SQL Server Array Library

The Johns Hopkins University, Department of Physics and Astronomy
Eötvös University, Department of Physics of Complex Systems
http://voservices.net/sqlarray, dobos@complex.elte.hu

Introduction

TBW

Supported systems

The SQL Array Library supports Microsoft SQL Server 2008 (R1 and R2) x64. The current code cannot be compiled to 32 bit.

Installation

Installation using the precompiled scripts

The package is distributed as two install scripts (install.sql, uninstall.sql) for SQL Server deployment, and as a bunch of DLL files for .net application. The SQL scripts already contain the DLL files. When installing the scripts, never attempt to open them using any text editor as the extremely long lines in the scripts will very likely crash the text editor. Use the sqlcmd utility to execute them. First create an empty database called SqlArray on the server instance where you intend to install the library. Enable CLR integration with the following SQL command using SQL Server Management Studio.

```
EXEC sp_configure 'show advanced options' , '1';
go
reconfigure;
go
EXEC sp_configure 'clr enabled' , '1'
go
reconfigure;
-- Turn advanced options back off
EXEC sp_configure 'show advanced options' , '0';
go
```

Because SQL Array uses unsafe binaries, it is either necessary to mark the SqlArray database trustworthy, or to use a keyfile. To mark the database trustworthy, the following SQL command has to executed.

```
ALTER DATABASE SqlArray
SET TRUSTWORTHY ON
GO
```

Alternatively, a key file can be used to grant the assemblies unsafe access. The key file is located in the .\Key folder and can be installed with the InstallKey.sql script. Edit this SQL script in Management Studio to point to the right key file location and execute.

Now the SQL Array library can be installed. Open a command line window, enter into the directory where the install scripts are located and execute the following command (parameters are case sensitive!):

sqlcmd -S *servernameinstace* -E -d SqlArray -i Install.sql**

-E is used for windows authentication. If you want to use SQL Server authentication specify the **-U *username* -P *password*** parameters instead.

Compiling the source code

Compiling the source code requires the following software installed:

- Windows 7/2008 Server SDK
- Visual Studio 2008
- Visual Studio 2010

Both VS 2008 and VS 2010 are required! Because of a bug in .net 2.0, the install script generator only works in .net 4.0 but Visual C 2008 is required to compile C++/CLI code for .net 2.0, VC 2010 only compiles code for .net 4.0 which cannot be hosted in SQL Server 2008. Hopefully, this issue will be able to be fixed once the bug in .net is fixed.

Building the project is done by building the GenerateInstallScript project under build utilities. This should recompile everything and execute the script generator automatically. The generated scripts will be placed under .\scripts.

Array basics

Storage classes of arrays

The SQL Array Library uses the SQL Server binary data-type to store arrays.

SQL Server supports to ways of storing binary data. If the structure size is under 8000 bytes it is stored in-page and the corresponding SQL data-type is varbinary(n) where n ≤ 8000. If data is bigger than 8000 bytes it is stored in a B-tree “out-of-page”, the corresponding SQL data-type is varbinary(max).

When you want to create a table that stores arrays just create columns of type varbinary(n) or varbinary(max), there are no user-defined types for arrays.

Accessing data stored in B-trees has some overhead compared to inline data. Reading and writing B-trees is inherently slower than reading and writing single database pages but also accessing the BLOBs from .net requires routing bytes through a BinaryReader/BinaryWriter which has a significant additional cost.

In order to reduce the overhead for very small arrays but allow huge arrays at the same time the SQL Array Library distinguishes two storage classes: ShortArray and MaxArray. ShortArray uses varbinary(n) as storage and accesses data via memory buffers (fast) while MaxArray uses varbinary(max) and accesses data stored as a B-tree using BinaryReader and BinaryWriter which is about 10 times slower.

ShortArray

Short arrays must be less than 8000 bytes including the raw data and a 24 byte header. They are indexed using 16 bit integers (smallint, Int16, short). The maximum number of indices is limited to 8.

Short arrays are always completely read into the memory.

Short arrays are usually used for short vectors that index array elements.

MaxArray

Max arrays must be less than 2 GB and have a varying header size because they support an unlimited number of indices.

Max arrays can be loaded partially – if only a subset of the data is required for a certain operation. This can significantly speed up data access in specific scenarios.

Types of arrays

The following element types are supported for both ShortArrays and MaxArrays: (types are listed as SQL, .net, c#)

- tinyint, Byte, byte
- smallint, Int16, short
- int, Int32, int
- bigint, Int64, long
- real, Single, float
- float, Double, double
- SqlRealComplex
- SqlFloatComplex

User-defined types for SqlRealComplex and SqlFloatComplex are implemented.

As SQL Server does not support overloading of user-defined functions, functions handling the different data-types/storage classes are organized under different schema names. The schema names are constructed as follows:

*Typename*Array*Storage*,

where *Typename* is the element type, Array is a constant part of the name, and *Storage* is either nothing or Max for the varbinary(max) storage type. For example:

- SmallIntArray: a short array of Int16 values, typically used for indexing short arrays
- IntArray: a short array of Int16 values, typically used for indexing max arrays.
- RealArray: a short array of single precision floating point numbers, may be used for storing data points
- FloatArrayMax: an array for storing a large number of double precision floating point numbers

- FloatComplexArrayMax: an array for storing a large number of double precision complex numbers

Arrays of different types can be handled only by the appropriate functions, see later.

