

# Extending SQL Server Reporting Services with SQL CLR Table-Valued Functions

SQL Server Technical Article

Writer: Ryan Ackley

Published: February 2007

Applies To: Microsoft® SQL Server™ 2005 Reporting Services

Summary: This article describes how to take advantage of SQL CLR table-valued functions to combine different types of data sources to create rich and exciting SQL Server Reporting Services reports.

### Copyright

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This White Paper is for informational purposes only. **MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT.**

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Unless otherwise noted, the example companies, organizations, products, domain names, e-mail addresses, logos, people, places and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, email address, logo, person, place or event is intended or should be inferred.

 2007 Microsoft Corporation. All rights reserved.

Microsoft, MapPoint, SQL Server, Visual Basic, Visual C#, Visual Studio, and Windowsare either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

### Table of Contents

[Introduction 1](#_Toc158608480)

[Extending Reporting Services 1](#_Toc158608481)

[Using Table-Valued Functions 1](#_Toc158608482)

[Using Table-Valued Functions as an Alternative to Data Processing Extensions 4](#_Toc158608483)

[Advantages 4](#_Toc158608484)

[Disadvantages 4](#_Toc158608485)

[Using Table-Valued Functions with SQL Reporting Services 4](#_Toc158608486)

[Event Log Report 6](#_Toc158608487)

[Web Services and Table-Valued Functions 7](#_Toc158608488)

[Using the MapPoint Web Service 7](#_Toc158608489)

[Deploying and Debugging 12](#_Toc158608490)

[Creating a Report Using the MapPoint Web Service 13](#_Toc158608491)

[Conclusion 15](#_Toc158608492)

#### Introduction

A new feature of Microsoft® SQL Server™ 2005 is its integration with the Microsoft .NET Framework common language runtime (CLR). This allows .NET Framework classes and functions to be incorporated into Transact-SQL statements and queries.

There are several different CLR integration mechanisms available. These include:

* CLR user-defined functions (including table-valued functions)
* CLR user-defined types
* CLR stored procedures
* CLR triggers

This white paper will show how CLR table-valued functions can be used to create report data from various sources in addition to databases to create robust Reporting Services reports.

#### Extending Reporting Services

There are several ways to extend Reporting Services and integrate CLR functionality using the .NET Framework, including the following:

* Delivery extensions, which deliver reports in response to an event.
* Rendering extensions, which display a report in a format other than one that is supported by Reporting Services.
* Security extensions, which provide your own authentication and authorization mechanism for viewing and managing reports.
* Data processing extensions, which can be developed to process data from data sources that are not supported by Reporting Services.
* Custom report items, which are customized server controls that can be embedded in reports to provide additional functionality beyond the built-in controls.

This paper discusses how to implement table-valued functions to process data as an alternative to using data processing extensions. For more information about extending Reporting Services, see [Report Services Extensions](http://go.microsoft.com/fwlink/?LinkID=74085) in SQL Server Books Online.

#### Using Table-Valued Functions

Table-valued functions are used to programmatically create a table at run time. The tables they create can then be used in Transact-SQL query statements like any other database table. When table-valued functions were introduced in SQL Server 2000, they could only be created using Transact-SQL. The following is an example of a table-valued function implemented in Transact-SQL.

CREATE function EmployeeNames()

returns @employeeNames table (id int, name nvarchar(20), )

as begin

INSERT @employeeNames values(1, 'Ryan');

INSERT @employeeNames values(2, 'John');

INSERT @employeeNames values(3, 'Bob');

return

end

The function can then be referenced from a select statement as if it were a table:

SELECT name from EmployeeNames() where id = 1

The query returns:

name

--------------------

Ryan

While this is useful, it is limited by the Transact-SQL language, which was designed for use with relational data. If you try to leave that domain, Transact-SQL becomes somewhat inflexible. In SQL Server 2005, you can now use your favorite .NET Framework language to create table-valued functions, opening up some amazing possibilities. Programmers now can abstract any information that they want into a relational database table.

