ABSTRACT
The integration of the .NET Common Language Runtime (CLR) into the SQL Server DBMS enables rich business logic written in modern .NET languages to run close to the data. Database application developers can write business logic as functions, stored procedures, and triggers. They can also extend the native capabilities of the DBMS by adding new scalar data types, and aggregates. A previous paper [2] described the architecture and design principles of the integration of the CLR inside SQL Server. Here we present new aspects of this work. First, we describe the extensibility contracts for user-defined types and aggregates in detail. Second, we present the advances to the CLR integration in SQL Server 2008 which significantly enhances the breadth of applications supported by SQL Server. In particular, we describe the support for large (greater than 8000 byte) user-defined types and aggregates, multiple-input user-defined aggregates, and order-aware table valued functions. Third, we show how we leveraged scalar type extensibility to provide a hierarchical identifier data type that enables encoding of keys describing hierarchies as well as built-in support for spatial applications. This support includes both flat- and round-earth spatial types, as well as a spatial index. Fourth, we present how we use Language Integrated Query (LINQ) enhancements in .NET languages to improve developer productivity when creating routines that require data access. Finally, we present preliminary performance results showing the efficiency of streaming TVFs and aggregates relative to equivalent native features.

Categories and Subject Descriptors:
H.2 [Database Management], D.3 [Programming Languages]

General Terms: Algorithms, Management, Design, Languages

Keywords: Server Programming, Extensibility, Abstract Data Types

1. INTRODUCTION
The ability to run application code inside the database in a reliable, secure, scalable, and efficient manner adds significant flexibility to the design of applications that require complex business logic to run close to the data and cannot afford the cost of shipping data to a middle-tier process to perform computation outside the database. This is becoming more important as data volumes continue to explode from terabytes to petabytes and exabytes.

A previous paper [2] described the architecture and design principles of the integration of the CLR inside SQL Server and briefly mentioned some of the features enabled by this integration. This paper provides a detailed description of the extensibility contracts for functions, types, and aggregates. Stored procedures, triggers, and functions usually need to execute SQL queries and updates. In [2] we presented how data access was achieved via an in-process version of the ADO.NET API. This paper presents how the new LINQ query expressions are used to enhance programmers’ productivity when creating routines that need data access.

The rest of the paper is organized as follows. Section 2 recaps some basic .NET and CLR integration concepts. Section 3 presents the extensibility features based on the Visual Studio tools. Sections 4 and 5 describe user-defined types and user-defined aggregates and their extensibility contracts, including multi-input aggregates. These sections include preliminary performance results demonstrating the efficiency of user-defined aggregates and TVFs relative to equivalent features implemented natively by SQL Server. Section 6 describes the in-proc data access strategy, including Language Integrated Query (LINQ). Section 7 describes support for hierarchical and spatial data added natively to SQL Server 2008 using the extensibility framework described in the previous sections. Section 8 describes the development experience for the extensibility features based on the Visual Studio tools suite. Section 9 presents a summary of previous work and Section 10 presents our conclusions and future work.

2. BASIC CONCEPTS
This section briefly describes some basic .NET and CLR Hosting concepts that are referenced throughout this paper. In the .NET framework, many different high-level programming languages can be used to construct programs. The compilation of the program produces a file, called an assembly, containing the compiled code in the Microsoft Intermediate Language (MSIL), and a manifest containing all references to dependent assemblies. The manifest is an integral part of every assembly that renders the assembly self-describing. The assembly manifest contains the assembly's metadata, which describes all structures, fields, properties, classes, inheritance relationships, functions, and methods defined in the assembly. The manifest additionally establishes the assembly identity, specifies the files that make up the assembly implementation, itemizes the compile-time dependencies on other assemblies, and specifies the set of permissions required for the assembly to run properly. This information is used at run time to resolve references, enforce
version binding policy, and validate the integrity of loaded assemblies.

The .NET Framework supports a mechanism called custom attributes for annotating classes, properties, functions and methods with additional information or facets the application may want to capture in metadata. All .NET compilers consume these annotations without interpretation and store them in the assembly. These annotations can be examined using a common set of reflection APIs.

Managed code is MSIL executed in the CLR rather than directly by the operating system. Managed code applications gain CLR services such as automatic garbage collection, runtime type checking, and security support. These services help provide uniform platform- and language-independent behavior of managed-code applications. At execution time, a just-in-time (JIT) compiler translates the MSIL into native code (e.g., Intel X86 code). During this translation, code must pass a verification process that examines the MSIL and metadata to find out whether the code can be determined to be type safe.

3. FUNCTIONS

CLR user-defined functions (UDFs) provide a mechanism for extending SQL Server with new scalar and table-valued routines written in .NET languages.

3.1 SCALAR FUNCTIONS

A static method can be registered as a scalar CLR UDF. For example, a new function for a mathematical or statistical operation not natively supported by SQL Server can be written as a static function in C# and registered as a CLR UDF. CLR UDFs also allow SQL Server to leverage the functionality available in the .NET framework. For example, the libraries for compression (System.IO.Compression), regular expressions (System.Text.RegularExpressions), and cryptography (System.Security.Cryptography) can all be used inside CLR UDFs.

CLR UDFs have many advantages over those written in Transact-SQL (T-SQL) (the natively-supported SQL Server language containing SQL plus procedural extensions):

- While T-SQL UDFs are restricted to the T-SQL language, CLR UDFs can be written in any CLR-supported language. The procedural constructs of the T-SQL language are not as rich as those of the CLR. For example, T-SQL limits recursion to a nest level of 32 function calls and has limited library support compared to the .NET framework.
- T-SQL is an interpreted language and the CLR languages are compiled to MSIL and JITed to machine instructions. Thus a CLR UDF outperforms an equivalent T-SQL UDF, especially for computationally complex functions.
- SQL Server supports parallel query processing (executing a single query across multiple threads). T-SQL UDFs prevent a query from executing in parallel whereas CLR UDFs are parallelizable. This allows queries using CLR UDFs to achieve better performance on multiprocessor systems.

3.2 UDF Contract

Scalar functions have a very simple extensibility contract. A mapping from T-SQL types to CLR types for the UDF's parameters must be provided as part of this registration. The static method can be annotated with the Microsoft.SqlServer.Server.SqlFunction custom attribute. This attribute defines some properties of the function which SQL Server uses to optimize CLR UDF invocation. These properties include:

- **DataAccess/SystemDataAccess**: If the value of either of these properties is Read, it indicates that the CLR function reads user or system tables inside SQL Server. If the value is None, it indicates the function doesn't do any data access. This property determines whether the function can be parallelized and enables optimizations around the invocation of the function.
- **IsPrecise**: A Boolean property that indicates whether or not the UDF involves imprecise calculations such as floating point operations. Imprecise functions cannot be indexed.
- **IsDeterministic**: A Boolean property that indicates whether the function produces the same output given the same input values, state of the local database, and execution context. This value affects optimizations in expression execution and indexability.

A SQL query that uses a UDF often invokes the UDF on a per-row basis during a table scan. This makes UDF invocation performance critical to achieving good query performance. To get
the desired performance, SQL Server compiles a Microsoft Intermediate language (IL) stub. This stub handles marshalling the UDFs parameters from SQL types to CLR types, invoking the function and retrieving the results. The stub also maintains T-SQLs by-value parameter passing semantics. The CLR just in time (JIT) compiler turns this stub into a function pointer which is the runtime representation of the UDF. This allows for efficient invocation at execution time without the need to look up any metadata related to the UDF.

3.3 TABLE-VALUED FUNCTIONS (TVFs)

A powerful extensibility feature initially enabled by the CLR integration in SQL Server 2005 is the ability to write table-valued functions (TVFs). CLR TVFs provide a mechanism for converting raw data (e.g., in a text/binary file or on the network) into a relation. To accomplish this conversion, CLR TVFs use a pull model to stream the data into a relation. SQL Server requests (pulls) and processes one row of data from the CLR TVF at a time as opposed to the TVF materializing the entire relation and then the entire relation being processed. This allows SQL Server to begin consuming results immediately after the first row is produced instead of having to wait for the entire return table to be populated. In our design, the CLR TVF itself is merely a class implementing the IEnumerable interface.

CLR TVFs have the same advantages as CLR UDFs. They also have the following unique advantages compared to T-SQL TVFs:

- T-SQL TVFs use a push model which materializes the entire relation before it can be processed by SQL Server. The pull model used by CLR TVFs give them a lower memory footprint and significantly better performance for returning smaller sized relations [2].

- CLR TVFs provide a customizable and flexible interface for data import operations. T-SQL is restricted to the APIs provided by bulk import or to a series of INSERT statements.

3.4 TVF Contract

The contract for CLR TVFs is best described through an example. Consider a TVF that takes as input a binary object (BLOB) in a known format and decomposes this parameter to produce a relation. Such a TVF is useful for dealing with source data that is natively in a compressed or encrypted form, for example, after traveling across a network. The contract for a CLR TVF consists of three parts: the TVF initialization method, an IEnumerable implementation, and a TVF fill row method. TVF execution begins with the initialization method, followed by one call to the IEnumerator per row to retrieve a row’s data and one call per row to the fill row method to break each row into column values.

The TVF initialization method has the same contract as a CLR UDF except that it must return an IEnumerable object. This method also specifies the name of the fill row method for the TVF in the Microsoft.SqlServer.Server.SqlFunction attribute. The IEnumerable returned by the TVF Init() method is what SQL Server uses to pull rows into the database. The LOB decomposition TVF’s initialization method shown below takes a SqlBytes parameter as input and returns a BlobCrackingIEnumerable. The BlobCrackingIEnumerable simply returns a specialized IEnumerable, in this example a BlobCrackingIEnumerable.

```csharp
public static IEnumerable LobCrackingTVFInit(SqlBytes blob)
{
    return new BlobCrackingIEnumerable(blob);
}
```

The IEnumerable decomposes the SqlBytes parameter into rows. The interface contains three methods:

- **Current:** This property returns an object representing the current row the enumerator is situated on. This object will be decoded by the fill row method to produce the values for each column in the row.

- **MoveNext():** This method moves the enumerator to the next row. If there are no more rows remaining it returns false. In our example MoveNext() parses the next row of data from the SqlBytes LOB parameter and stores it in the m_CurrRow member. This is the member Current returns. We assume each LOB consists of 2 rows each with one integer column and one double column

- **Reset():** This method sets the IEnumerable back to the initial state.

```csharp
private class BlobCrackingIEnumerator : IEnumerable
{
    // Binary data we want to crack
    private SqlBytes m_blob;
    // Object representing current row
    private CCrackedRow m_CurrRow;
    // offset in the m_blob we have read up to
    private int m_pos;
    // number of rows we have read
    private int m_cRows;

    public BlobCrackingIEnumerator(SqlBytes blob)
    {
        m_blob = blob;
        Reset();
    }

    public object Current {
        get { return m_CurrRow; }
    }

    public bool MoveNext()
    {
        if (m_cRows == 3)
        { // No more rows left
            return false;
        }
        else
        {
            m_CurrRow.IntColumn = ReadInt(m_pos,m_blob);
            m_pos += sizeof(int);
            m_CurrRow.DblColumn = ReadDbl(m_pos,m_blob);
            m_pos += sizeof(double);
            m_cRows++;
            return true;
        }
    }

    public void Reset()
    {
        m_pos = 0;
    }
}
```
The fill row method takes the object returned by the IEnumerator Current call and decodes it to produce individual column values. In our example it takes a CCrackedRow object and decodes it into an integer column and a double column.

```csharp
public static void LobCrackingTVFreadRow(object row, out int IntColumn, out double DoubleColumn)
{
    CCrackedRow crackedRow = (CCrackedRow)row;
    IntColumn = crackedRow.IntColumn;
    DoubleColumn = crackedRow.DoubleColumn;
}
```

The CLR TVF Init() method is registered with SQL Server just like a CLR UDF. The TVF can be used in a SQL query anywhere a table expression is allowed.

### 3.5 Order-aware TVFs

CLR TVFs have no notion of indexing; they are streamed one row at a time. For large return tables this can limit the efficiency of the query processor. This can be mitigated by using an order hint in SQL Server 2008. The order hint tells the query optimizer that the CLR TVF's rows conform to a predefined sort order so sorts can be avoided. The following query below uses the Initmethod() TVF to return EventLog entries as a row set.

```sql
SELECT TOP 10 *
FROM dbo.Initmethod ('System')
ORDER BY timeWritten
```

Examining the execution plan for this query, we notice that the optimizer introduces a sort operator before returning the results so it can guarantee the top events given the particular requested order (see Figure 2).

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Ordered TVF (ms)</th>
<th>Unordered TVF (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT query with TOP 1000 and ORDER BY matching TVF ORDER</td>
<td>14</td>
<td>3771</td>
</tr>
<tr>
<td>SELECT query with ORDER BY matching TVF ORDER returning all rows</td>
<td>8649</td>
<td>12486</td>
</tr>
<tr>
<td>INSERT into a user table with an index matching TVF ORDER</td>
<td>7013</td>
<td>9129</td>
</tr>
<tr>
<td>Join with a 1000 row table with an index matching TVF ORDER</td>
<td>98</td>
<td>3301</td>
</tr>
</tbody>
</table>

### 4. USER-DEFINED TYPES

SQL Server 2005 introduced user-defined types (UDTs) as a mechanism to extend the scalar type system of SQL. UDTs are not a general object-relational (O-R) mapping mechanism. That is, UDTs should not be used to model complex business objects such as employee, contact, customer, or product. For more general O-R mapping mechanisms, consider the LINQ to SQL or LINQ to
Entities frameworks available as part of the ADO.NET framework in Visual Studio 2008 [1,4,7,10,11]. SQL Server UDTs, implemented as CLR classes or structs, are designed to model atomic scalar types such as monetary currency, spatial types (see Section 7.2), specialized date or time, etc. UDTs allow users to define a type using the .NET framework, and to deploy and use these types within the database. Once deployed, UDTs can be used practically everywhere a native SQL Server scalar type can be used. In SQL Server 2005, UDTs were limited to 8000 bytes. SQL Server 2008 lifts this limit, increasing it to 2GB, mirroring the size limits for other large object types in the server.

4.1 Serialization
To persistently store UDT instances, it must be possible to transform the state of the instance to and from its internal binary format. This is achieved through a pair of routines for serialization and deserialization. The serialization routine transforms the runtime state of a CLR class or struct instance to a binary stream. The deserialization routine performs the inverse operation. SQL Server supports two forms of serialization: native (Format.Native) and user-defined (Format.UserDefined).

Native serialization requires that the only members of the class in question are basic types such as integers, floating-point numbers, and characters; even types such as arrays and strings are prohibited in native serialization. While restrictive, this serialization method does all of the serialization work for the user, and additionally guarantees that the instance is byte-ordered (see below) according to the order in which the members of the class are defined.

User-defined serialization allows users to define their own serialization format by implementing the .NET IBinarySerialize interface. While this serialization places more responsibility on the UDT author, it also removes all restrictions on the structure of the class; the user needs to only provide the code to serialize them. This is currently the only option for users wanting to serialize more than 8000 bytes of data.

4.2 Accessing Members
Members of a UDT can be accessed using the usual dot notation, e.g. instance.method(). When a method is invoked, the instance is deserialized into an actual instance of the class in the CLR, and CLR code is executed to invoke the method. Member properties and variables are accessed similarly.

4.3 Sorting and Indexing
In order to sort instances of a type, as well as to be able to build an index on one, the system must be able to sort them. Safely supporting a user-defined sort poses the risk of problems such as infinite loops during sorting. In order to avoid these problems, SQL Server only allows sorting based on the binary value of the instance.

To enable sorting and indexing of a UDT, the UDT must be declared as being byte-ordered, which declares that the sort order for instances of the type matches the dictionary-order of the bytes in the serialization of the instances. Even for very simple instances this is not trivially true. E.g., for normal two’s-compliment integer values, negative numbers have a leading 1 in their binary value, whereas positive numbers have a leading 0; these values must be rectified if their logical sort order is to match the binary value.

Native serialization handles this binary ordering for all types it allows.

4.4 Conversion
UDTs can be easily converted to and from character, binary, and XML types within the server.

The serialization methods, whether native or user-defined, provide for conversion between the UDT and a binary value. In addition to using this to persist instances, SQL Server uses these methods to provide conversion to and from binary types within the server.

The contract for UDTs also requires that the user provide a Parse() method, which takes a string and returns an instance of the UDT, and a ToString() method which performs the reverse operation. These types are used by the server to provide conversion to and from character types in the server.

Finally, SQL Server makes use of the XML conversion built into the .NET framework to provide conversion to and from XML instances within the server. These can be customized by implementing the .NET IXmlSerializable interface.

4.5 Nullability
In order to support null values within the database, the server requires that the type implement two additional properties: IsNull and Null. IsNull is a Boolean value that should be true only for null instances of the type. Null is a property providing a null instance of the type.

4.6 Contract Overview
In order to conform to SQL Server’s scalar type semantics, UDTs need to follow a contract, which SQL Server uses to implement common operations, such as serialization, nullability, text and XML conversions, comparisons and more.

The first part of the contract is the SqlUserDefinedTypeAttribute, which must be specified on the type. The primary purpose of the attribute is to define the serialization behavior (native or user-defined, byte ordered or not). This attribute allows specification of an optional validation method that SQL Server can use to test values coming from outside the server for correctness, enforcing data integrity at its perimeter.

The second part of the contract is a set of required and optional interfaces. System.Data.SqlTypes.INullable is required because SQL Server handles NULL values differently than managed languages and must be able to achieve its semantics. System.Data.IBinarySerialize is required if user-defined serialization is used. Finally, if the user wants to customize the XML serialization, System.Xml.Serialization.IXmlSerializable can be implemented.

The final part of the contract is a set of required methods. static void Parse (string) is used by the server to convert, either implicitly or explicitly, string values to UDT instances. string ToString() is used to do the opposite. A static Null() method also needs to be provided in order for the server to retrieve a null instance for this particular UDT.

4.7 Limitations
The database environment imposes significant challenges to an extensible type system. SQL Server places a number of limitations on UDTs in order to make effective use of them, including the size, serialization, sorting, and indexing restrictions discussed above. In addition:

- Inheritance is not supported. Inheritance poses a number of hard challenges to a database system, particularly when types are modified or removed, as it is difficult to track the actual types in use in the database. In order to avoid these...
problems, SQL Server disallows inheritance in the definition of UDTs.

- Serialization is all-or-nothing. Although for many applications, being able to partially deserialize an instance would be desirable, this does not allow a complete CLR class to be generated.
- Overloaded methods cannot be called.
- Static data members must be read-only.
- Native serialization cannot exceed 8000 bytes.

4.8 Use with Managed Clients
Since CLR UDTs are simply CLR classes with some additional restrictions, they can be instantiated on a managed client as well as in the server. When using the ADO.NET SqlClient to access the database, clients can make use of this by mapping the type directly to a matching instance of the type on the client. Not only can users have the same behavior on the client as in the database, the same code is running in both places.

4.9 Example
The following example shows a code fragment describing the definition of a compressed string type as a large UDT. To define a UDT as large, one only needs to specify a value of -1 for the MaxByteSize property of the SqlUserDefinedType custom attribute.

```csharp
//**************************************************************
// CompressedString UDT
//**************************************************************
[Serializable]
[SqlUserDefinedType(Format.UserDefined,
    IsByteOrdered = false,
    MaxByteSize = -1, IsFixedLength = false)]
public struct CompressedString : INullable,
    IBinarySerialize
{
    //**************************************************************
    // Private Fields
    //**************************************************************
    private bool m_fNotNull;
    private SqlString m_value;
    //**************************************************************
    // Constructors
    //**************************************************************
    private CompressedString(bool fNull)
    {
        m_fNotNull = false;
        m_value = SqlString.Null;
    }
    public CompressedString(SqlString value)
    {
        m_value = value;
        m_fNotNull = true;
    }
    //**************************************************************
    // INullable interface
    //**************************************************************
    public bool IsNull
    {
        get { return !m_fNotNull; }
    }
    public static CompressedString Null
    {
        get { return new CompressedString(false); }
    }
    //**************************************************************
    // IBinarySerialize Methods
    //**************************************************************
    public void Read(BinaryReader r)
    {
        m_fNotNull = r.ReadBoolean();
        m_value = Decompress(r.BaseStream);
    }
    public void Write(BinaryWriter w)
    {
        w.Write(m_fNotNull);
        w.Write(Compress(m_value.ToString()));
    }
    //**************************************************************
    // UDT Methods
    //**************************************************************
    [return: SqlFacet(MaxSize = -1)]
    public override string ToString()
    {
        if (IsNull)
            return "Null";
        else
            return Value.ToString();
    }
    public static CompressedString Parse(
        SqlString s)
    {
        if (s.IsNull)
            return CompressedString.Null;
        return new CompressedString(s);
    }
    //**************************************************************
    // Compression Methods
    //**************************************************************
    public static byte[] Compress(string input)
    {
        byte[] indata = (new System.Text.ASCIIEncoding()).
            GetBytes(input);
        MemoryStream ms = new MemoryStream();
        DeflateStream CompressedStringzipStream =
            New DeflateStream(ms, CompressionMode.
            Compress, true);
        CompressedStringzipStream.Write(indata, 0,
            indata.Length);
        CompressedStringzipStream.Close();
        return ms.GetBuffer();
    }
    public static string Decompress(Stream input)
    {
        DeflateStream ds =
            new DeflateStream(input, CompressionMode.
            Decompress, true);
        return (new StreamReader(ds)).ReadToEnd();
    }
    [SqlFacet(MaxSize = -1)]
    public SqlString Value
    {
        get {
            if (m_fNotNull)
                return m_value;
            else
                throw new SqlNullValueException();
        }
    }
```
4.10 System CLR Types in SQL Server 2008

SQL Server 2008 introduces three new data types, hierarchyid, geography and geometry, as a new class of database object: the system CLR type. These system CLR types are almost identical to UDTs; they are implemented using the UDT framework with some small additional enhancements.

The UDT architecture was chosen for these types for several reasons. First, these types express new concepts in SQL Server and do not have legacy behavior requirements, which makes interoperability with many built-in intrinsic functions unnecessary. Reusing the UDT platform allowed higher development and testing focus on the rich functionality the types offer, rather than on exposing new types throughout the product. Finally, since there are currently no existing managed types that are equivalent to these types, we can improve our programmability by allowing users to use these types both on the server and on the client.

Although most of the architecture is identical to that of UDTs, there are some notable differences:

- While UDTs must be deployed by the user, system CLR types are shipped with the server and are available without any action by the user.
- Although a copy of each UDT must be deployed to each database on the server in which they are to be used, system CLR types are present only once on the server, but available in every database.
- User-written assemblies are stored in the database in which they are registered, while assemblies containing system CLR types are stored directly on the file system.

5. USER-DEFINED AGGREGATES

It is not possible to anticipate and ship in the database every aggregate function a user may require. CLR user-defined aggregates (UDAs) give users the ability to write their own aggregates to suit their own specific needs. Some common applications include string processing, and statistical or mathematical computations. CLR UDAs are tightly integrated into the system, in the same manner as native aggregates, and can be parallelized by the system just like native aggregates.

In SQL Server 2005, CLR UDAs cannot be larger than 8000 bytes. SQL Server 2008 removes this restriction, allowing them to be as large as 2 GB. Large UDAs are implemented in the same fashion as large UDTs (described in Section 4). Additionally, in SQL Server 2008, CLR UDAs are allowed to have multiple inputs.

5.1 UDA Contract

An example aggregate for computing population covariance (a measure of how two sets of data vary together) is shown below to highlight the different parts of the UDA contract. The contract consists of four methods: Init(), Accumulate(), Merge() and Terminate(). The aggregation operator makes a call to Init() to initialize the UDA. This is followed by a call to Accumulate() for each row in the input. Finally, a call to Terminate() produces the result of the aggregation. Merge() is used to merge two partial aggregations together during parallel query processing. The implementations of these methods for the population covariance UDA are given below. The SqlUserDefinedAggregate attribute defines some properties of the UDA. The only property pertinent to this example is Format. This property is the same as the one used for UDTs, and specifies how the UDA should be serialized.

The Init() method is called once at the start of each group in the query. For covar_pop, this initializes the aggregates members to zero. The Accumulate() method is called once per row. The covar_pop UDA works over two sets of integers and thus the accumulate method has two integer inputs. The Accumulate() method calculates four separate sums which are needed to produce the final result at the end of the group. The Merge() method is used to merge together two partial aggregations for the same group. Aggregates need this method so they can be used in parallel queries. For example, if the aggregation for a group is executed across two threads, then half of the group's aggregation can be calculated by a UDA instance on the first thread and half by another UDA instance on the second. Merge() is then called to combine the aggregates and produce the aggregate for the entire group. For covar_pop, Merge() merely adds the fields of the two aggregates together.

The Terminate() method is called at the end of the group (after all Accumulate() and Merge() calls have completed) and produces the result of the aggregation. For covar_pop, this involves calculating an expression involving the four sums gathered during accumulation.
into SQL Server, and are optimized for the single aggregation they carry out. Built-in aggregates provide a good baseline for the best possible performance.

The table below summarizes the execution times of simple select queries over a table with 500,000 rows using two different aggregates. For simple aggregates such a sum, the UDA is noticeably slower, but for more complicated aggregates such as standard deviation (STD DEV) the difference between the two is less pronounced. The last row in the table also demonstrates that UDAs can be efficiently parallelized. For this scenario, the UDA for standard deviation is executed across 8 threads, and shows a significant performance gain over the single thread execution of the same UDA in the row above.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>CLR UDA (ms)</th>
<th>Native Agg (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial query SUM</td>
<td>270</td>
<td>95</td>
</tr>
<tr>
<td>Serial query STD DEV</td>
<td>936</td>
<td>607</td>
</tr>
<tr>
<td>Parallel query STD DEV</td>
<td>140</td>
<td>77</td>
</tr>
</tbody>
</table>

5.3 Large UDAs

One common request is an aggregate method for string concatenation. This has many uses, e.g. to assemble a series of email addresses separated by a semicolon, and is an excellent candidate for a UDA. The Accumulate() function for such an aggregate might look something like:

```csharp
public void Accumulate (SqlString Value)
{
    if (!Value.IsNull)
        result+= Value.Value + ";";
}
```