Declaring variables for arrays

Variables are stored in variables of data type varbinary(n) or varbinary(max). In case of fixed size binary variables, the size of the structure have to be estimated from the size of the element type + 24 bytes for the header. Because varbinary(max) is automatically sized, no estimate on the actual data size is necessary.

A variable to store an array can be declared the following way:

```
DECLARE @a varbinary(100)
```

It can also be combined with initializing the value:

```
DECLARE @a varbinary(100) = SqlArray.RealArray.Vector_3(1.0, 2.0, 3.0)
```

Note, that the function Vector_3 will create a one dimensional vector with three elements, not a 1×2×3 array. When using the SQL Array library, there is no necessity to allocate empty arrays prior to use them, so no function to create empty arrays of a given size exists.

Calling SqlArray functions

All array operations are mapped to ordinary CLR functions. Calling them requires supplying the database name that contains the CLR assemblies, the schema name that identifies the data type/storage class and the function name. For example, retrieving an item from a vector is done by the following function call:

```
DECLARE @a varbinary(100) = SqlArray.RealArray.Vector_3(1.0, 2.0, 3.0)
SELECT SqlArray.RealArray.Item_1(@a, 2)
```

Here SqlArray is the name of the database containing the assemblies and functions, RealArray is the schema name corresponding to the array type and Item_1 is the function that indexes one dimensional arrays.

Because overloading of functions is not supported in SQL CLR, several functions have different versions for different number of indices, usually denoted with *_r* where *r* is the number of indices. These functions usually have an alternative version called _N where indices are passed as an array and not as a bunch of SQL scalars, as the following example demonstrates.

```
DECLARE @a varbinary(100) = SqlArray.RealArray.Vector_3(1.0, 2.0, 3.0)
SELECT SqlArray.RealArray.Item_N(@a, SqlArray.SmallIntArray.Vector_1(2))
```

Here the function Item_N accepts a vector indexing the array @a as a second parameter. Short arrays are indexed using SmallIntArrays while max arrays are indexed by IntArrays.

Library Reference

Parse function

Arrays can be directly created from string representation. The syntax of the string representation uses brackets to denote arrays and sub-arrays while commas to separate array elements. A two index array (matrix) is written the following way: [[1, 2],[3, 4]]. Note, that sub-arrays on the same level must always have the same number of elements (no jagged arrays allowed).

Syntax:

SqlArray.*arraytype*.Parse(@s nvarchar(max))

Parameters:

@s is the string representation of the array

Return value:

The binary representation of the array as a varbinary(8000) or varbinary(max).

Example:

```
DECLARE @a varbinary(100) = SqlArray.FloatArray.Parse('[[1, 2],[3, 4]]')
```

ToString function

Convers an array into string.

Syntax:

SqlArray.*arraytype*.ToString(@v varbinary(n))

Parameters:

@v is an array in binary representation

Return value:

The string representation of the array as a varchar(max).

Example:

```
DECLARE @a varbinary(100) = SqlArray.FloatArray.Parse('[[1, 2],[3, 4]]')
SELECT SqlArray.FloatArray.ToString(@a)
```

Vector_n functions

The Vector_n function creates a one index array with n elements. The value of n can be any number between 1 and 10.

Syntax:

SqlArray.*arraytype*.Vector_n(@v1, @v2, ... , @vn)

Parameters:

@v1, @v2, … , @vn are parameters of the element type of the array.

Returns:

The binary representation of the array as varbinary(8000) or varbinary(max).

Example:

```
DECLARE @a varbinary(100) = SqlArray.FloatArray.Vector_5(1, 2, 3, 4, 5)
```

Matrix_n functions

Works similar to Vector_n except creates 2×2 or 3×3 matrices. The value of n can only be 2 or 3.

Syntax:

SqlArray.*arraytype*.Matrix_n(@v1, @v2, ... , @vn)

Parameters:

@v1, @v2, … , @vn are parameters of the element type of the array.

Returns:

The binary representation of the array as varbinary(8000) or varbinary(max).

Example:

```
DECLARE @a varbinary(100) = SqlArray.FloatArray.Matrix_2(1, 2, 3, 4)
```

Rank function

Returns the “rank” of the array which is defined as the number of indices (sometimes mentioned as dimensions).

Syntax:

SqlArray.arraytype.Rank(@a)

Parameters:

@a is a parameter of type varbinary(8000) or varbinary(max) containing the binary representation of an array.

Returns:

The number of ranks as a smallint or int, depending on the array storage class.

Example:

```
DECLARE @a varbinary(100) = SqlArray.FloatArray.Matrix_2(1, 2, 3, 4)
SELECT SqlArray.FloatArray.Rank(@a) -- returns 2
```

TotalLength function

Returns the total number of elements in the array. The total length is calculated as the product of index intervals.