For example, the following code is a SQL Server 2005 table-valued function implemented in Microsoft Visual C# that returns a table created from the system event logs.

using System;

using System.Data.Sql;

using Microsoft.SqlServer.Server;

using System.Collections;

using System.Data.SqlTypes;

using System.Diagnostics;

public class TabularEventLog

{

 [SqlFunction(TableDefinition="logTime datetime,Message ” +

 “nvarchar(4000),Category nvarchar(4000),InstanceId bigint",

 Name="ReadEventLog", FillRowMethodName = "FillRow")]

 public static IEnumerable InitMethod(String logname)

 {

 return new EventLog(logname, Environment.MachineName).Entries;

 }

 public static void FillRow(Object obj, out SqlDateTime timeWritten,

 out SqlChars message, out SqlChars category,

 out long instanceId)

 {

 EventLogEntry eventLogEntry = (EventLogEntry)obj;

 timeWritten = new SqlDateTime(eventLogEntry.TimeWritten);

 message = new SqlChars(eventLogEntry.Message);

 category = new SqlChars(eventLogEntry.Category);

 instanceId = eventLogEntry.InstanceId;

 }

}

The table-valued function is implemented as two static methods on the **TabularEventLog** class. The first method (**InitMethod**) is given the **SqlFunction** attribute to designate it as the entry point for the table-valued function. This method must return an **IEnumerable** or **IEnumerator** object. This object contains the data that will be used to fill the return table. When executing the function, SQL Server will iterate through each object in the IEnumerator object and use this to fill a row of data. It does this by passing the object to the second method in the class, **FillRow**. This method converts the object into a row in the return table. This method is specified in the *FillRowMethodName* parameter of the **SqlFunction** attribute.

Additional metadata is defined in the arguments of the **SqlFunction** attribute. In the preceding example, the column names and types are defined as well as the name of the return table in this attribute.

After deploying this function to an instance of SQL Server, you can run the following query to see the last ten items in the application log.

SELECT TOP 10 T.logTime, T.Message, T.InstanceId

FROM dbo.ReadEventLog(N'Application') as T

The results are shown in Figure 1.



Figure 1

For more in-depth information, see [CLR Table-Valued Functions](http://go.microsoft.com/fwlink/?LinkID=74086) in SQL Server Books online.

#### Using Table-Valued Functions as an Alternative to Data Processing Extensions

Reporting Services data processing extensions allow a data source to be modeled by implementing a set of ADO.NET interfaces. This is conceptually similar to how table-valued functions can be used with Reporting Services. Table-valued functions have important advantages over data processing extensions.

##### Advantages

First, table-valued functions can be much easier to implement than data processing extensions. Only two methods have to be created to implement a table-valued function. You have to implement a number of interfaces for a data processing extension. Also, the deployment model is more straightforward. Microsoft Visual Studio® 2005 can automatically deploy a .NET Framework table-valued function to SQL Server, where it becomes immediately available for use from Reporting Services. To deploy a data processing extension, you must copy the assembly to the client and the report server, and edit XML configuration files in both places.

Another important advantage of a table-valued function is that it can be part of a join in the database where it is attached. This means that relational data in SQL Server can be mixed and filtered with the custom data defined in the function before it is placed into the report. This is impossible with a data processing extension because Reporting Services does not support join queries between data sources.

##### Disadvantages

Data processing extensions are much more powerful and flexible than table-valued functions. A table-valued function can model only a single database table, whereas a data processing extension can model the equivalent of an entire database. Also, a data processing extension functions as a fully custom data source and it can have its own query language and connection syntax. Using SQL as a query language isn’t always ideal for different types of data. For example, Reporting Services includes a data processing extension for XML data that uses a query language similar to XPath. Data extensions are useful when a developer wants full control over the data access code path.

#### Using Table-Valued Functions with SQL Reporting Services

There are three things that you must do before you can use table-valued functions with Reporting Services. First, SQL Server must be configured to allow CLR integration. Next, the table-valued function must be developed in Visual Studio. Finally, the function has to be deployed to an instance of SQL Server.

