**Common Language Runtime (CLR) Integration Overview**

Microsoft SQL Server now features the integration of the common language runtime (CLR) component of the .NET Framework for Microsoft Windows. The CLR supplies managed code with services such as cross-language integration, code access security, object lifetime management, and debugging and profiling support. For SQL Server users and application developers, CLR integration means that you can now write stored procedures, triggers, user-defined types, user-defined functions (scalar and table-valued), and user-defined aggregate functions using any .NET Framework language, including Microsoft Visual Basic .NET and Microsoft Visual C#. Note that Visual Studio .NET 2003 cannot be used for CLR integration programming. SQL Server includes the .NET Framework version 2.0 SP1 pre-installed, and Visual Studio .NET 2003 cannot use the .NET Framework 2.0 assemblies.

Among the major benefits of this integration are:

* **A better programming model.** The .NET Framework languages are in many respects richer than Transact-SQL, offering constructs and capabilities previously not available to SQL Server developers. Developers may also leverage the power of the .NET Framework Library, which provides an extensive set of classes that can be used to quickly and efficiently solve programming problems.
* **Improved safety and security.** Managed code runs in a common language run-time environment, hosted by the Database Engine. SQL Server leverages this to provide a safer and more secure alternative to the extended stored procedures available in earlier versions of SQL Server.
* **Ability to define data types and aggregate functions.** User defined types and user defined aggregates are two new managed database objects which expand the storage and querying capabilities of SQL Server.
* **Streamlined development through a standardized environment.** Database development is integrated into future releases of the Microsoft Visual Studio .NET development environment. Developers use the same tools for developing and debugging database objects and scripts as they use to write middle-tier or client-tier .NET Framework components and services.
* **Potential for improved performance and scalability.** In many situations, the .NET Framework language compilation and execution models deliver improved performance over Transact-SQL.

**Architecture of CLR Integration**

SQL Server integration with the .NET Framework common language runtime (CLR) enables database programmers to use languages such as Visual C#, Visual Basic .NET, and Visual C++. Functions, stored procedures, triggers, data types, and aggregates are among the kinds of business logic that programmers can write with these languages.

This section describes the architecture of CLR integration inside SQL Server and how this architecture provides a safe, scalable, secure, and efficient environment to run user code.

The following table lists the topics in this section.

[CLR Hosted Environment](http://msdn.microsoft.com/en-us/library/ms131047.aspx)

Discusses architectural design goals, mechanisms, and benefits of CLR integration.

[Performance of CLR Integration](http://msdn.microsoft.com/en-us/library/ms131075.aspx)

Covers performance considerations of the CLR integration architecture.

**CLR Hosted Environment**

The Microsoft .NET Framework common language runtime (CLR) is an environment that executes many modern programming languages, including Microsoft Visual C#, Microsoft Visual Basic, and Microsoft Visual C++. The CLR features garbage-collected memory, pre-emptive threading, metadata services (type reflection), code verifiability, and code access security. The CLR uses metadata to locate and load classes, lay out instances in memory, resolve method invocations, generate native code, enforce security, and set run-time context boundaries.

The CLR and SQL Server differ as run-time environments in the way they handle memory, threads, and synchronization. This topic describes the way in which these two run times are integrated so that all system resources are managed uniformly. This topic also covers the way in which CLR code access security (CAS) and SQL Server security are integrated to provide a reliable and secure execution environment for user code.

http://i.msdn.microsoft.com/Global/Images/clear.gif Basic Concepts of CLR Architecture

In the .NET Framework, a programmer writes in a high-level language that implements a class defining its structure (for example, the fields or properties of the class) and methods. Some of these methods can be static functions. The compilation of the program produces a file called an assembly that contains the compiled code in the Microsoft intermediate language (MSIL), and a manifest that contains all references to dependent assemblies.

|  |
| --- |
| **ms131047.note(en-us,SQL.100).gifNote:** |
| Assemblies are a vital element in the architecture of the CLR. They are the units of packaging, deployment, and versioning of application code in the .NET Framework. Using assemblies, you can deploy application code inside the database and provide a uniform way to administer, back up, and restore complete database applications. |

The assembly manifest contains metadata about the assembly, describing all of the structures, fields, properties, classes, inheritance relationships, functions, and methods defined in the program. The manifest establishes the assembly identity, specifies the files that make up the assembly implementation, specifies the types and resources that make up the assembly, 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 custom attributes for annotating classes, properties, functions, and methods with additional information the application may capture in metadata. All .NET Framework compilers consume these annotations without interpretation and store them as assembly metadata. These annotations can be examined in the same way as any other metadata.