Syntax:

SqlArray.arraytype.TotalLength(@a)

Parameters:

@a is a parameter of type varbinary(8000) or varbinary(max) containing the binary representation of an array.

Returns:

The total number of elements as a smallint or int, depending on the array storage class.

Example:

```
DECLARE @a varbinary(100) = SqlArray.FloatArray.Matrix_2(1, 2, 3, 4)
SELECT SqlArray.FloatArray.TotalLength(@a) -- returns 4
```

Length function

Returns the length of a single rank of an array (size of a specific dimension).

Syntax:

SqlArray.arraytype.Length(@a, @k)

Parameters:

@a is a parameter of type varbinary(8000) or varbinary(max) containing the binary representation of an array.

@k is a parameter of type tinyint that indexes the ranks.

Returns:

The the length of a single rank of an array as a smallint or int, depending on the array storage class.

Example:

```
DECLARE @a varbinary(100) = SqlArray.FloatArray.Parse('[[1, 2],[3, 4],[5, 6]]')
SELECT SqlArray.FloatArray.Length(@a, 0)     -- returns 2
SELECT SqlArray.FloatArray.Length(@a, 1)     -- returns 3
```

Lengths function

Returns all the length of the array in the form of an array.

Syntax:

SqlArray.arraytype.Lengths(@a)

Parameters:

@a is a parameter of type varbinary(8000) or varbinary(max) containing the binary representation of an array.

Returns:

An array of smallint or int containing the length of the ranks of the array @a.

Example:

```
DECLARE @a varbinary(100) = SqlArray.FloatArray.Parse('[[1, 2],[3, 4],[5, 6]]')
SELECT SqlArray.SmallIntArray.ToString(SqlArray.FloatArray.Lengths(@a))
-- returns [2,3]
```

Cast function

Can be used to convert an existing BLOB into an array by prefixing it with a header. An offset can be specified if the original BLOB already has a header. BLOBs are raw binaries containing a the array elements in a column major format.

Syntax:

SqlArray.arraytype.Cast(@b, @offset, @size)

Parameters:

@b is a binary blob containing the raw element data in a column major format. The internal binary representation must match the element type of the array.

@offset is a small in or int (depending on the array storage class) that can be used to skip the header of the BLOB, if there is any.

@size is the size of the target array. The number of elements of the target array must match the number of elements in the raw BLOB.

Returns:

An array in binary representation with a valid SQL Array header.

Example:

```
DECLARE @size varbinary(100) = SqlArray.SmallIntArray.Vector_2(2,2)
DECLARE @a varbinary(100) = SqlArray.TinyIntArray.Cast(0x00010203, 0, @size)
SELECT SqlArray.TinyIntArray.ToString(@a) -- returns [[0,1],[2,3]]
```

Raw function

Returns the raw bytes of the array, without the proprietary header.

Syntax:

SqlArray.arraytype.Raw(@a)

Parameters:

@a is an array in binary representation with the proprietary header.

Returns:

An array in binary representation with a valid SQL Array header.

Example:

```
DECLARE @a varbinary(100) = SqlArray.SmallIntArray.Parse('[[1, 2],[3, 4],[5, 6]]')
SELECT SqlArray.SmallIntArray.Raw(@a) -- returns 0x010002000300040005000600
```

Reshape function

Changes the lengths of the array without changing the individual items. Reshaping is only possible if the original and the resulting total number of items is the same. No buffer reallocation will happen if the header sizes are the same. This latter always applies to the short array storage class. In case of max arrays, buffer reallocation can be omitted only when the number of indices remains the same.

Syntax:

SqlArray.arraytype.Raw(@a)

Parameters:

@a is an array in binary representation.

Returns:

An array in binary representation with a valid SQL Array header.

Example:

```
DECLARE @a varbinary(100) = SqlArray.FloatArray.Parse('[[1, 2],[3, 4],[5, 6]]')
SELECT SqlArray.SmallIntArray.ToString(SqlArray.FloatArray.Lengths(@a))
  -- returns [2,3]
DECLARE @size varbinary(100) = SqlArray.SmallIntArray.Vector_2(3,2)
SET @a = SqlArray.FloatArray.Reshape(@a, @size)
SELECT SqlArray.SmallIntArray.ToString(SqlArray.FloatArray.Lengths(@a))
  -- returns [3,2]
```

Transpose function

Performes a transpose operation on an array with exactly two indices (a matrix). A more general "Permute" function exists to more detailed manipulation of multi-index arrays.

Syntax:

SqlArray.arraytype.Transpose(@a)

Parameters:

@a is an array in binary representation.

Returns:

The transposed array in binary representation.

Example:

```
DECLARE @a varbinary(100) = SqlArray.FloatArray.Parse('[[1, 2],[3, 4],[5, 6]]')
SET @a = SqlArray.FloatArray.Transpose(@a)
SELECT SqlArray.FloatArray.ToString(@a)      -- returns [[1,3,5],[2,4,6]]
```

Permute function

Performes a generalized transform operation over an array.

Syntax:

SqlArray.arraytype.Permute(@a, @i)

Parameters:

@a is an array in binary representation.

@i is the new order of indices in the form of an array.