For SQL Server to allow CLR integration, a flag has to be set either by using the SQL Server Surface Area Configuration tool or by running a query.

To configure SQL Server to allow CLR integration

1. Click the **Start** button, point to **All Programs**, point to **Microsoft SQL Server 2005**, point to **Configuration Tools**, and then click **Surface Area Configuration**.
2. In the SQL Server 2005 Surface Area Configuration tool, click **Surface Area Configuration for Features**.
3. Select your server instance, expand the **Database Engine** options, and then click **CLR Integration**.
4. Select **Enable CLR integration**.

Alternatively, you can run the following query in SQL Server (this query requires ALTER SETTINGS permission).

USE master

sp\_configure 'show advanced options', 1;

GO

RECONFIGURE;

GO

sp\_configure 'clr enabled', 1;

GO

RECONFIGURE;

GO

To develop a table-valued function

To develop a table-valued function, create a new SQL Server project in Visual Studio. To create a SQL Server project, open the New Project dialog box, expand Visual C#, and then select Database. You should be prompted for database connection information. For more information, see [How to: Create a SQL Server Project](http://go.microsoft.com/fwlink/?LinkID=74088) in SQL Server Books Online. After you set up your database connection, you can write a table-value function Create an empty .cs file in your project with the name EventLog.cs and copy and paste the example function from the previous section into that file.

To deploy a table-valued function

To deploy, you must register the function and the assembly that contains it with a SQL Server instance. This is done using Transact-SQL commands. The following script registers the **tvfEventLogs** assembly and the **ReadEventLog** function:

CREATE ASSEMBLY tvfEventLog

 FROM'D:\assemblies\tvfEventLog\tvfeventlog.dll'

 WITH PERMISSION\_SET = SAFE

 GO

CREATE FUNCTION ReadEventLog(@logname nvarchar(100))

 RETURNS TABLE (logTime datetime,Message nvarchar(4000),

 Category nvarchar(4000),InstanceId bigint)

 AS EXTERNAL NAME tvfEventLog.TabularEventLog.InitMethod GO

Alternatively, you can deploy your assembly directly from Visual Studio by right-clicking the project in Solution Explorer and selecting the **Deploy** option. Visual Studio uses the **SqlFunction** attribute to determine the function signature and other necessary metadata automatically.

##### Deployment Permissions

SQL Server uses permission sets to run code securely in its hosted environment. When you create a database project in Visual Studio, the default permission set is SAFE. This is the only permission set that allows you to deploy directly from Visual Studio with no other configuration required. To give your assembly a permission set other than SAFE you must give your assembly a strong name and perform other configuration steps before deploying the assembly to the database.

There are three possible permission sets available when registering an assembly with SQL Server: SAFE, EXTERNAL ACCESS, and UNSAFE.

* SAFE allows only internal computation and local data access from code in the assembly.
* EXTERNAL ACCESS allows access to external system resources such as files, network resources, and the registry.
* UNSAFE allows code in the assembly to run unrestricted.

To deploy your assembly with a permission set other than SAFE, you must follow some additional steps. First, an asymmetric key must be created from the assembly you want to register with SQL Server. Next, use that key to create a login. Finally, the appropriate permission set must be granted to this login. The following Transact-SQL statements use these steps to grant the UNSAFE permission set to the assembly created from the tvfEventLogs example in the previous section.

USE master

GO

CREATE ASYMMETRIC KEY EventLogKey FROM EXECUTABLE FILE =

'D:\assemblies\tvfEventLog\tvfeventlog.dll'

CREATE LOGIN EventLogLogin FROM ASYMMETRIC KEY EventLogKey

GRANT UNSAFE ASSEMBLY TO EventLogLogin

GO

This has to be done only once, not every time you deploy the assembly. For more information about using the different permission sets and registering assemblies with SQL Server, see [CLR Integration Code Access Security](http://msdn2.microsoft.com/en-us/library/ms345101.aspx) and [Creating an Assembly](http://go.microsoft.com/fwlink/?LinkID=74089) in SQL Server Books Online.