Managed code is MSIL executed in the CLR, rather than directly by the operating system. Managed code applications acquire CLR services, such as automatic garbage collection, run-time type checking, and security support. These services help provide uniform platform- and language-independent behavior of managed code applications.

http://i.msdn.microsoft.com/Global/Images/clear.gif Design Goals of CLR Integration

When user code runs inside the CLR-hosted environment in SQL Server (called CLR integration), the following design goals apply:

##### Reliability (Safety)

User code should not be allowed to perform operations that compromise the integrity of the Database Engine process, such as popping a message box requesting a user response or exiting the process. User code should not be able to overwrite Database Engine memory buffers or internal data structures.

##### Scalability

SQL Server and the CLR have different internal models for scheduling and memory management. SQL Server supports a cooperative, non-preemptive threading model in which the threads voluntarily yield execution periodically, or when they are waiting on locks or I/O. The CLR supports a preemptive threading model. If user code running inside SQL Server can directly call the operating system threading primitives, then it does not integrate well into the SQL Server task scheduler and can degrade the scalability of the system. The CLR does not distinguish between virtual and physical memory, but SQL Server directly manages physical memory and is required to use physical memory within a configurable limit.

The different models for threading, scheduling, and memory management present an integration challenge for a relational database management system (RDBMS) that scales to support thousands of concurrent user sessions. The architecture should ensure that the scalability of the system is not compromised by user code calling application programming interfaces (APIs) for threading, memory, and synchronization primitives directly.

##### Security

User code running in the database must follow SQL Server authentication and authorization rules when accessing database objects such as tables and columns. In addition, database administrators should be able to control access to operating system resources, such as files and network access, from user code running in the database. This becomes important as managed programming languages (unlike non-managed languages such as Transact-SQL) provide APIs to access such resources. The system must provide a secure way for user code to access machine resources outside the Database Engine process. For more information, see [CLR Integration Security](http://msdn.microsoft.com/en-us/library/ms131071.aspx).

##### Performance

Managed user code running in the Database Engine should have computational performance comparable to the same code run outside the server. Database access from managed user code is not as fast as native Transact-SQL. For more information, see [Performance of CLR Integration](http://msdn.microsoft.com/en-us/library/ms131075.aspx).

http://i.msdn.microsoft.com/Global/Images/clear.gif CLR Services

The CLR provides a number of services to help achieve the design goals of CLR integration with SQL Server.

##### Type safety verification

Type-safe code is code that accesses memory structures only in well-defined ways. For example, given a valid object reference, type-safe code can access memory at fixed offsets corresponding to actual field members. However, if the code accesses memory at arbitrary offsets inside or outside the range of memory that belongs to the object, then it is not type-safe. When assemblies are loaded in the CLR, prior to the MSIL being compiled using just-in-time (JIT) compilation, the runtime performs a verification phase that examines code to determine its type-safety. Code that successfully passes this verification is called verifiably type-safe code.

##### Application domains

The CLR supports the notion of application domains as execution zones within a host process where managed code assemblies can be loaded and executed. The application domain boundary provides isolation between assemblies. The assemblies are isolated in terms of visibility of static variables and data members and the ability to call code dynamically. Application domains are also the mechanism for loading and unloading code. Code can be unloaded from memory only by unloading the application domain. For more information, see [Application Domains and CLR Integration Security](http://msdn.microsoft.com/en-us/library/ms345100.aspx).

##### Code Access Security (CAS)

The CLR security system provides a way to control what kinds of operations managed code can perform by assigning permissions to code. Code access permissions are assigned based on code identity (for example, the signature of the assembly or the origin of the code).