Returns:

The permuted array in binary representation.

Example:

```
DECLARE @i varbinary(100) = SqlArray.TinyIntArray.Vector_2(1, 0)
DECLARE @a varbinary(100) = SqlArray.FloatArray.Parse('[[1, 2],[3, 4],[5, 6]]')
SET @a = SqlArray.FloatArray.Permute(@a, @i)
SELECT SqlArray.FloatArray.ToString(@a)      -- returns [[1,3,5],[2,4,6]]
```

Empty function

This is a function that does not do anything. This is useful, however, for performance testing when the pure overhead of CLR calls needs investigation.

Syntax:

SqlArray.arraytype.Empty(@a, @i)

Parameters:

@a is an array in binary representation.

@i is a number.

Returns:

0

Item, Item_N and Item_r functions

Return an item of the array. **Item** returns the item by its linear index, **Item_N** accepts the index as an array while **Item_*r*** support a fixed number of *r* indices.

Syntax:

SqlArray.arraytype.Item(@a, @li)
SqlArray.arraytype.Item_N(@a, @ia)
SqlArray.arraytype.Item_r(@a, @i1, @i2, … @ir)

Parameters:

@a is the array in binary representation.

@li is a number, the linear index of the item in a column-major element ordering scheme.

@ia is an array containing the indices of the element.

@i1, … @ir are numbers indexing the element

Returns:

The indexed element.

Example:

```
DECLARE @ia varbinary(100) = SqlArray.SmallIntArray.Vector_2(1, 1)
DECLARE @a varbinary(100) = SqlArray.FloatArray.Parse('[[1, 2],[3, 4],[5, 6]]')
SELECT SqlArray.FloatArray.Item(@a, 4)              -- returns 5
SELECT SqlArray.FloatArray.Item_N(@a, @ia)          -- returns 4
SELECT SqlArray.FloatArray.Item_2(@a, 1, 2)         -- returns 6
```

Items function

Returns a list of items. The indices of the items are passed as an array and the values are returned also as an array.

Syntax:

SqlArray.arraytype.Items(@a, @ix)

Parameters:

@a is the input array

@ix is the array containing the indices.

Returns:

An array containing the indexed items, in binary representation

Example:

```
DECLARE @a varbinary(100) = SqlArray.FloatArray.Matrix_2(1, 2, 3, 4)
DECLARE @ix varbinary(100) = SqlArray.SmallIntArray.Parse('[[0,0],[1,1],[0,1]]')
DECLARE @b varbinary(100) = SqlArray.FloatArray.Items(@a, @ix)
SELECT SqlArray.FloatArray.ToString(@b)             -- returns [1,4,3]
```

UpdateItem, UpdateItem_N and UpdateItem_r functions

These functions are used for updating an item. The indexing scheme is the same as in case of the **Item** functions. They return the whole array.

Syntax:

SqlArray.arraytype.UpdateItem(@a, @i, @v)
SqlArray.arraytype.UpdateItem_N(@a, @ia, @v)
SqlArray.arraytype.UpdateItem_r(@a, @i1, @i2, … @ir, @v)

Parameters:

@a is the array in binary representation.

@i is a number, the linear index of the item in a column-major element ordering scheme.

@ia is an array containing the indices of the element.

@i1, … @ir are numbers indexing the element

@v is the new value of the item.

Returns:

The updated array in its binary representation.

Example:

```
DECLARE @ia varbinary(100) = SqlArray.SmallIntArray.Vector_2(1, 1)
DECLARE @a varbinary(100) = SqlArray.FloatArray.Parse('[[0, 0],[0, 0],[0, 0]]')
SET @a = SqlArray.FloatArray.UpdateItem(@a, 4, 1)
SELECT SqlArray.FloatArray.ToString(@a)             -- returns [[0,0],[0,0],[1,0]]
SET @a = SqlArray.FloatArray.UpdateItem_N(@a, @ia, 2)
SELECT SqlArray.FloatArray.ToString(@a)             -- returns [[0,0],[0,2],[1,0]]
SET @a = SqlArray.FloatArray.UpdateItem_2(@a, 0, 1, 3)
SELECT SqlArray.FloatArray.ToString(@a)             -- returns [[0,0],[3,2],[1,0]]
```

UpdateItems function

Updates items in an array. The items are indexes by indices listed in an array and the new values are also stored in an array.

Syntax:

SqlArray.*arraytype*.UpdateItems(@a, @ix, @x)

Parameters:

@a is the original array, in binary representation

@ix is the array containing the indices

@v is the array containing the new values

Returns:

The updated array in binary representation.

Example:

```
DECLARE @size varbinary(100) = SqlArray.SmallIntArray.Vector_2(2, 3)
```

```
DECLARE @a varbinary(100) = SqlArray.FloatArray.FromScalar(@size, 0)
SELECT SqlArray.FloatArray.ToString(@a)             -- returns [[0,0],[0,0],[0,0]]
DECLARE @ix varbinary(100) = SqlArray.SmallIntArray.Parse('[[0,0],[0,1],[1,0],[1,1]]')
DECLARE @v varbinary(100) = SqlArray.FloatArray.Parse('[1,2,3,4]')
DECLARE @b varbinary(100) = SqlArray.FloatArray.UpdateItems(@a, @ix, @v)
SELECT SqlArray.FloatArray.ToString(@b)             -- returns [[1,3],[2,4],[0,0]]
```

Subarray function

Returns a subarray of an array. Collapse determines whether ranks with zero length are removed so the resulting subarray will have less indices than the original if any of the length values is 1.