In SQL Server 2005, however, this aggregate quickly runs into the 8000-byte UDA size limitation. SQL Server 2008 lifts this restriction: by setting the max byte size to -1, the internal state can grow as large as 2GB. This can then be registered in SQL Server with the return value specified as `nvarchar(max)`:

```sql
CREATE AGGREGATE dbo.concat(@Value nvarchar(4000))
RETURNS nvarchar(max)
EXTERNAL NAME [concatProject].[concat];
```

This aggregate can be used in a query as any other aggregate function would be used:

```sql
SELECT dbo.concat(email)
FROM dbo.attendees
WHERE ...
```

The user might additionally need the format of each concatenated value to depend on flags included in a separate column of the attendees table. In SQL Server 2008, it is now possible to create aggregates that take multiple columns as input as in the following query:

```sql
SELECT dbo.concat (email, format_flags)
```

This capability makes it simple to write powerful aggregates such as weighted averages, or other statistical functions such as median.

6. DATA ACCESS in ROUTINES

Stored procedures and table-valued functions are examples of routines that encapsulate business logic containing a mixture of computation and data access. Traditionally, data access from .NET routines is done via the ADO.NET SqlClient interfaces. Microsoft has recently introduced new Language Integrated Query (LINQ) technologies that extend .NET languages natively with query expressions to further reduce, and for some scenarios completely eliminate, the impedance mismatch for applications. Consider a table containing employee information as follows:

```sql
CREATE TABLE Employees (  
    EmpId int primary key,  
    Name varchar(100),  
    Email varchar(100),  
    Phone varchar(10),  
    Title varchar(20),  
    HireDate date);
```

A table function to obtain Employees hired prior to some date would be written using ADO.NET as follows:

```sql
CREATE TABLE Employees (  
    EmpId int primary key,  
    Name varchar(100),  
    Email varchar(100),  
    Phone varchar(10),  
    Title varchar(20),  
    HireDate date);
```

```csharp
public void Accumulate (SqlString Value)
{
    if (!Value.IsNull)
        result+= Value.Value + ";";
}
```

In SQL Server 2005, however, this aggregate quickly runs into the 8000-byte UDA size limitation. SQL Server 2008 lifts this restriction: by setting the max byte size to -1, the internal state can grow as large as 2GB. This can then be registered in SQL Server with the return value specified as `nvarchar(max)`:

```sql
CREATE AGGREGATE dbo.concat(@Value nvarchar(4000))
RETURNS nvarchar(max)
EXTERNAL NAME [concatProject].[concat];
```

This aggregate can be used in a query as any other aggregate function would be used:

```sql
SELECT dbo.concat(email)
FROM dbo.attendees
WHERE ...
```

The same function using LINQ will be written as:

```csharp
public static void FillEmpRdr (Object o,  
    out datet ime hiredate, out string name){  
    name = (datetime)((IDataRecord)o)["HireDate"];  
    city = (string)((IDataRecord)o)["Name"];  
}
```

The same function using LINQ will be written as:

```csharp
public static void FillEmpRdr (Object o,  
    out datet ime hiredate, out string name){  
    name = (dateTime)((IDataRecord)o)["HireDate"];  
    city = (string)((IDataRecord)o)["Name"];  
}
```

The function using LINQ will be written as:

```csharp
public static void FillEmpRdr (Object o,  
    out datet ime hiredate, out string name){  
    name = (dateTime)((IDataRecord)o)["HireDate"];  
    city = (string)((IDataRecord)o)["Name"];  
}
```

The same function using LINQ will be written as:

```csharp
public static void FillEmpRdr (Object o,  
    out datet ime hiredate, out string name){  
    name = (dateTime)((IDataRecord)o)["HireDate"];  
    city = (string)((IDataRecord)o)["Name"];  
}
```

The same function using LINQ will be written as:

```csharp
public static void FillEmpRdr (Object o,  
    out datet ime hiredate, out string name){  
    name = (dateTime)((IDataRecord)o)["HireDate"];  
    city = (string)((IDataRecord)o)["Name"];  
}
```
select e;
return emps;
}
}

Clear advantages of LINQ include compile-time type checking of query expressions, and the elimination of the `FillEmpRdr()` routine by using generic collection types `IEnumerable<Employee>`.

The amount of information that needs to be specified inside the `SqlFunction` custom attribute is also reduced.

7. EXTENSIBILITY EXAMPLES

7.1 Hierarchical Identifiers
SQL Server 2008 introduces the `hierarchyid` built-in type, which is designed to make it easier to store and query hierarchical data. Hierarchical data is defined as a set of data items linked to one another by a set of parent-child relations. Common examples include organizational structures, hierarchical file systems, tasks in a project, taxonomies of language terms, single-inheritance type hierarchies, and part-subpart relationships.

Consider the example organizational hierarchy in Figure 4. The number sequence indicated at the upper right corner of every box represents a hierarchical identifier for the corresponding department.

Figure 4 - An Organizational Hierarchy.

The `hierarchyid` type is constructed as a system CLR type and exposes methods to help maintain these identifiers. The table below lists some of the key methods exposed by this type.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static SqlHierarchyId GetRoot()</td>
<td>Returns the root of the hierarchy type</td>
</tr>
<tr>
<td>SqlHierarchyId GetDescendant( SqlHierarchyId child1, SqlHierarchyId child2 )</td>
<td>Returns a child Id x such that child1 &lt; x and child2 &gt; x.</td>
</tr>
<tr>
<td>SqlBoolean IsDescendant(SqlHierarchyId child)</td>
<td>Returns true if child is &gt;= this.</td>
</tr>
<tr>
<td>SqlInt16 GetLevel()</td>
<td>Returns an integer representing the depth of the node this in the tree. Returns 0 if this is the root.</td>
</tr>
<tr>
<td>SqlHierarchyId GetAncestor( int n )</td>
<td>Returns a hierarchical id representing the nth ancestor of this.</td>
</tr>
</tbody>
</table>

Using this type, one could model this organization using the following table and index:

```sql
CREATE TABLE organization ( node hierarchyid primary key clustered,
                          level as node.GetLevel() persisted,
                          empid int unique,
                          name nvarchar(100)
                          )
CREATE UNIQUE INDEX org_idx ON organization (level, node)
```

We can now insert the top-level CEO record:

```sql
INSERT organization (node, empid, name) VALUES (hierarchyid::GetRoot(), 123, 'Frank Smith')
```

To add a record for an employee, we use the `GetDescendant()` method to obtain a `HierarchyId` value under the proper manager:

```sql
CREATE PROC AddEmp (@mgrid int, @empid int, @name nvarchar(100) ) AS
BEGIN
DECLARE @mnode hierarchyid, @lc hierarchyid
SELECT @mnode = node
FROM organization
WHERE empid = @mgrid
BEGIN TRANSACTION
SELECT @lc = max(node)
FROM organization
WHERE @mnode = node.GetAncestor(1)
INSERT organization (node, empid, name)
VALUES (@mnode.GetDescendant(@lc, NULL), @empid, @name)
COMMIT
END
```

We expect this new data type will enable the efficient implementation of applications where data is organized hierarchically.

7.2 Spatial Framework
Even though location is a concept prevalent in everything we do, and has long been used in specialized computer systems, it is only in recent years that location-awareness has become an integral part of everyday software applications. With the proliferation of mapping frameworks, GPS and other location aware devices it is increasingly common to encounter spatial data, which often needs to be stored, queried, and reasoned upon. SQL Server 2008 leverages the CLR integration to provide native spatial data support.

Many business applications have some form of location data: sales regions, delivery routes, factory or point of sale locations, or employee addresses. Not only is this information often required to be stored in the database, it is also common to want to run queries that make use of the spatial semantics: What is the average distance that customers have to the nearest point of sale location? How do sales for a region compare to those from adjacent ones?

SQL Server 2008 supports storing and querying of geospatial data, that is, location data referenced to the Earth. This is commonly modeled either using planar or geodetic coordinate systems, the main distinction being that the latter takes into account the curvature
of the earth. SQL Server 2008 introduces two new data types, `geometry` and `geography` which correspond to the planar and geodetic models.

These data types are implemented as system CLR types, making it easy to create columns of either of these types on a table:

```sql
CREATE TABLE points_of_sale
(id int, name nvarchar(50), location geography);
```

The Open Geospatial Consortium (OGC) defines a canonical textual representation for a geometry known as the Well-Known-Text (WKT). We can use the WKT to insert data into the `points_of_sale` table by using a static method that converts from WKT to an actual instance:

```sql
INSERT INTO points_of_sale (id, name, location)
VALUES (1000, N'Main Store',
geometry::STPointFromText('POINT (-122.7 44.5)', 4326));
```

We can later write queries against the table created above to answer business questions. For example, assume that we have a `geography` variable that holds a polygon representing a given urban area. We can then write a query to retrieve from the database all points of sale that fall within the given urban area.

```sql
SELECT id, name FROM points_of_sale
WHERE location.STIntersects (@Urban_area) = 1;
```

SQL Server 2008 also provides new spatial indexing capabilities to speed up processing of queries involving spatial operations. As an example the following statement would create an index on the table above:

```sql
CREATE SPATIAL INDEX SIndex_points_of_sale
ON points_of_sale(location);
```

The `geography` data type works with latitude/longitude data on an ellipsoidal model of the Earth. This is the model most non-expert users are accustomed to, but is also critical for operations performed over large areas.

The `geometry` type works on a flat plane, and is appropriate for local mapping, e.g., tracking assets within a warehouse. Additionally, GIS experts commonly deal with geospatial data that has been projected to a plane either for legacy or legislative reasons.

The Open Geospatial Consortium (OGC) defines a canonical textual representation for a geometry known as the Well-Known-Text (WKT). We can use the WKT to insert data into the `points_of_sale` table by using a static method that converts from WKT to an actual instance:

```sql
INSERT INTO points_of_sale (id, name, location)
VALUES (1000, N'Main Store',
geometry::STPointFromText('POINT (-122.7 44.5)', 4326));
```

We can later write queries against the table created above to answer business questions. For example, assume that we have a `geography` variable that holds a polygon representing a given urban area. We can then write a query to retrieve from the database all points of sale that fall within the given urban area.

```sql
SELECT id, name FROM points_of_sale
WHERE location.STIntersects (@Urban_area) = 1;
```

SQL Server 2008 also provides new spatial indexing capabilities to speed up processing of queries involving spatial operations. As an example the following statement would create an index on the table above:

```sql
CREATE SPATIAL INDEX SIndex_points_of_sale
ON points_of_sale(location);
```

The `geography` data type works with latitude/longitude data on an ellipsoidal model of the Earth. This is the model most non-expert users are accustomed to, but is also critical for operations performed over large areas.

The `geometry` type works on a flat plane, and is appropriate for local mapping, e.g., tracking assets within a warehouse. Additionally, GIS experts commonly deal with geospatial data that has been projected to a plane either for legacy or legislative reasons.

8. DEVELOPMENT EXPERIENCE

Besides powerful extensibility capabilities offered by user-defined types, user-defined aggregates and streaming table-valued functions, the biggest advantage of CLR integration is the ease in which database developers can develop server-side applications. The ease of development comes from following various aspects involved in application development:

1. Programming model and surface
2. .NET Framework libraries
3. Development Environment and tools

8.1 Programming Model and Surface

CLR integration enables developers to use one of the several modern programming languages, such as Microsoft Visual C# and Microsoft Visual Basic .NET. These languages provide significantly better procedural programming flexibility than does T-SQL. Additionally, developers can make use of the powerful .NET framework in their code.

8.2 .NET Framework Libraries

There is extensive functionality available in the .NET framework libraries that developers can leverage in their applications, including regular expression matching and XSLT transformations. Reusing these from the .NET framework not only saves development time, but also allows the use of implementations that are optimized, secure, and reliable.

8.3 Development Environment and Tools

Another major advantage of using CLR integration is streamlined development though powerful development environments and tools available for .NET. Users can use the same set of tools for developing and debugging CLR database objects as they use for middle- or client-tier .NET applications. For example, development, deployment and debugging of CLR integration objects are fully supported in Microsoft Visual Studio. Visual Studio 2005 and 2008 have a new SQL Server project that has code templates, automatic deployment to SQL Server, debugging capabilities and ability to add references to other assemblies in the database. The experience is very similar to any other project type, enabling users to leverage most of their knowledge and expertise.

The SQL Server project supports a single-click build and deploy operation that compiles the code into an assembly, uploads the assembly into the SQL Server database associated with the project, and automatically creates the routines, types, and aggregates defined in the code in the database based on the custom attributes (SqlProcedure, SqlFunction, SqlTrigger, etc.) used. This operation also uploads the source code and debugging symbols (the .pdb file) associated with the assembly for providing the ability to view/debug in the future.

9. RELATED WORK

The integration of the CLR inside SQL Server provides a powerful mechanism to database programmers to extend the programming capabilities in a safe, easy, and efficient way. Adding extensibility mechanisms to a relational DBM is not new and significant work has happened in this area in the past.

The exploration of this area started more than two decades ago when Stonebraker et al. [9] presented their work on Abstract Data Types (ADT) and its integration in the INGRES database system. Conceptually, an ADT is similar to SQL Server’s user-defined type. A user needs to: (1) define the ADT and its operators and functions, including certain functions that might be required for conversion operations; (2) register the ADT and its operators with the database manager; and (3) code and compile the functions and operators associated with the ADT in C. Once the above is done, users can declare columns to store instances of the ADT. A disadvantage of using C as the language for extensibility is that the extensions are less safe than those written a managed programming environment like .NET.

Oracle introduced the ability to write stored procedures in Java in its 8i release [6]. Alternatively, extensions can also be written in PL/SQL for database developers and allows them to use the rich programming environment and libraries in their server-side application. Development experience for Java stored procedures in Oracle is similar to CLR integration where a user needs to: (1) develop the classes in Java and compile them; (2) upload the java
source, classes and resource files in the database using the loadjava utility; and (3) register the methods in Java classes as stored procedures or functions in PL/SQL. Oracle’s extensibility approach includes contracts for adding user-defined indexes as well as providing cardinality and cost estimates to the query optimizer. The Java VM is not as deeply integrated to the Oracle runtime for the management of resources (memory, threads, synchronization).

Other extensibility mechanisms provided by database systems include: (1) Extended Stored Procedures in Microsoft SQL Server; (2) .NET CLR integration in Oracle 10g Release 2; and (3) .NET CLR integration in IBM DB2 Stinger.

CLR integration has several advantages over the above extensibility mechanisms offered by other products:

- The code runs inside the SQL Server process space using CLR hosting mechanisms.
- SQL Server has a very tight control over the execution of code inside the CLR. This is achieved by using hosting APIs exposed by the CLR. SQL Server uses this to control resource consumption like memory, CPU etc. as well as control the behavior during critical exceptions like out-of-memory or stack overflow.
- The code (assembly) resides inside the database and, as any other database object registered to SQL, all users enjoy the benefits of all the database utilities such as backup/restore, business intelligence, etc.

10. CONCLUSIONS AND OUTLOOK

This paper presented key aspects of the programmability and extensibility features enabled by the integration of the CLR inside the SQL Server DBMS. First, we described the extensibility contracts for scalar and table UDFs including order-aware TVFs. Second, we presented the contract for scalar UDTs in detail. Third, we presented the contract of user-defined aggregates and included preliminary performance results demonstrating the overhead of UDA's relative to native SQL aggregates. Fourth, we presented how LINQ can be used to express data access request inside routines. Fifth, we illustrated two uses of the extensibility framework to implement hierarchical identifier type and a spatial framework. Sixth, we presented the extensions to the Visual Studio development environment to enhance database developer productivity. Finally, we discussed some related work in this area.

We are continuing to investigate and address extensibility in SQL Server. Future work may include improved integration with the query optimizer for user-defined constructs, improved flexibility in our extensibility contracts, as well as the exposure of new contracts, e.g., for index extensibility.

11. ACKNOWLEDGEMENTS

We would like to thank all members of the server programmability team who helped build this product.

12. REFERENCES