The CLR provides for a computer-wide policy that can be set by the computer administrator. This policy defines the permission grants for any managed code running on the machine. In addition, there is a host-level security policy that can be used by hosts such as SQL Server to specify additional restrictions on managed code.

If a managed API in the .NET Framework exposes operations on resources that are protected by a code access permission, the API will demand that permission before accessing the resource. This demand causes the CLR security system to trigger a comprehensive check of every unit of code (assembly) in the call stack. Only if the entire call chain has permission will access to the resource be granted.

Note that the ability to generate managed code dynamically, using the Reflection.Emit API, is not supported inside the CLR-hosted environment in SQL Server. Such code would not have the CAS permissions to run and would therefore fail at run time. For more information, see [CLR Integration Code Access Security](http://msdn.microsoft.com/en-us/library/ms345101.aspx).

##### Host Protection Attributes (HPAs)

The CLR provides a mechanism to annotate managed APIs that are part of the .NET Framework with certain attributes that may be of interest to a host of the CLR. Examples of such attributes include:

* SharedState, which indicates whether the API exposes the ability to create or manage shared state (for example, static class fields).
* Synchronization, which indicates whether the API exposes the ability to perform synchronization between threads.
* ExternalProcessMgmt, which indicates whether the API exposes a way to control the host process.

Given these attributes, the host can specify a list of HPAs, such as the SharedState attribute, that should be disallowed in the hosted environment. In this case, the CLR denies attempts by user code to call APIs that are annotated by the HPAs in the prohibited list. For more information, see [Host Protection Attributes and CLR Integration Programming](http://msdn.microsoft.com/en-us/library/ms403276.aspx).

http://i.msdn.microsoft.com/Global/Images/clear.gif How SQL Server and the CLR Work Together

This section discusses how SQL Server integrates the threading, scheduling, synchronization, and memory management models of SQL Server and the CLR. In particular, this section examines the integration in light of scalability, reliability, and security goals. SQL Server essentially acts as the operating system for the CLR when it is hosted inside SQL Server. The CLR calls low-level routines implemented by SQL Server for threading, scheduling, synchronization, and memory management. These are the same primitives that the rest of the SQL Server engine uses. This approach provides several scalability, reliability, and security benefits.

##### Scalability: Common threading, scheduling, and synchronization

CLR calls SQL Server APIs for creating threads, both for running user code and for its own internal use. In order to synchronize between multiple threads, the CLR calls SQL Server synchronization objects. This allows the SQL Server scheduler to schedule other tasks when a thread is waiting on a synchronization object. For example, when the CLR initiates garbage collection, all of its threads wait for garbage collection to finish. Because the CLR threads and the synchronization objects they are waiting on are known to the SQL Server scheduler, SQL Server can schedule threads that are running other database tasks not involving the CLR. This also enables SQL Server to detect deadlocks that involve locks taken by CLR synchronization objects and employ traditional techniques for deadlock removal.

Managed code runs preemptively in SQL Server. The SQL Server scheduler has the ability to detect and stop threads that have not yielded for a significant amount of time. The ability to hook CLR threads to SQL Server threads implies that the SQL Server scheduler can identify "runaway" threads in the CLR and manage their priority. Such runaway threads are suspended and put back in the queue. Threads that are repeatedly identified as runaway threads are not allowed to run for a given period of time so that other executing workers can run.

|  |
| --- |
| **ms131047.note(en-us,SQL.100).gifNote:** |
| Long-running managed code that accesses data or allocates enough memory to trigger garbage collection will yield automatically. Long-running managed code that does not access data or allocate enough managed memory to trigger garbage collection should explicitly yield by calling the System.Thread.Sleep() function of the .NET Framework. |

##### Scalability: Common memory management

The CLR calls SQL Server primitives for allocating and de-allocating its memory. Because the memory used by the CLR is accounted for in the total memory usage of the system, SQL Server can stay within its configured memory limits and ensure the CLR and SQL Server are not competing with each other for memory. SQL Server can also reject CLR memory requests when system memory is constrained, and ask CLR to reduce its memory use when other tasks need memory.