Syntax:

SqlArray.*arraytype*.Subarray(@a, @offset, @length, @collapse)

Parameters:

@a is the original array in its binary representation.

@offset is an array indexing the starting element of the subarray.

@length is an array defining the rank lengths of the subarray.

@collapse is either 1 or 0. If it is 1 that means that the ranks of the resulting subarray with length equal to 1 will be collapsed so that the number of indices is reduced.

Returns:

The subarray in binary representation.

Example:

```
DECLARE @a varbinary(100) = SqlArray.FloatArray.Parse('[[1, 2, 3],[4, 5, 6],[7, 8, 9]]')
DECLARE @offset varbinary(100) = SqlArray.SmallIntArray.Vector_2(1, 1)
DECLARE @size varbinary(100) = SqlArray.SmallIntArray.Vector_2(2, 2)
DECLARE @s varbinary(100) = SqlArray.FloatArray.SubArray(@a, @offset, @size, 0)
SELECT SqlArray.FloatArray.ToString(@s)             -- returns [[5,6],[8,9]]
```

Subarrays function

Returns multiple subarrays of a big array. All subarrays must have the same size. Collapse determines whether ranks with zero length are removed so the resulting subarray will have less indices than the original if any of the length values is 1.

Syntax:

SqlArray.*arraytype*.Subarray(@a, @offsets, @length, @collapse)

Parameters:

@a is the original array in its binary representation.

@offsets is an array containing the indexes the starting elements of the subarrays to extract.

@length is an array defining the rank lengths of the subarrays.

@collapse is either 1 or 0. If it is 1 that means that the ranks of the resulting subarrays with length equal to 1 will be collapsed so that the number of indices is reduced.

Returns:

The subarrays in binary representation. The first index will index the subarrays.

Example:

```
DECLARE @a varbinary(100) = SqlArray.FloatArray.Parse('[[1, 2, 3],[4, 5, 6],[7, 8, 9]]')
DECLARE @offsets varbinary(100) = SqlArray.SmallIntArray.Parse('[[0, 0],[1, 1]]')
DECLARE @size varbinary(100) = SqlArray.SmallIntArray.Vector_2(2, 2)
DECLARE @s varbinary(100) = SqlArray.FloatArray.SubArrays(@a, @offsets, @size, 0)
SELECT SqlArray.FloatArray.ToString(@s)      -- returns [[[1,5],[2,6]],[[4,8],[5,9]]]
```

UpdateSubarray

Similar to Subarray except it updates the values and returns the updates array.

Syntax:

SqlArray.arraytype.UpdateSubarray(@a, @offset, @v)

Parameters:

@a is the original array.

@offset is an array containing the indices of the origin of the subarray to be updated.

@v is the new value of the subarray indexed by @offset

Returns:

The update array in binary representation.

Example:

```
DECLARE @a varbinary(100) = SqlArray.FloatArray.Parse('[[1, 2, 3],[4, 5, 6],[7, 8, 9]]')
DECLARE @offset varbinary(100) = SqlArray.SmallIntArray.Vector_2(1, 1)
DECLARE @s varbinary(100) = SqlArray.FloatArray.Parse('[[0, 0],[0, 0]]')
SET @a = SqlArray.FloatArray.UpdateSubarray(@a, @offset, @s)
SELECT SqlArray.FloatArray.ToString(@a)             -- returns [[1,2,3],[4,0,0],[7,0,0]]
```

UpdateSubarrays

Updates a series of subarrays in an array.

Syntax:

SqlArray.arraytype.UpdateSubarrays(@a, @offsets, @values)

Parameters:

@a is the original array that will be updated.

@offsets contains the indices of the subarrays to update

@values contains the new subarray values

Returns:

The updated array in binary representation.

Example:

```
DECLARE @a varbinary(100) = SqlArray.FloatArray.Parse('[[0, 0, 0],[0, 0, 0],[0, 0, 0]]')
DECLARE @offsets varbinary(100) = SqlArray.SmallIntArray.Parse('[[0, 0],[1, 1]]')
DECLARE @s varbinary(100) = SqlArray.FloatArray.Parse('[[[1,5],[2,6]],[[4,8],[5,9]]]')
SET @a = SqlArray.FloatArray.UpdateSubarrays(@a, @offsets, @s)
SELECT SqlArray.FloatArray.ToString(@a)             -- returns [[1,2,0],[4,5,6],[0,8,9]]
```

Converting arrays to table and vice versa

ToTable function

This function converts the array into a SQL table. ToTable returns indices as arrays.

