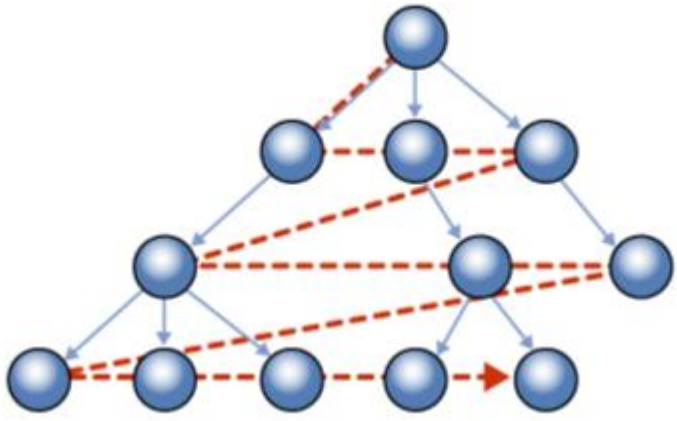
-- tylko .NET	
hierarchyid.Read()	
hierarchyid.Write()	

Tabela 1. Metody typu hierarchyid w MS SQL Server

Typ danych HierarchyID obsługuje generowanie i indeksowanie zmaterializowanych ścieżek w hierarchicznym modelowaniu danych. Dla realizacji optymalnego przeszukiwania struktury danych w typie HierarchyID udostępniono dwa typy indeksowania: „depth first" i „breadth first". Realizowaną strategię przeszukiwania w ramach tego indeksowania przedstawiają odpowiednio rysunek 2 i 3.



Rys.2 Strategia indeksowania „depth firts (DF)"



Rys.3 Strategia indeksowania „Breadth first (BF)"

Przykłady wykorzystania danych typy „hierarchyid" zawartymi w bazie danych AdeventureWork2008 w tabeli AdventureWorks2008.HumanResources.Employee zostały przedstawione w skryptach Lab09.01 i Lab09.02.

```
-- SQL Lab09.01
SELECT E.BusinessEntityID, P.FirstName + ' ' + P.LastName as 'Name',
  OrganizationNode, OrganizationNode.ToString() as 'HierarchyID.ToString()',
  OrganizationLevel
FROM AdventureWorks2008.HumanResources.Employee E
  JOIN AdventureWorks2008.Person.Person P
  ON E.BusinessEntityID = P.BusinessEntityID
GO
```

```
-- SQL Lab09.02
DECLARE @CurrentEmployee hierarchyid

SELECT @CurrentEmployee = OrganizationNode
FROM AdventureWorks2008.HumanResources.Employee
WHERE OrganizationNode = '/5/'

SELECT OrganizationNode.ToString() AS 'Hierarchy',
   p.FirstName + ' ' + p.LastName AS 'Name',
   e.OrganizationLevel,
   e.JobTitle
FROM AdventureWorks2008.HumanResources.Employee e
  INNER JOIN AdventureWorks2008.Person.Person p ON e.BusinessEntityID =
p.BusinessEntityID
  WHERE OrganizationNode.GetAncestor(1) = @CurrentEmployee
GO
```

Maksymalna wartość pola HierarchyID to 892 bajty. Jeżeli wartość zawarta w tym polu przekroczy tą wartość to dodanie nowego węzła się nie powiedzie. Poniżej prezdstawiono przykład budowania drzewa do momentu przekroczenia wartości możliwej do zapisania – skrypt Lab09.03.

```
-- SQL Lab09.03
if (object_id('dbo.hidtest')is not null)
        drop table dbo.hidtest
create table dbo.hidtest(n HierarchyID)
go

insert into dbo.hidtest(n) values (HierarchyID::GetRoot())

declare @hid HierarchyID
set @hid = ( HierarchyID::GetRoot()).GetDescendant(null,null)

declare @i int
set @i = 0
while @i > -1
begin
        begin try
                insert into hidtest (n) values (@hid)
                set @hid = @hid.GetDescendant(null,null)
                set @i = @i + 1
        end try
```

```
        begin catch
            declare @emes nvarchar(2048),@esev int, @esta int, @enum int
            set @emes = ERROR_MESSAGE()
            set @esev = ERROR_SEVERITY()
            set @esta = ERROR_STATE()
            set @enum = ERROR_NUMBER()
            raiserror(@emes, @esev, @esta, @enum)
            print 'Wartosc HID: ' + @hid.ToString()
            print 'Poziom HID ' + cast(@hid.GetLevel() as varchar(10))
            set @i = -1
        end catch
end
```