##### Reliability: Application domains and unrecoverable exceptions

When managed code in the .NET Framework APIs encounters critical exceptions, such as out-of-memory or stack overflow, it is not always possible to recover from such failures and ensure consistent and correct semantics for their implementation. These APIs raise a thread abort exception in response to these failures.

When hosted in SQL Server, such thread aborts are handled as follows: the CLR detects any shared state in the application domain in which the thread abort occurs. The CLR does this by checking for the presence of synchronization objects. If there is shared state in the application domain, then the application domain itself is unloaded. The unloading of the application domain stops database transactions that are currently running in that application domain. Because the presence of shared state can widen the impact of such critical exceptions to user sessions other than the one triggering the exception, SQL Server and the CLR have taken steps to reduce the likelihood of shared state. For more information, see the .NET Framework documentation.

##### Security: Permission sets

SQL Server allows users to specify the reliability and security requirements for code deployed into the database. When assemblies are uploaded into the database, the author of the assembly can specify one of three permission sets for that assembly: SAFE, EXTERNAL\_ACCESS and UNSAFE.

|  |  |  |  |
| --- | --- | --- | --- |
| Permission set | SAFE | EXTERNAL\_ACCESS | UNSAFE |
| Code Access Security | Execute only | Execute + access to external resources | Unrestricted |
| Programming model restrictions | Yes | Yes | No restrictions |
| Verifiability requirement | Yes | Yes | No |
| Ability to call native code | No | No | Yes |

SAFE is the most reliable and secure mode with associated restrictions in terms of the allowed programming model. SAFE assemblies are given enough permission to run, perform computations, and have access to the local database. SAFE assemblies need to be verifiably type safe and are not allowed to call unmanaged code.

UNSAFE is for highly trusted code that can only be created by database administrators. This trusted code has no code access security restrictions, and it can call unmanaged (native) code.

EXTERNAL\_ACCESS provides an intermediate security option, allowing code to access resources external to the database but still having the reliability guarantees of SAFE.

SQL Server uses the host-level CAS policy layer to set up a host policy that grants one of the three sets of permissions based on the permission set stored in SQL Server catalogs. Managed code running inside the database always gets one of these code access permission sets.

### Programming Model Restrictions

The programming model for managed code in SQL Server involves writing functions, procedures, and types which typically do not require the use of state held across multiple invocations or the sharing of state across multiple user sessions. Further, as described earlier, the presence of shared state can cause critical exceptions that impact the scalability and the reliability of the application.

Given these considerations, we discourage the use of static variables and static data members of classes used in SQL Server. For SAFE and EXTERNAL\_ACCESS assemblies, SQL Server examines the metadata of the assembly at CREATE ASSEMBLY time and fails the creation of such assemblies if it finds the use of static data members and variables.

SQL Server also disallows calls to .NET Framework APIs that are annotated with the **SharedState**, **Synchronization** and **ExternalProcessMgmt** host protection attributes. This prevents SAFE and EXTERNAL\_ACCESS assemblies from calling any APIs that enable sharing state, performing synchronization, and affecting the integrity of the SQL Server process. For more information, see [CLR Integration Programming Model Restrictions](http://msdn.microsoft.com/en-us/library/ms403273.aspx).

**Performance of CLR Integration**

This topic discusses some of the design choices that enhance the performance of Microsoft SQL Server integration with the Microsoft .NET Framework common language runtime (CLR).

http://i.msdn.microsoft.com/Global/Images/clear.gif The Compilation Process

During compilation of SQL expressions, when a reference to a managed routine is encountered, a Microsoft intermediate language (MSIL) stub is generated. This stub includes code to marshal the routine parameters from SQL Server to the CLR, invoke the function, and return the result. This "glue" code is based on the type of parameter and on parameter direction (in, out, or reference).

The glue code enables type-specific optimizations and ensures efficient enforcement of SQL Server semantics, such as nullability, constraining facets, by-value, and standard exception handling. By generating code for the exact types of the arguments, you avoid type coercion or wrapper object creation costs (called "boxing") across the invocation boundary.

The generated stub is then compiled to native code and optimized for the particular hardware architecture on which SQL Server executes, using the JIT (just-in-time) compilation services of the CLR. The JIT services are invoked at the method level and allow the SQL Server hosting environment to create a single compilation unit that spans both SQL Server and CLR execution. Once the stub is compiled, the resulting function pointer becomes the run-time implementation of the function. This code generation approach ensures that there are no additional invocation costs related to reflection or metadata access at run time.

**Fast Transitions Between SQL Server and CLR**

The compilation process yields a function pointer that can be called at run time from native code. In the case of scalar-valued user-defined functions, this function invocation happens on a per-row basis. To minimize the cost of transitioning between SQL Server and the CLR, statements that contain any managed invocation have a startup step to identify the target application domain. This identification step reduces the cost of transitioning for each row.

http://i.msdn.microsoft.com/Global/Images/clear.gif Performance Considerations

The following summarizes performance considerations specific to CLR integration in SQL Server. More detailed information can be found in "[Using CLR Integration in SQL Server 2005](http://go.microsoft.com/fwlink/?LinkId=50332)" on the MSDN Web site. General information regarding managed code performance can be found in "[Improving .NET Application Performance and Scalability](http://go.microsoft.com/fwlink/?LinkId=50333)" on the MSDN Web site.

**User-Defined Functions**

CLR functions benefit from a quicker invocation path than that of Transact-SQL user-defined functions. Additionally, managed code has a decisive performance advantage over Transact-SQL in terms of procedural code, computation, and string manipulation. CLR functions that are computing-intensive and that do not perform data access are better written in managed code. Transact-SQL functions do, however, perform data access more efficiently than CLR integration.

**User-Defined Aggregates**

Managed code can significantly outperform cursor-based aggregation. Managed code generally performs slightly slower than built-in SQL Server aggregate functions. We recommend that if a native built-in aggregate function exists, you should use it. In cases in which the needed aggregation is not natively supported, consider a CLR user-defined aggregate over a cursor-based implementation for performance reasons.

**Streaming Table-Valued Functions**

Applications often need to return a table as a result of invoking a function. Examples include reading tabular data from a file as part of an import operation, and converting comma-separated-values to a relational representation. Typically, you can accomplish this by materializing and populating the result table before it can be consumed by the caller. The integration of the CLR into SQL Server introduces a new extensibility mechanism called a streaming table-valued function (STVF). Managed STVFs perform better than comparable extended stored procedure implementations.

STVFs are managed functions that return an **IEnumerable** interface. **IEnumerable** has methods to navigate the result set returned by the STVF. When the STVF is invoked, the returned **IEnumerable** is directly connected to the query plan. The query plan calls **IEnumerable** methods when it needs to fetch rows. This iteration model allows results to be consumed immediately after the first row is produced, instead of waiting until the entire table is populated. It also significantly reduces the memory consumed by invoking the function.

**Arrays vs. Cursors**

When Transact-SQL cursors must traverse data that is more easily expressed as an array, managed code can be used with significant performance gains.

**String Data**

SQL Server character data, such as **varchar**, can be of the type SqlString or SqlChars in managed functions. SqlString variables create an instance of the entire value into memory. SqlChars variables provide a streaming interface that can be used to achieve better performance and scalability by not creating an instance of the entire value into memory. This becomes particularly important for large object (LOB) data. Additionally, server XML data can be accessed through a streaming interface returned by **SqlXml.CreateReader()**.

**CLR vs. Extended Stored Procedures**

The Microsoft.SqlServer.Server application programming interfaces (APIs) that allow managed procedures to send result sets back to the client perform better than the Open Data Services (ODS) APIs used by extended stored procedures. Furthermore, the System.Data.SqlServer APIs support data types such as **xml**, **varchar(max)**, **nvarchar(max)**, and **varbinary(max)**, introduced in SQL Server 2005, while the ODS APIs have not been extended to support the new data types.

With managed code, SQL Server manages use of resources such as memory, threads, and synchronization. This is because the managed APIs that expose these resources are implemented on top of the SQL Server resource manager. Conversely, SQL Server has no view or control over the resource usage of the extended stored procedure. For example, if an extended stored procedure consumes too much CPU or memory resources, there is no way to detect or control this with SQL Server. With managed code, however, SQL Server can detect that a given thread has not yielded for a long period of time, and then force the task to yield so that other work can be scheduled. Consequently, using managed code provides for better scalability and system resource usage.