Syntax:

SqlArray.*arraytype*.ToTable(@a)

Parameters:

@a is the array to convert to a table

Returns:

A table with the following columns.

li: linear index of the item
ix: index of the item as an array
v: value of the item

Example:

```
DECLARE @a varbinary(100) = SqlArray.FloatArray.Parse('[[1, 2, 3],[4, 5, 6],[7, 8, 9]]')
SELECT li, SqlArray.SmallIntArray.ToString(i), v FROM SqlArray.FloatArray.ToTable(@a)
```

li	ix	v
0	[0,0]	1
1	[1,0]	2
2	[2,0]	3
3	[0,1]	4
4	[1,1]	5
5	[2,1]	6
6	[0,2]	7
7	[1,2]	8
8	[2,2]	9

ToTable_n functions

These functions convert the array into a SQL table. ToTable_n returns indices in n columns containing scalars.

Syntax:

SqlArray.arraytype.ToTable(@a)
SqlArray.arraytype.ToTable_r(@a)

Parameters:

@a is the array to convert to a table

Returns:

A table with the following columns.

li: linear index of the item
in: index of the item (various number of scalar columns)
v: value of the item

Example:

```
DECLARE @a varbinary(100) = SqlArray.FloatArray.Parse('[[1, 2, 3],[4, 5, 6],[7, 8, 9]]')
SELECT * FROM SqlArray.FloatArray.ToTable_2(@a)
```

li	i0	i1	v
0	0	0	1
1	1	0	2
2	2	0	3
3	0	1	4
4	1	1	5
5	2	1	6
6	0	2	7
7	1	2	8
8	2	2	9

MatrixToTable_n functions

These function convert matrices to tables, the value of n can be between 2 and 10.

Syntax:

SqlArray.*arraytype*.MatrixToTable_n(@a)

Parameters:

@a is the array containing the matrix to convert to a table

Returns:

A table with the following columns.

i: index of the row

vn: value of the item in the n^{th} column

Example:

```
DECLARE @a varbinary(100) = SqlArray.FloatArray.Parse('[[1, 2, 3],[4, 5, 6],[7, 8, 9]]')
SELECT * FROM SqlArray.FloatArray.MatrixToTable_3(@a)
```

i	v0	v1	v2
0	1	2	3
1	4	5	6
2	7	8	9

Split function

Splits the array into equally sized subarrays and returns them row-by-row in a data table. An offset and subarrays size must also be specified.

Syntax:

SqlArray.arraytype.Split(@a, @offset, @size)

Parameters:

@a is the array to split

@offset is an array with the starting index of splitting

@size is an array with the size of the partitions to split into

Returns:

A table with the following columns:

ci: collapsed index

Example:

```
DECLARE @a varbinary(200) =
SqlArray.FloatArray.Parse('[[[1,2,3,4],[3,4,5,6]],[[5,6,7,8],[7,8,9,0]]]')
DECLARE @offset varbinary(100) = SqlArray.SmallIntArray.Vector_3(0,0,0)
DECLARE @size varbinary(100) = SqlArray.SmallIntArray.Vector_3(2,1,1)
SELECT
  SqlArray.SmallIntArray.ToString(ci) ci,
  SqlArray.SmallIntArray.ToString(i) i ,
  SqlArray.FloatArray.ToString(v) v
FROM SqlArray.FloatArray.Split(@a, @offset, @size, 1)
```

ci	i	v
[0,0,0]	[0,0,0]	[1,2]
[1,0,0]	[2,0,0]	[3,4]
[0,1,0]	[0,1,0]	[3,4]
[1,1,0]	[2,1,0]	[5,6]

[0,0,1]	[0,0,1]	[5,6]
[1,0,1]	[2,0,1]	[7,8]
[0,1,1]	[0,1,1]	[7,8]
[1,1,1]	[2,1,1]	[9,0]

ItemsToTable function

Accepts an array and another containing indices of items and returns the indexed items as a table.

Syntax:

SqlArray.*arraytype*.ItemsToTable(@a, @items)

Parameters:

@a is the array to sample from

@items is an array of indices of the elements to be returned

Returns:

A table with the following columns:

li: linear index of the element
i: the index of the element as an array
v: the value of the element

Example:

```
DECLARE @a varbinary(100) = SqlArray.FloatArray.Parse('[[1, 2, 3],[4, 5, 6],[7, 8, 9]]')
DECLARE @items varbinary(100) = SqlArray.SmallIntArray.Parse('[[0,0],[0,1],[2,2]]')
SELECT li, SqlArray.SmallIntArray.ToString(i) i, v FROM
SqlArray.FloatArray.ItemsToTable(@a, @items)
```

li	i	v
0	[0,0]	1
1	[0,1]	4
2	[2,2]	9

SubarraysToTable function

Accepts an array containing indices of items and returns the indexed items as a table.