##### Event Log Report

After you deploy the table-valued function, a virtual table of the system event log entries for the computer is added to the database. Since SQL Server treats the function just like a table, it can be used seamlessly in Reporting Services.

After you deploy the assembly, use Visual Studio to create a new Reporting Services project. (If you aren’t familiar with creating a report using Reporting Services Report Designer, see the [Reporting Services Tutorials](http://go.microsoft.com/fwlink/?LinkId=74090) in SQL Server Books Online.) Create a report with a SQL Server data source that connects to the same database where the function was installed. Next, create a dataset that uses the following query:

SELECT TOP 10 T.logTime, T.Message, T.InstanceId

FROM dbo.ReadEventLog(N'Security') as T

WHERE T.Category = N'Logon/Logoff'

After the dataset is defined, add a table data region to the report layout and add the fields from the dataset to the table detail row. Finally, run the report. It will show the last ten security events in the Logon/Logoff category for the local computer. Figure 2 shows an example of the report.



Figure 2

This simple example can be expanded to include other computer management and monitoring tasks. For example, a table-valued function can be created to parse Microsoft Internet Information Services (IIS) logs. Reporting Services can then be used to create a Web traffic monitoring application.

#### Web Services and Table-Valued Functions

One interesting feature of table-valued functions is the ability to pull data from Web services. This can be used to create unconventional reports. I will demonstrate how to use the Microsoft MapPoint® Web service in a table-value function and join this with data in the AdventureWorks database to render spatial data onto a map and render it in a report.

##### Using the MapPoint Web Service

It is necessary to obtain a free developer account from Microsoft before starting to develop against the MapPoint Web service. This can be obtained by visiting the [MapPoint customer service site](https://mappoint-css.partners.extranet.microsoft.com/MwsSignup/Eval2.aspx). A good place to get information before starting to develop against the Web service is [Getting Started with the MapPoint Web Service SDK](http://msdn2.microsoft.com/en-us/library/aa492550.aspx).

You will need to add a Web Reference to your project using Visual Studio that points to the .wsdl file on the MapPoint staging server. For more information about configuring a Web Reference for the MapPoint Web Service, see [Accessing the MapPoint Web Service SOAP API](http://msdn2.microsoft.com/en-us/library/aa491863.aspx).

MapPoint provides four services, each having its own SOAP endpoint:

* The Common Service provides functionality that can be used by the other services. This is used to retrieve metadata and for utility functions.
* The Find Service can be used to search for locations, find the latitude and longitude of an address (“geocoding”), and to find points of interest near a location.
* The Routing Service routes driving directions from one location to another.
* The Render Service can be used to create a map image using location and routing information.

##### MapPoint Web Service Table-Valued Function

Ultimately, I want my table-valued function to use the MapPoint Web service to perform the following tasks:

1. Use the Find Service to find the latitude and longitude of an AdventureWorks bicycle shop.
2. Use the Find Service to find the five closest automatic teller machines (ATMs) to this latitude and longitude.
3. Use the Routing Service to find the route from the store location to the ATM.
4. Use the Render Service to render this route on a map.

First, I must define a table-valued function called **GetProximity**. The following Transact-SQL code shows the signature of my table-valued function:

CREATE FUNCTION GetProximity(@city nvarchar(200), @state nvarchar(2),

 @count int, @entityTypeName nvarchar(200))

RETURNS TABLE

(HitName nvarchar(200), HitAddress nvarchar(200), MapImage varbinary(max))

**GetProximity** takes a city name and a two-digit state code for the initial location. It takes the number of entities to return and an entity-type name for which to search. It searches for the *n* closest entities where *n* is specified by the count parameter and the entity type is specified by the entityTypeName parameter. It returns a table containing columns for the name, address, and a map (binary image) that contains directions to each entity.

The C# method signatures look like this:

public static IEnumerable InitMap(string city, string state, int count,

 string entityTypeName)

public static void FillRow(Object obj, out SqlChars name, out SqlChars

 address, out SqlBinary map)

Note that the **nvarchar** Transact-SQL data type maps to the **SqlChars** .NET Framework data type, and the **varbinary** Transact-SQL data type maps to the **SqlBinary** .NET Framework data type. For a complete list of the mappings between data types, see the documentation for the [System.Data.SqlTypes](http://msdn2.microsoft.com/en-us/library/system.data.sqltypes.aspx) namespace.

In the **InitMap** method, I convert the city and state into a latitude and longitude. Next, I find all entities close to this coordinate. Finally, I find driving directions between the initial location and the found entity. The return value is an array of **Route** objects that encapsulate the driving directions.

public static IEnumerable InitMap(string city, string state, int count, string entityTypeName)

{

 FindServiceSoap find = new FindServiceSoap();

 find.PreAuthenticate = true;

 find.Credentials = new NetworkCredential(username, passwd);

 // Geocode the initial city and state

 FindAddressSpecification findSpec = new FindAddressSpecification();

 Address findAddr = new Address();

 findAddr.CountryRegion = "US";

 findAddr.Subdivision = state;

 findAddr.PrimaryCity = city;

 findSpec.InputAddress = findAddr;

 findSpec.DataSourceName = "MapPoint.NA";

 findSpec.Options = new FindOptions();

 findSpec.Options.ThresholdScore = 0.45;

 FindResults results = find.FindAddress(findSpec);

 if (results.NumberFound > 0)

 {

 // If the city and state exist, get the latitude and longitude

 Location startLocation = results.Results[0].FoundLocation;

 LatLong startPoint = startLocation.LatLong;

 // Find the nearby entities

 FindNearbySpecification findNearby = new FindNearbySpecification();

 FindFilter filter = new FindFilter();

 filter.EntityTypeName = entityTypeName;

 findNearby.Filter = filter;

 FindOptions options = new FindOptions();

 options.Range = new FindRange();

 // Set the count limit

 options.Range.Count = count;

 findNearby.Options = options;

 findNearby.DataSourceName = "NavTech.NA";

 findNearby.LatLong = startPoint;

 findNearby.Distance = 10.0;

 results = find.FindNearby(findNearby);

 Route[] routes = new Route[results.Results.Length];

 RouteServiceSoap routeService = new RouteServiceSoap();

 routeService.PreAuthenticate = true;

 routeService.Credentials = new NetworkCredential(username,passwd);

 RouteSpecification spec = new RouteSpecification();

 spec.DataSourceName = "MapPoint.NA";

 // Create the route to each entity

 spec.Segments = new SegmentSpecification[2];

 spec.Segments[0] = new SegmentSpecification();

 spec.Segments[0].Waypoint = new Waypoint();

 spec.Segments[0].Waypoint.Location = startLocation;

 spec.Segments[0].Waypoint.Name = "start";

 for (int x = 0; x < results.Results.Length; x++)

 {

 spec.Segments[1] = new SegmentSpecification();

 spec.Segments[1].Waypoint = new Waypoint();

 spec.Segments[1].Waypoint.Location = results.Results[x].FoundLocation;

 spec.Segments[1].Waypoint.Name = "end";

 routes[x] = routeService.CalculateRoute(spec);

 }

 return routes;

 }

 return null;

}

In the **FillRow** method, I use the Render service to convert each **Route** object into a map image. I then populate a row using this image and the location data of the entity.

public static void FillRow(Object obj, out SqlChars name, out SqlChars address, out SqlBinary map)

{

 Route route = (Route)obj;

 // build the address string

 Address endAddress = route.Specification.Segments[1].Waypoint.Location.Address;

 string entityAddress = endAddress.AddressLine;

 string enitityCity = endAddress.PrimaryCity;

 string entityState = endAddress.Subdivision;

 string entityName = route.Specification.Segments[1].Waypoint.Location.Entity.DisplayName;

 // Assign the values of two of the columns

 name = new SqlChars(entityName);

 address = new SqlChars(entityAddress + ' ' + enitityCity + ' ' + entityState);

 // Get the view of the route

 ViewByHeightWidth view = route.Itinerary.View.ByHeightWidth;

 RenderServiceSoap renderService = new RenderServiceSoap();

 renderService.PreAuthenticate = true;

 renderService.Credentials = new NetworkCredential(username, passwd);

 // Render the map with the route

 MapSpecification mapSpec = new MapSpecification();

 mapSpec.DataSourceName = "MapPoint.NA";

 mapSpec.Views = new MapView[]{view};

 mapSpec.Route = route;

 // Assign the map image to the map column

 MapImage[] image = renderService.GetMap(mapSpec);

 map = new SqlBinary(image[0].MimeData.Bits);

}

##### Deploying and Debugging GetProximity

Deploying a table-valued function that uses Web services is more involved than the previous example. The comprehensive steps to deploy a table valued function that uses Web services are as follows:

1. Configure the project containing the **GetProximity** function to pre-generate the XML serialization assembly. When the .NET Framework makes Web service calls, it dynamically generates an assembly to handle the serialization and de-serialization of the SOAP XML. This presents a problem because the SQL Server CLR host does not allow assemblies to be dynamically loaded at run time. Therefore, the XML serialization assembly for the Web service calls must be generated at compile time and registered with SQL Server. To pre-generate this assembly from Visual Studio, from the **Project** menu, click **Properties** and select **Build**. Set **Generate serialization assembly** to **On**. The XML serialization DLL will be built with your project and added to the bin directory. It is given the name [ProjectName].XmlSerializers.dll.
2. Add the System.Security.AllowPartiallyTrustedCallers attribute to the assembly. This can be done by adding the following line to AssemblyInfo.cs in the Project:

[assembly: System.Security.AllowPartiallyTrustedCallers]

This allows the XML serialization assembly to talk to the main assembly containing the **GetProximity** function.

1. Register the XML serialization DLL created in step 1 with SQL Server. The SAFE permission set will be sufficient.
2. Create an asymmetric key for the DLL that contains the **GetProximity** table-valued function.
3. Create a login for the asymmetric key and grant it the EXTERNAL ACCESS permission set.
4. Register the assembly containing **GetProximity** with the EXTERNAL ACCESS permission set.
5. Register the table-valued function, **GetProximity**.

Since there is a relatively long and complex dependency chain, I abandoned Visual Studio’s deployment mechanism in favor of a Transact-SQL script that runs as a post-build step that performs the deployment steps 3-7. It is included with the sample project.

Debugging table-valued functions is very straightforward. Database projects have a **Test Scripts** directory. Scripts can be added to this directory and run directly from Visual Studio. After the function has been successfully deployed, you can create a Transact-SQL query that calls the function and step through the C# code for the function without leaving Visual Studio.

To test **GetProximity**, create a test script called “Test.sql” in the **Test Scripts** directory and add the following query to the file:

SELECT \* FROM GetProximity(‘Redmond’, ‘WA’, 5, ‘SIC3578’)

Notice the function arguments. I am centering my proximity query on the city of Redmond in the State of Washington so I used "Redmond" for the @city argument and "WA" for the @state argument. I provided the number 5 for the @count value, which is the number of entities I want returned. I also provided the value "SIC3578" for the @entityTypeName argument, which is the entity name for ATMs in the MapPoint data source I am using. For more information on MapPoint data sources and entity types, see [MapPoint Data Sources](http://msdn2.microsoft.com/en-us/library/ms982193.aspx).

To run the query in Visual Studio, right-click the Test.sql file in Solution Explorer and select **Debug Script**. You should get something similar to the following results in the Visual Studio Output window:

HitName HitAddress MapImage

----------------------------------------------------------------------

Woodgrove Bank 8502 160th Ave NE Redmond WA <BINARY>

Woodgrove Bank 16025 NE 85th St Redmond WA <BINARY>

Woodgrove Bank 16150 NE 85th St Redmond WA <BINARY>

Woodgrove Bank 8867 161st Ave NE Redmond WA <BINARY>

Woodgrove Bank 15600 Redmond Way Redmond WA <BINARY>

No rows affected.

(5 row(s) returned)

To debug the **GetProximity** function, set a breakpoint in the C# code for the function and run the script again. The execution will halt at the specified point and you can debug it as you would any other managed process.

##### Creating a Report Using the MapPoint Web Service

The AdventureWorks sample database that comes with SQL Server 2005 represents a fictitious bicycle and bicycle accessories manufacturer that sells to retail stores throughout the United States. For this example, Adventure Works Cycles has decided to stop accepting credit cards or checks. From now on, they would like all invoices to be paid by cash only. As a service to their customers during this transition, they will be creating a report that shows an address and a map to the five closest ATMs to their customers’ store locations. This is not a realistic scenario, but it serves the purpose of demonstrating how to join a traditional data source (SQL database) with a non-traditional data source (MapPoint Web service) by using table-valued functions.

The first step in creating our report is to create a new Report Server project in Visual Studio and specify a data source. The data source for my report is the SQL Server 2005 AdventureWorks sample database. It has the MapPoint table valued-function that I created previously installed. There is one dataset for the report. It contains fields for the store name, store city, store state, ATM name, ATM address, and ATM directions map.

For each store, we will want to call **GetProximity** and get the five closest ATMs. In SQL Server 2005, there is the new APPLY clause type to handle doing this. This is a little different than a join because we want to join on the function arguments instead of the function results. This means that the table-valued function is called for each row returned by the left side of the APPLY. A union of the function results can then be joined to the rest of the query. Following is the Transact-SQL query for the report dataset.

SELECT TOP(40) Sales.Store.Name, Person.Address.City,

 Person.StateProvince.StateProvinceCode, GetProximity\_1.HitName,

 GetProximity\_1.HitAddress, GetProximity\_1.MapImage

 FROM Sales.CustomerAddress

 INNER JOIN Person.Address

 ON Sales.CustomerAddress.AddressID = Person.Address.AddressID

 AND Sales.CustomerAddress.AddressID = Person.Address.AddressID

 INNER JOIN Sales.Store

 INNER JOIN Sales.StoreContact ON Sales.Store.CustomerID = Sales.StoreContact.CustomerID

 ON Sales.CustomerAddress.CustomerID = Sales.StoreContact.CustomerID

 INNER JOIN Person.StateProvince ON Person.Address.StateProvinceID = Person.StateProvince.StateProvinceID

 AND Person.Address.StateProvinceID = Person.StateProvince.StateProvinceID

CROSS APPLY dbo.GetProximity(Person.Address.City,

 Person.StateProvince.StateProvinceCode, 5, 'SIC3578') AS GetProximity\_1

Notice the use of CROSS APPLY to link the **GetProximity** function arguments with the other query data represented by Person.Address.City and Person.StateProvince.StateProvinceCode.

**Note** To use the APPLY clause in a dataset query, you must use the generic query designer. The GUI-based query designer cannot display it graphically and it will throw an exception.

My report design uses two nested lists. The inner list contains a text box for the ATM name and address and an image for the map. The image in my report is set to **AutoSize** so that it will grow depending on the size of the image coming from the Web service. The outer list contains text boxes for the store name and location. The outer list is grouped by store name. An image of my report in Layout mode is shown in Figure 3.



Figure 3

Figure 4 shows the rendered report with the maps to the ATM locations.



Figure 4

#### Conclusion

This paper has shown how table-valued functions in SQL Server can be used to extend SQL Server Reporting Services data access functionality. Table-valued functions provide flexibility to programmers and report designers to enable reporting scenarios where data is not stored directly in database tables.

**About the author**

**Ryan Ackley** is a Software Development Engineer with the Microsoft SQL Server Business Intelligence group. His main area is the data and report processing engine of SQL Server Reporting Services.

For more information:

<http://msdn2.microsoft.com/en-us/sql/aa336316.aspx>

Did this paper help you? Please give us your feedback. On a scale of 1 (poor) to 5 (excellent), how would you rate this paper?