Managed code may incur additional overhead necessary to maintain the execution environment and perform security checks. This is the case, for example, when running inside SQL Server and numerous transitions from managed to native code are required (because SQL Server needs to do additional maintenance on thread-specific settings when moving out to native code and back). Consequently, extended stored procedures can significantly outperform managed code running inside SQL Server for cases in which there are frequent transitions between managed and native code.

|  |
| --- |
| **ms131075.note(en-us,SQL.100).gifNote:** |
| It is recommended that you do not develop new extended stored procedures, because this feature has been deprecated. |

**Native Serialization for User-Defined Types**

User-defined types (UDTs) are designed as an extensibility mechanism for the scalar type system. SQL Server implements a serialization format for UDTs called **Format.Native**. During compilation, the structure of the type is examined to generate MSIL that is customized for that particular type definition.

Native serialization is the default implementation for SQL Server. User-defined serialization invokes a method defined by the type author to do the serialization. **Format.Native** serialization should be used when possible for best performance.

**Normalization of Comparable UDTs**

Relational operations, such as sorting and comparing UDTs, operate directly on the binary representation of the value. This is accomplished by storing a normalized (binary ordered) representation of the state of the UDT on disk.

Normalization has two benefits: it makes the comparison operation considerably less expensive by avoiding the construction of the type instance and the method invocation overhead; and it creates a binary domain for the UDT, enabling the construction of histograms, indexes, and histograms for values of the type. Consequently, normalized UDTs have a very similar performance profile to the native built-in types for operations that do not involve method invocation.

**Scalable Memory Usage**

In order for managed garbage collection to perform and scale well in SQL Server, avoid large, single allocation. Allocations greater than 88 kilobytes (KB) in size will be placed on the Large Object Heap, which will cause garbage collection to perform and scale much worse than many smaller allocations. For example, if you need to allocate a large multi-dimensional array, it is better to allocate a jagged (scattered) array.

**Overview of CLR Integration**

The common language runtime (CLR) is the heart of the Microsoft .NET Framework and provides the execution environment for all .NET Framework code. Code that runs within the CLR is referred to as managed code. The CLR provides various functions and services required for program execution, including just-in-time (JIT) compilation, allocating and managing memory, enforcing type safety, exception handling, thread management, and security. See the .NET Framework SDK for more information.

With the CLR hosted in Microsoft SQL Server (called CLR integration), you can author stored procedures, triggers, user-defined functions, user-defined types, and user-defined aggregates in managed code. Because managed code compiles to native code prior to execution, you can achieve significant performance increases in some scenarios.

Managed code uses Code Access Security (CAS) to prevent assemblies from performing certain operations. SQL Server uses CAS to help secure the managed code and prevent compromise of the operating system or database server.

http://i.msdn.microsoft.com/Global/Images/clear.gif Advantages of CLR Integration

Transact-SQL is specifically designed for direct data access and manipulation in the database. While Transact-SQL excels at data access and management, it is not a full-fledged programming language. For example, Transact-SQL does not support arrays, collections, for-each loops, bit shifting, or classes. While some of these constructs can be simulated in Transact-SQL, managed code has integrated support for these constructs. Depending on the scenario, these features can provide a compelling reason to implement certain database functionality in managed code.

Microsoft Visual Basic .NET and Microsoft Visual C# offer object-oriented capabilities such as encapsulation, inheritance, and polymorphism. Related code can now be easily organized into classes and namespaces. When you are working with large amounts of server code, this allows you to more easily organize and maintain your code.

Managed code is better suited than Transact-SQL for calculations and complicated execution logic, and features extensive support for many complex tasks, including string handling and regular expressions. With the functionality found in the .NET Framework Library, you have access to thousands of pre-built classes and routines. These can be easily accessed from any stored procedure, trigger or user defined function. The Base Class Library (BCL) includes classes that provide functionality for string manipulation, advanced math operations, file access, cryptography, and more.

|  |
| --- |
| **ms131045.note(en-us,SQL.100).gifNote:** |
| While many of these classes are available for use from within CLR code in SQL Server, those that are not appropriate for server-side use (for example, windowing classes), are not available. For more information, see [Supported .NET Framework Libraries](http://msdn.microsoft.com/en-us/library/ms403279.aspx). |

One of the benefits of managed code is type safety, or the assurance that code accesses types only in well-defined, permissible ways. Before managed code is executed, the CLR verifies that the code is safe. For example, the code is checked to ensure that no memory is read that has not previously been written. The CLR can also help ensure that code does not manipulate unmanaged memory.

CLR integration offers the potential for improved performance. For information, see [Performance of CLR Integration](http://msdn.microsoft.com/en-us/library/ms131075.aspx).

http://i.msdn.microsoft.com/Global/Images/clear.gif Choosing Between Transact-SQL and Managed Code

When writing stored procedures, triggers, and user-defined functions, one decision you must make is whether to use traditional Transact-SQL, or a .NET Framework language such as Visual Basic .NET or Visual C#. Use Transact-SQL when the code will mostly perform data access with little or no procedural logic. Use managed code for CPU-intensive functions and procedures that feature complex logic, or when you want to make use of the BCL of the .NET Framework.

**Choosing Between Execution in the Server and Execution in the Client**

Another factor in your decision about whether to use Transact-SQL or managed code is where you would like your code to reside, the server computer or the client computer. Both Transact-SQL and managed code can be run on the server. This places code and data close together, and allows you to take advantage of the processing power of the server. On the other hand, you may wish to avoid placing processor intensive tasks on your database server. Most client computers today are very powerful, and you may wish to take advantage of this processing power by placing as much code as possible on the client. Managed code can run on a client computer, while Transact-SQL cannot.

http://i.msdn.microsoft.com/Global/Images/clear.gif Choosing Between Extended Stored Procedures and Managed Code

Extended stored procedures can be built to perform functionality not possible with Transact-SQL stored procedures. Extended stored procedures can, however, compromise the integrity of the SQL Server process, while managed code that is verified to be type-safe cannot. Further, memory management, scheduling of threads and fibers, and synchronization services are more deeply integrated between the managed code of the CLR and SQL Server. With CLR integration, you have a more secure way than extended stored procedures to write the stored procedures you need to perform tasks not possible in Transact-SQL. For more information about CLR integration and extended stored procedures, see [Performance of CLR Integration](http://msdn.microsoft.com/en-us/library/ms131075.aspx).

**Enabling CLR Integration**

The common language runtime (CLR) integration feature is off by default, and must be enabled in order to use objects that are implemented using CLR integration. To enable CLR integration, use the **clr enabled** option of the **sp\_configure** stored procedure:

[[http://i.msdn.microsoft.com/Global/Images/clear.gif](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl01other');)Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl01other');)

sp\_configure 'show advanced options', 1;

GO

RECONFIGURE;

GO

sp\_configure 'clr enabled', 1;

GO

RECONFIGURE;

GO

You can disable CLR integration by setting the **clr enabled** option to 0. When you disable CLR integration, SQL Server stops executing all CLR routines and unloads all application domains.

|  |
| --- |
| **ms131048.note(en-us,SQL.100).gifNote:** |
| To enable CLR integration, you must have ALTER SETTINGS server level permission, which is implicitly held by members of the **sysadmin** and **serveradmin** fixed server roles. |
| **ms131048.note(en-us,SQL.100).gifNote:** |
| Computers configured with large amounts of memory and a large number of processors may fail to load the CLR integration feature of SQL Server when starting the server. To address this issue, start the server by using the **-g** **memory\_to\_reserve** SQL Server service startup option, and specify a memory value large enough. For more information, see [Using the SQL Server Service Startup Options](http://msdn.microsoft.com/en-us/library/ms190737.aspx). |

|  |
| --- |
| **ms131048.note(en-us,SQL.100).gifNote:** |
| Common language runtime (CLR) execution is not supported under lightweight pooling. Before enabling CLR integration, you must disable lightweight pooling. For more information, see [lightweight pooling Option](http://msdn.microsoft.com/en-us/library/ms178074.aspx). |