Syntax:

SqlArray.*arraytype*.ItemsToTable(@a, @offsets, @size, @collapse)

Parameters:

@a is the array to sample from

@offsets is an array containing the indexes to the origin of the subarrays

@size if the size of the subarrays (same for all)

@collapse determines whether subarray ranks with length 1 are removed

Returns:

A table with the following columns:

li: linear index of the subarray
i: index of the origin of the subarray
v: value of the subarray

Example:

```
DECLARE @a varbinary(100) = SqlArray.FloatArray.Parse('[[1, 2, 3],[4, 5, 6],[7, 8, 9]]')
DECLARE @offsets varbinary(100) = SqlArray.SmallIntArray.Parse('[[0,0],[0,1],[2,2]]')
DECLARE @size varbinary(100) = SqlArray.SmallIntArray.Parse('[1,1]')
SELECT
  li,
  SqlArray.SmallIntArray.ToString(i) i,
  SqlArray.FloatArray.ToString(v) v
FROM SqlArray.FloatArray.SubarraysToTable(@a, @offsets, @size, 0)
```

li	i	V
0	[0,0]	[[1]]
1	[0,1]	[[4]]
2	[2,2]	[[9]]

ScaMul, ScaDiv, ScaAdd, ScaSub functions

Multiplies, divides, increases or decreases the value of every array element with a scalar value.

Syntax:

SqlArray.*arraytype*.ScaMul(@a, @m)
SqlArray.*arraytype*.ScaDiv(@a, @m)
SqlArray.*arraytype*.ScaAdd(@a, @m)
SqlArray.*arraytype*.ScaSub(@a, @m)

Parameters:

@a is the original array in binary representation

@m is a scalar value of the array's element type

Returns:

The modified array in binary representation.

Example:

```
DECLARE @a varbinary(100) = SqlArray.RealArray.Vector_3(1.0, 2.0, 3.0)
DECLARE @b varbinary(100) = SqlArray.RealArray.ScaMul(@a, 2)
SELECT SqlArray.RealArray.ToString(@b)       -- returns [2,4,6]
SET @b = SqlArray.RealArray.ScaDiv(@a, 2)
SELECT SqlArray.RealArray.ToString(@b)       -- returns [0.5,1,1.5]
```

```
SET @b = SqlArray.RealArray.ScaAdd(@a, 2)
SELECT SqlArray.RealArray.ToString(@b)       -- returns [3,4,5]
SET @b = SqlArray.RealArray.ScaSub(@a, 2)
SELECT SqlArray.RealArray.ToString(@b)       -- returns [-1,0,1]
```

FromScalar function

Creates an array with a given size and initializes the element values to a given scalar number.

Syntax:

SqlArray.*arraytype*.FromScalar(@size, @v)

Parameters:

@size is an array containing the size of the target array

@v is the scalar value to initialize the elements of the array to

Returns:

The initialized array in binary representation.

Example:

```
DECLARE @size varbinary(100) = SqlArray.SmallIntArray.Vector_3(2, 3, 2)
DECLARE @a varbinary(200) = SqlArray.FloatArray.FromScalar(@size, 1)
SELECT SqlArray.FloatArray.ToString(@a)
 -- returns [[[1,1],[1,1],[1,1]],[[1,1],[1,1],[1,1]]]
```

Mul, Div, Add, Sub functions

Multiplies, divides, sums or subtracts to arrays, itemwise. The two arrays must be the same size.

Syntax:

SqlArray.*arraytype*.Mul(@a, @b)

SqlArray.*arraytype*.Div(@a, @b)

SqlArray.*arraytype*.[Add](@a, @b)

SqlArray.*arraytype*.[Sub](@a, @b)

Parameters:

@a and @b are arrays of the same size in binary representation.

Returns:

An array of the same size as the original ones in binary representation.

Example:

```
DECLARE @a varbinary(100) = SqlArray.FloatArray.Vector_3(1.0, 2.0, 3.0)
DECLARE @b varbinary(100) = SqlArray.FloatArray.Vector_3(2.0, 4.0, 6.0)
SELECT SqlArray.FloatArray.ToString(SqlArray.FloatArray.Mul(@a, @b))
 -- returns [2,8,18]
SELECT SqlArray.FloatArray.ToString(SqlArray.FloatArray.Div(@a, @b))
 -- returns [0.5,0.5,0.5]
```

```
SELECT SqlArray.FloatArray.ToString(SqlArray.FloatArray.[Add](@a, @b))
 -- returns [3,6,9]
SELECT SqlArray.FloatArray.ToString(SqlArray.FloatArray.Sub(@a, @b))
 -- returns [-1,-2,-3]
```

DotProd function

Calculates the dot product of two vectors of the same size.

Syntax:

SqlArray.*arraytype*.DotProd(@a, @b)

Parameters:

@a and @b are vectors (one-index arrays) with the same length

Returns:

The dot product (a scalar).

Example:

```
DECLARE @a varbinary(100) = SqlArray.FloatArray.Vector_3(1.0, 2.0, 3.0)
DECLARE @b varbinary(100) = SqlArray.FloatArray.Vector_3(2.0, 4.0, 6.0)
SELECT SqlArray.FloatArray.DotProd(@a, @b)    -- returns 28
```

CrossProd function

Calculates the cross product of two 3 dimensional vectors.

Syntax:

SqlArray.*arraytype*.CrossProd(@a, @b)

Parameters:

@a and @b are vectors with 3 elements.

Returns:

The cross product of the vectors as a vector, in binary representation.