Na koniec przykład realizacji projektu z wykorzystaniem typu danych HierarchID do prezentacji danych związanych z strukturą administracyjną w Polsce. – skrypt Lab09.04.

```
-- SQL Lab09.04
if (object_id('dbo.SimpleDemo')is not null)
   drop table dbo.SimpleDemo
create table dbo.SimpleDemo
    (Node hierarchyid not null,
    [Geographical Name] nvarchar(30) not null,
    [Geographical Type] nvarchar(15) NULL);

insert dbo.SimpleDemo
values
-- second level data
 ('/1/1/','Boles³awieckie','Powiat')
,('/1/2/','G³ogowskie','Powiat')
,('/1/3/','Wa³brzyski','Powiat')
,('/2/1/','Lubelski','Powiat')
,('/2/2/','Rycki','Powiat')
,('/3/1/','Rawski','Powiat')
,('/4/1/','Nowotarski','Powiat')

-- first level data
,('/1/','Dolnoœl¹skie','Wojewodztwo')
,('/2/','Lubelskie','Wojewodztwo')
,('/3/','Łódzkie','Wojewodztwo')
,('/4/','Ma³opolskie','Wojewodztwo')
,('/5/','Opolskie','Wojewodztwo')
,('/6/','Pomorskie','Wojewodztwo')

-- third level data
,('/1/1/1/','Boles³awiec','Miasto')
,('/1/2/1/','G³ogów','Miasto')
,('/1/3/1/','Wa³brzych','Miasto')
,('/2/1/1/','Lublin','Miasto')
,('/2/2/1/','Rycki','Miasto')
,('/3/1/1/','Rawa Mazowiecka','Miasto')
,('/4/1/1/','Nowy Targ','Miasto')

-- root level data
```

```
,('/', 'Polska', 'Kraj')


-- display without sort order returns
-- rows in input order
select
 Node
,Node.ToString() AS [Node Text]
,Node.GetLevel() [Node Level]
,[Geographical Name]
,[Geographical Type]
        from dbo.SimpleDemo
```

Ćwiczenie B. CLR UDT w MS SQL Server – C#

W ramach ćwiczenia przedstawione zostanie technologia tworzenia CLR UDT (User Defined Type).
Poniżej linki do materiałów opisujących CLR UDT na stronie Microsoft.

- https://technet.microsoft.com/en-us/library/ms186366%28v=sql.105%29.aspx
- https://msdn.microsoft.com/pl-pl/library/ms131120%28v=sql.105%29.aspx
- https://docs.microsoft.com/en-us/sql/relational-databases/clr-integration/database-objects/clr-integration-custom-attributes-for-clr-routines?view=sql-server-2017

Identyfikacja CLR UDT realizowanego przez klasę (strukturę), która implementuje UDT opisuje atrybut SqlUserDefinedType.

```
SqlUserDefinedType  [( własność udt [, ...])]

własność udt :: =
   Format = { Native | UserDefined }
  | MaxByteSize = n
  | IsByteOrdered = { true | false }
  | ValidationMethod   =  string
  | IsFixedLength = { true | false }
  | Nazwa = string
```

Atrybuty CLR UDT i ich własności

Atrybut	Właściwość	Wartość	Opis
Serializable	brak	brak	Wskazuje, że UDT może być serializowany i deserializowany.
SqlUserDefinedType	Format.Native	brak	Określa, że UDT korzysta z natywnego formatu serializacji. Format natywny jest najbardziej wydajnym formatem przy serializacji i deserializacji, ale wprowadza pewne ograniczenia. Można użyć tylko

			typów danych .NET przechowujących wartości (Char, Integer i tak dalej) jako pola. Nie można wykorzystać referencyjnych typów danych (String, Array i tak dalej).
SqlUserDefinedType	Format.UserDefined	brak	Określa, że UDT korzysta z formatu serializacji zdefiniowanego przez użytkownika. Gdy jest użyty, UDT musi implementować interfejs IBinarySerialize i należy przygotować metody Write() oraz Read() serializujące i deserializujące dane UDT.
SqlUserDefinedType	IsByteOrdered	true/false	Pozwala porównywać i sortować wartości UDT w oparciu o ich reprezentację binarną. Jest również wymagane, jeśli tworzymy indeksy na kolumnach zdefiniowanych jako typ CLR UDT.
SqlUserDefinedType	IsFixedLength	true/false	Powinna być ustawiona na true, jeśli serializowana instancja UDT ma stałą długość.
SqlUserDefinedType	MaxByteSize	<=8000 lub –1	Maksymalny rozmiar serializowanych instancji w bajtach. Wartość może być z zakresu od 1 do 8000 lub przyjąć –1, co oznacza maksymalny rozmiar 2,1 GB.

Do realizacji zadania wykorzystamy Visual Studio, typ danych zostanie utworzony w języku C# oraz zostanie umieszczony w bazie danych „testCLR”.

Po utworzeniu nowego projektu w Visual Studio, wybieramy bazę danych „testCLR” i tworzymy szablon realizujący funkcjonalność – UDT. Poniżej odpowiedni kod, który należy umieścić w otwartym szablonie. Przedstawiony poniżej kod realizuje liczbę zespoloną i umożliwia dodanie dwóch liczb zespolonych poprzez wbudowaną metodę realizującą to działanie.

```
// Skrypt Lab09.05
[Serializable]
[Microsoft.SqlServer.Server.SqlUserDefinedType(Format.Native)]
public struct ComplexNumber : INullable
{
   private double _x; //real part
   private double _y; //imaginary part

   public bool IsNull
   {
      get { return m_Null; }
   }

   public static ComplexNumber Null
   {
      get
      {
         ComplexNumber h = new ComplexNumber();
         h.m_Null = true; return h;
      }
   }

   public ComplexNumber(double x, double y)
   {
      _x = x;
      _y = y;
```

```
        m_Null = false;
    }

    public ComplexNumber(bool nothing)
    {
        this._x = this._y = 0;
        this.m_Null = true;
    }

    public double RealPart
    {
        get { return _x; }
        set { _x = value; }
    }

    public double ImaginaryPart
    {
        get { return _y; }
        set { _y = value; }
    }

    public override string ToString()
    {
        return _x.ToString() + "+" + _y.ToString() + "i";
    }

    public static ComplexNumber Parse(SqlString s)
    {
        string value = s.Value;
        if (s.IsNull || value.Trim() == "")
            return Null;
        string xstr = value.Substring(0, value.IndexOf('+'));
        string ystr = value.Substring(value.IndexOf('+') + 1,
                    value.Length - xstr.Length - 2);
        double xx = double.Parse(xstr);
        double yy = double.Parse(ystr);
        return new ComplexNumber(xx, yy);
    }

    // Dodawanie liczb zespolonych
    public static ComplexNumber Add(ComplexNumber c1, ComplexNumber c2)
    {
        return new ComplexNumber(c1._x + c2._x, c1._y + c2._y);
    }

    // Private member
    private bool m_Null;
        }
```

Poprawność utworzonego typu danych sprawdzimy w SSMS wykonując poniższy polecenia T-SQL.

```
create table test ( complexField dbo.ComplexNumber);
insert into test (complexField) values('25+52i');
```

```
select complexField.ToString() from test;
insert into test values(ComplexNumber::[Add]('12+25i','25+12i'));
select complexField.ToString() from test;
drop table test;
```

W drugim przykładzie tworzymy typ opisujący punkt w układzie współrzędnych. Wartości współrzędnych należą do zbioru liczb całkowitych dodatnich. Dla przedstawionego typu należy opracować przykładowe dane testowe sprawdzające poprawność zaimplementowanych metod.

```
// Skrypt Lab09.06
using System;
using System.Data;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;
using System.Text;
[Serializable]
[SqlUserDefinedType(Format.Native,
IsByteOrdered = true, ValidationMethodName = "SprawdzPunkt")]
public struct Punkt : INullable
{
    private bool is_Null;
    private Int32 _x;
    private Int32 _y;
    public bool IsNull
    {
        get
        { return (is_Null); }
    }
    public static Punkt Null
    {
        get
        {
            Punkt pt = new Punkt();
            pt.is_Null = true;
            return pt;
        }
    }
    public override string ToString()
    {
        if (this.IsNull)
            return "NULL";
        else
        {
            StringBuilder builder = new StringBuilder();
            builder.Append(_x);
            builder.Append(",");
            builder.Append(_y);
            return builder.ToString();
        }
    }
    [SqlMethod(OnNullCall = false)]
    public static Punkt Parse(SqlString s)
    {
```

```
        if (s.IsNull)
            return Null;
        Punkt pt = new Punkt();
        string[] xy = s.Value.Split(",".ToCharArray());
        pt.X = Int32.Parse(xy[0]);
        pt.Y = Int32.Parse(xy[1]);
        if (!pt.SprawdzPunkt())
            throw new ArgumentException("Invalid XY coordinate values.");
        return pt;
    }
    public Int32 X
    {
        get
        { return this._x; }
        set
        {
            Int32 temp = _x;
            _x = value;
            if (!SprawdzPunkt())
            {
                _x = temp; throw new ArgumentException("Zła współrzędna X.");
            }
        }
    }
    public Int32 Y
    {
        get
        { return this._y; }
        set
        {
            Int32 temp = _y;
            _y = value;
            if (!SprawdzPunkt())
            {
                _y = temp;
                throw new ArgumentException("Zła współrzędna X.");
            }
        }
    }
    private bool SprawdzPunkt()
    {
        if ((_x >= 0) && (_y >= 0))
        {
            return true;
        }
        else
        {
            return false;
        }
    }
    [SqlMethod(OnNullCall = false)]
    public Double OdlegloscOdXY(Int32 iX, Int32 iY)
    {
        return Math.Sqrt(Math.Pow(iX - _x, 2.0) + Math.Pow(iY - _y, 2.0));
```

```
    }
    [SqlMethod(OnNullCall = false)]
    public Double Odleglosc()
    {
        return OdlegloscOdXY(0, 0);
    }
    [SqlMethod(OnNullCall = false)]
    public Double OdlegloscOd(Punkt pFrom)
    {
        return OdlegloscOdXY(pFrom.X, pFrom.Y);
    }
}
```

Ćwiczenie C. Realizacja CLR UDA w MS SQL Server – C#

W kolejnym ćwiczeniu przedstawione zostaną funkcje agregujące. Poniżej link do dokumentacji na stronie firmy Microsoft:

- https://docs.microsoft.com/en-us/sql/relational-databases/clr-integration-database-objects-user-defined-functions/clr-user-defined-aggregates?view=sql-server-2017

Funkcje agregujące definiowane przez użytkownika mają funkcjonalność podobną do wbudowanych funkcji agregujących (tj. SUM czy AVG), działają od razu na całym zbiorze danych, w odróżnieniu od przetwarzania element po elemencie. Funkcje agregujące CLR UDA mają dostęp do funkcjonalności .NET i mogą operować na typach danych numerycznych, znakowych, daty i czasu lub UDT. Wyróżniamy dwa rodzaje CLR UDA – proste i zaawansowane. Proste CLR UDA (Format.Native) wymagają zdefiniowania czterech metod.

- **public void Init( )** - UDA wywołuje swoją metodę, gdy silnik SQL Server przygotowuje agregat. Kod tej metody może resetować poszczególne zmienne do stanu początkowego, inicjalizować bufory i wykonywać inne czynności inicjalizacyjne.
- **public void Accumulate(input_type value)** - wywoływana jest przy przetwarzaniu każdego wiersza, pozwalając na dołączanie przekazanych danych. Metoda Accumulate() może zwiększać licznik, dodawać wartość wiersza do sumy lub wykonywać inne bardziej złożone obliczenia na danych z wiersza.
- **public void Merge(udagg_class value)** - wywoływana jest, gdy SQL Server zdecyduje, by wykorzystać przetwarzanie równoległe do zakończenia tworzenia agregatu. Jeśli silnik zapytania MS Server zdecyduje, by zastosować przetwarzanie równoległe, tworzy wiele instancji UDA i wywołuje metodę Merge(), by połączyć wyniki w pojedynczy agregat.
- **public return_type Terminate( )** - końcowa metoda UDA. Jest wywoływana po przetworzeniu wszystkich wierszy i połączeniu wszystkich agregatów utworzonych przy przetwarzaniu równoległym. Metoda Terminate() zwraca końcowy wynik z agregatu do silnika zapytania.

Tworząc funkcję agregującą mamy dodatkowe własności, które definiujemy w trakcie tworzenia agregatu.

SqlUserDefinedAggregate [(*aggregate-attribute* [,...])]

***aggregate-attribute*::=**
Format = { Native | UserDefined }
IsInvariantToDuplicates = { true | false }
IsInvariantToNulls = { true | false }
IsInvariantToOrder = { true | false }
IsNullIfEmpty = { true | false }
| MaxByteSize = *n*

Format = { Native | UserDefined }

Określa format serializacji dla danych przetwarzanych przez agregat. Dla typów prostych używamy serializacji natywnej „native" natomiast dla typów złożonych, referencyjnych używamy serializacji zdefiniowanej przez użytkownika „UserDefined". Serializacja natywna jest zalecana dla prostych typów zawierających tylko pola następujących typów: bool, bajt, sbyte, short, ushort, int, uint, long, ulong, float, double, SqlByte, SqlInt16, SqlInt32, SqlInt64, SqlDateTime, SqlSingle, SqlDateTime, SqlSingle, SqlDouble , SqlMoney i SqlBoolean. Macierzysta serializacja może również zawierać UDT, które używają rodzimej serializacji. Serializacja natywna nie może być ustawiona dla typów posiadających własność MaxByteSize oraz typów, które nie mają zdefiniowanej serializacji.

Serializacja zdefiniowana przez użytkownika steruje serializacją za pomocą kodu i wymaga określenia właściwości MaxByteSize atrybutu SqlUserDefinedAggregate oraz dodatkowo klasa lub struktura implementująca typ musi implementować metody Read() i Write() interfejsu IBinarySerializable, aby odczytywać i zapisywać strumień bajtów.

IsInvariantToDuplicates

Określa, czy agregat jest niezmienny dla duplikatów. Na przykład MAX i MIN są niezmienne dla duplikatów, a AVG i SUM nie.

IsInvariantToNulls

Określa, czy agregat jest niezmienny do wartości pustych. Na przykład MAX i MIN są niezmienne do wartości ***null***, a COUNT nie (ponieważ wartości ***null*** są uwzględniane w liczniku).

IsInvariantToOrder

Określa, czy agregat jest niezmienny do kolejności wartości. Określenie ***true*** daje optymalizatorowi zapytań większą elastyczność w wyborze planu wykonania i może skutkować poprawą wydajności.

IsNullIfEmpty

Określa, czy agregat zwraca wartość zerową, jeśli nie zgromadzono żadnych wartości.

W przeciwnym razie zwracana jest wartość, którą zainicjowana wartość zmiennej zwróconej przez metodę Terminate().

MaxByteSize

Maksymalny rozmiar instancji UDT. Wartość MaxByteSize musi być określona, jeśli właściwość Format jest ustawiona na UserDefined.

Na początek funkcja agregująca zliczająca liczbę wartości znajdujących się w określonym przedziale. Do realizacji zadania wykorzystamy język C#, Visual Studio a nasz agregat umieścimy w bazie danych „testCLR”.

```
// Skrypt Lab09.07
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;

[Serializable]
[Microsoft.SqlServer.Server.SqlUserDefinedAggregate(Format.Native)]
public struct uda_CountOfRange
{
    private SqlInt32 iNumRange;
    public void Init()
    {
        this.iNumRange = 0;
    }
    public void Accumulate(SqlInt32 Value)
    {
        if ((Value) < 2 && (Value) > -2)
        {
            this.iNumRange += 1;
        }
    }
    public void Merge(uda_CountOfRange Group)
    {
        this.iNumRange += Group.iNumRange;
    }
    public SqlInt32 Terminate()
    {
        return ((SqlInt32)this.iNumRange);
    }
};
```

W ramach ćwiczenia proszę opracować odpowiedni kod SQL w bazie „testCLR” sprawdzający poprawność opracowanej funkcji agregującej. Na porzeby ćwiczenia proszę utworzyć tabelę „test1” z atrybutami: a1 typu integer i a2 typu char(10). Tabelę należy wypełnić przykładowymi danymi.

Kolejny agregat zlicza liczbę wartości negatywnych. Poprawność agregatu sprawdzić przygotowując odpowiednie dane w bazie danych „testCLR”.

```
// Skrypt Lab09.08
using System;
using System.Collections.Generic;
using System.Data;
using System.Data.SqlTypes;
using System.Runtime.InteropServices;
using Microsoft.SqlServer.Server;
[Serializable]
[Microsoft.SqlServer.Server.SqlUserDefinedAggregate(Format.Native)]
public struct uda_CountOfNegatives
{
   //class fields
   private SqlInt32 iNumOfNegatives;
   public void Init()
   {
      this.iNumOfNegatives = 0;
   }
   public void Accumulate(SqlInt32 Value)
   {
      if ((Value) < 0)
      {
         this.iNumOfNegatives += 1;
      }
   }
   public void Merge(uda_CountOfNegatives Group)
   {
      this.iNumOfNegatives += Group.iNumOfNegatives;
   }
   public SqlInt32 Terminate()
   {
      return ((SqlInt32)this.iNumOfNegatives);
   }
};
```

Ostatni agregat obliczy medianę w zbiorze danych. Poprawność agregatu należy sprawdzić przygotowując odpowiedni zestaw danych. Dodatkowo opracować funkcję znajdującą medianę korzystając z języka T-SQL. Agregat zostanie zrealizowany w wersji rozszerzonej CLR UDA.

```
// Skrypt Lab09.09
using System;
using System.Collections.Generic;
using System.Data;
using System.Data.SqlTypes;
using System.Runtime.InteropServices;
using Microsoft.SqlServer.Server;
namespace Apress.Examples
{
   [Serializable]
   [Microsoft.SqlServer.Server.SqlUserDefinedAggregate(
   Format.UserDefined,
   IsNullIfEmpty = true,
   MaxByteSize = 8000)]
   [StructLayout(LayoutKind.Sequential)]
   public struct Median : IBinarySerialize
```

```
{
    List<double> temp;
    public void Init()
    {
        this.temp = new List<double>();
    }
    public void Accumulate(SqlDouble number)
    {
        if (!number.IsNull)
        {
            this.temp.Add(number.Value);
        }
    }
    public void Merge(Median group)
    {
        this.temp.InsertRange(this.temp.Count, group.temp);
    }
    public SqlDouble Terminate()
    {
        SqlDouble result = SqlDouble.Null;
        this.temp.Sort();
        int first, second;
        if (this.temp.Count % 2 == 1)
        {
            first = this.temp.Count / 2;
            second = first;
        }
        else
        {
            first = this.temp.Count / 2;
            second = first + 1;
        }
        if (this.temp.Count > 0)
        {
            result = (SqlDouble)(this.temp[first] + this.temp[second]) / 2.0;
        }
        return result;
    }
    #region IBinarySerialize Members
    // Własna metoda do wczytywania serializowanych danych.
    public void Read(System.IO.BinaryReader r)
    {
        this.temp = new List<double>();
        int j = r.ReadInt32();
        for (int i = 0; i < j; i++)
        {
            this.temp.Add(r.ReadDouble());
        }
    }
    // Własna metoda do zapisywania serializowanych danych.
    public void Write(System.IO.BinaryWriter w)
    {
        w.Write(this.temp.Count);
        foreach (double d in this.temp)
```

```
            {
                w.Write(d);
            }
        }
        #endregion
    }
}
```

Zadanie Z1

Opracować dodatkowe metody do przedstawionego w przykładzie **Lab09.05** typu użytkownika (UDT) reprezentującego liczbę zespoloną realizujące następujące zadania: liczba zespolona sprzężona i moduł z liczby zespolonej.

Zadanie Z2

Opracować agregat zliczający liczby naturalne podzielne przez 3.

Zadanie Z3

Opracować własne agregaty CLR UDA realizujące funkcjonalność agregatów dostępnych w SQL Server Avg(), Stdev() oraz Var() i porównać ich wydajność z agregatami dostępnymi w SQL Server dla reprezentatywnego zbioru danych.