Example:

```
DECLARE @a varbinary(100) = SqlArray.FloatArray.Vector_3(0.0, 2.0, 3.0)
DECLARE @b varbinary(100) = SqlArray.FloatArray.Vector_3(2.0, 4.0, 6.0)
DECLARE @c varbinary(100) = SqlArray.FloatArray.CrossProd(@a, @b)
SELECT   SqlArray.FloatArray.ToString(@c)     -- returns [0,6,-4]
```

TensorProd function

Calculates the tensor product of two vectors.

Syntax:

SqlArray.*arraytype*.TensorProd(@a, @b)

Parameters:

@a and @b are vectors (one-index arrays) of the same length.

Returns:

The tensor product (a rectangular matrix) of the two vectors, in binary representation.

Example:

```
DECLARE @a varbinary(100) = SqlArray.FloatArray.Vector_3(0.0, 2.0, 3.0)
DECLARE @b varbinary(100) = SqlArray.FloatArray.Vector_3(2.0, 4.0, 6.0)
DECLARE @c varbinary(100) = SqlArray.FloatArray.TensorProd(@a, @b)
SELECT   SqlArray.FloatArray.ToString(@c)
-- returns [[0,4,6],[0,8,12],[0,12,18]]
```

MatProd function

Calculates the product of two matrices. The matrices have to be compatible.

Syntax:

SqlArray.*arraytype*.MatProd(@a, @b)

Parameters:

@a are @b are matrices in binary representation that are compatible for the product operator.

Returns:

The metrix products in binary representation

Example:

```
TBW
```

InnerProd function

Takes the tensor product of the two arrays and sums over the first indices. The range of the first indices must be the same.

Syntax:

SqlArray.*arraytype*.InnerProd(@a, @b)

Parameters:

@a and @b are arrays in binary representation.

Returns:

The resulting matrix in binary representation.

Example:

```
DECLARE @a varbinary(300) = SqlArray.FloatArray.Parse('[[1,2,3],[4,5,6],[7,8,9]]')
DECLARE @b varbinary(300) = SqlArray.FloatArray.Parse('[[7,4,5],[6,7,3],[4,8,9]]')
DECLARE @c varbinary(300) = SqlArray.FloatArray.InnerProd(@a, @b)
```

```
SELECT   SqlArray.FloatArray.ToString(@c)
 -- returns [[30,78,126],[29,77,125],[47,110,173]]
```

Avg, Min, Max, StDev, Sum, Var, Med functions

Calculates the statistics of the array elements.

Syntax:

SqlArray.*arraytype*.Avg(@a)
SqlArray.*arraytype*.Min(@a)
SqlArray.*arraytype*.Max(@a)
SqlArray.*arraytype*.StDev(@a)
SqlArray.*arraytype*.Sum(@a)
SqlArray.*arraytype*.Var(@a)
SqlArray.*arraytype*.Med(@a)

Parameters:

@a is an array in binary representation.

Returns:

Average, minimum, maximum, standard deviation, variance or median of the array elements.

Example:

```
DECLARE @a varbinary(100) = SqlArray.FloatArray.Matrix_3(1,2,3,4,5,6,7,8,9)
SELECT SqlArray.FloatArray.Avg(@a)       -- returns 3.5
```

FromTable function

Converts a table containing scalar values of the individual elements into an array. The data must reside in a table (which can be a temporary table) prior to calling the function. The size of the array has to be known early on and passed to the function. The source table must have one of the following formats:

1. linear index, value
2. indices as an array, value
3. i1, i2, … in, value

The order of the columns in the table is important, column names are ignored. Data types of the columns must be compatible with the array type: indices must be smallint for short arrays and int for max arrays. The value type must match the element type of the array.

Syntax:

SqlArray.*arraytype*.FromTable(@tablename, @size)

Parameters:

@table is name of the table to read the data from, can be a temporary table

@size is the size of the array

Returns:

The elements in an array, in binary representation.

Example:

```
CREATE TABLE ##arr (li smallint, v float)

INSERT ##arr (li, v)
VALUES (0,1), (1,2), (2,3), (3,4), (6,7), (7,8), (8,9)
```

```
DECLARE @size varbinary(100) = SqlArray.SmallIntArray.Vector_2(3, 3)
DECLARE @a varbinary(100) = SqlArray.FloatArray.FromTable('##arr', @size)
SELECT SqlArray.FloatArray.ToString(@a)
        -- returns [[1,2,3],[4,0,0],[7,8,9]]

DROP TABLE ##arr
```

FromSubarrayTable function

Converts a table containing small arrays into a larger array. The data must reside in a table (which can be a temporary table) prior to calling the function. The size of the array has to be known early on and passed to the function. The source table must have the following format:

1. offset as an array, value as an array

The order of the columns in the table is important, column names are ignored. Data types of the columns must be compatible with the array type: indices must be smallint for short arrays and int for max arrays. The element type of the value array must match the element type of the resulting array.

Syntax:

SqlArray.*arraytype*.FromSubarrayTable(@tablename, @size)

Parameters:

@table is name of the table to read the data from, can be a temporary table

@size is the size of the array

Returns:

The array, built up from the subarrays, in binary representation.

Example: