

Yukon Basics

XML, T-SQL, and the CLR Create a New World of Database Programming

[Eric Brown](#)



This article was based on Beta 1 of Microsoft SQL Server Code Name "Yukon" and all information contained herein is subject to change

Note: This document was developed prior to the product's release to manufacturing and, as such, we cannot guarantee that any details included herein will be exactly the same as those found in the shipping product. The information represents the product at the time this document was printed and should be used for planning purposes only. Information subject to change at any time without prior notice.

SUMMARY The next version of SQL Server, code-named "Yukon," includes quite a few enhancements and expanded language support. For example, Transact-SQL now conforms more closely to the ANSI-99 SQL specification and makes querying more flexible and expressive. Yukon can execute user-defined functions, stored procedures, and triggers written in CLR-targeted languages, including Visual Basic .NET and C#. It supports a subset of the W3C standard XQuery language, and has native XML support.

In this article, the author outlines the most significant language features and builds an order-entry sample app.

During the planning phase of the next version of Microsoft® SQL Server™, code-named "Yukon," much consideration was given to the future of databases and the programming capabilities of SQL Server. Developers inside Microsoft realized early on that the future had to include a more symmetrical programming model, plus significantly more flexibility for different data models. The goal of programming model symmetry would mean that typical data access and manipulation tasks could be performed in a variety of ways—using XML, the Microsoft .NET Framework, or Transact-SQL (T-SQL) code.

The result of this planning is a new database programming platform that's been extended in a number of directions. First, the ability to host the .NET Framework common language runtime (CLR) extends the database into the land of procedural programming and managed code. Second, .NET Framework hosting integration provides powerful object database capabilities from within SQL Server. Deep support for XML was gained via a full-fledged XML data type which carries all the capabilities of relational data types. Additionally, server-side support for XML Query (XQuery) and XML Schema Definition language (XSD) standards was added. Finally, SQL Server Yukon includes significant enhancements to the T-SQL language.

These new programming models and enhanced languages come together to create a range of programmability that both complements and builds upon the current relational database model. The net result of this architectural work is the ability to create more scalable, reliable, robust applications, and an increase in developer productivity. Another result of these models is a new application framework called the SQL Service Broker—a distributed application framework for asynchronous message delivery. I'll get into a deeper discussion of the Yukon SQL Service Broker later in this article, but let's first look at the major changes and enhancements in the programming model, starting with T-SQL.

Transact-SQL Enhancements

Yukon provides more new language constructs and primitives for the T-SQL language than can be enumerated here. You can find them all in SQL Server Yukon Books Online. The enhancements to the T-SQL language reflect greater conformity to the ANSI-99 SQL specification, and were prompted by customer requests. Many of the improvements in T-SQL are focused on greater expressiveness in queries. There are several new query types that allow for common scenarios to be covered in T-SQL code. For example, a recursive query provides the ability to generate a bill of materials or a hierarchical resultset.

T-SQL in Yukon provides new PIVOT and UNPIVOT operators. These operators perform a manipulation on an input table-valued expression and produce an output table as a resultset. The PIVOT operator rotates rows into columns, optionally performing aggregations or other mathematical calculations along the way. It widens the input table expression based on a given pivot column and generates an output table with a column for each unique value in the pivot column. The UNPIVOT operator performs an operation opposite to the one PIVOT performs; it rotates columns into rows. The UNPIVOT operator narrows the input table expression based on a pivot column.

Yukon introduces a new and simple but very powerful exception handling mechanism in the form of a TRY/CATCH T-SQL construct. Transaction abort errors that used to cause a batch to terminate can now be caught and handled. Additionally, there are new language constructs for security, replication, Notification Services, XML, and all of the

features that the .NET Framework provides. The .NET Framework implementation on the server is an important evolution in the SQL Server product cycle.

SQL Server Yukon .NET Programming

Developers can now write stored procedures, triggers, and user-defined functions (UDFs) with any Microsoft .NET-compliant language, including Visual Basic® .NET and C#. In addition, three new objects—user-defined types (UDTs), aggregates, and functions—can be created in managed code. The CLR is the heart of the .NET Framework and provides the execution environment for all .NET Framework-based code. The CLR provides various functions and services required for program execution, including just-in-time compilation, memory management and allocation, type safety enforcement, exception handling, thread management, and security. SQL Server Yukon acts as the host for this functionality in much the same way an operating system hosts an application.

As a result of CLR integration, Yukon also features automatic memory management, resource allocation, and garbage collection. SQL Server provides .NET assemblies direct access to database objects by entering the SQL Server execution space. Data access is achieved through a special form of ADO.NET. This article doesn't cover the ramifications of ADO.NET functionality within the database. However, because the access methods are similar to those developers are familiar with, using the functionality of the CLR is easy.

The way that Yukon integrates the .NET Framework with SQL Server provides several major benefits to database developers, including the following:

Enhanced programming model CLR-compliant languages are in many respects richer than T-SQL, offering constructs and capabilities previously not available to SQL developers. In addition, there is a set of class libraries (Framework APIs) which is much richer than the built-in functions SQL provides natively.

Enhanced safety and security Managed code runs in a CLR environment, hosted by the database engine. This allows .NET database objects to be safer and more secure than the extended stored procedures available in earlier versions of SQL Server.

User-defined types and aggregates Two new database objects that expand SQL Server storage and querying capabilities are enabled by hosting the CLR.

Common development environment Database development integration is a planned feature of future releases of the 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.

Performance and scalability In certain situations, the .NET language compilation and execution models deliver improved performance over T-SQL.

Taking Advantage of the CLR

In previous versions of SQL Server, database programmers were limited to using T-SQL when writing code on the server side. With CLR integration, database developers can now perform tasks that were impossible or difficult to achieve with T-SQL alone. Both Visual Basic .NET and C# are modern programming languages offering full support for arrays, structured exception handling, and collections. Developers can take advantage of CLR integration to write code that has more complex logic and is more suited for computational tasks than T-SQL can support.

What's more, Visual Basic .NET and C# offer object-oriented capabilities such as encapsulation, inheritance, and polymorphism. Related code can now be easily organized into classes and namespaces. When working with large amounts of code on the server, this allows you to more easily organize and maintain your code investments. This ability to logically and physically organize code into assemblies and namespaces is a huge benefit, and will allow you to better find and relate different pieces of code in a large database implementation.

When designing UDTs, you'll need to consider such things as nullability and check constraints. You can even use private functions to extend the capability of the data type; for example, you might natively add XML serialization capabilities.

Additionally, once registered, assemblies can reference other assemblies on the server. This means you can load an assembly that contains common functionality which can be used by one or more other assemblies on the server. This allows for incremental reuse of general code. The database developer will definitely appreciate this modular approach as the object model of their application gets larger and more complex.

SQL Server Yukon has implemented three levels of security that limit the capabilities of assemblies registered in the database. The security model is an integration of two security models: the SQL Server security model, which is based on user authentication and authorization, and the CLR security model, which employs code access security at the code level.

The highest security level, SAFE, allows only computation and access to data. The second level,

EXTERNAL_ACCESS, also allows access to external system resources. The least secure level, UNSAFE, has no restrictions except for operations that compromise the stability of the server.

Choosing Between T-SQL and Managed Code

Managed code in the data tier allows you to perform number crunching and complicated execution logic on database objects. The .NET Framework features extensive support for string handling, regular expressions, error capture, and more. Additionally, with the functionality found in the .NET Framework base class library, database developers now have full access to thousands of pre-built classes and routines that can be easily accessed from any stored procedure, trigger, or UDF. For scenarios that require string handling, mathematical functions, date operations, system resource access, advanced encryption algorithms, file access, image processing, or XML data manipulation, managed store procedures, functions, triggers, and aggregates provide a much simpler programming model than T-SQL.

Another benefit of managed code is type safety. Before managed code is executed, the CLR performs server checks in order to verify that the code is actually safe to run. For example, the code's memory access is checked to ensure that no memory is being read if it hasn't been written to.

When writing stored procedures, triggers, and UDFs, one decision programmers will now have to make is whether to use traditional T-SQL or a CLR-compliant language such as Visual Basic .NET or C#. The answer depends on the needs of a particular project. T-SQL is best used when the code will perform mostly data access with little or no procedural logic. CLR-compliant languages are best suited for mathematically intensive functions and procedures that feature complex logic, or when you want to build a solution upon the .NET Framework base class libraries.

Code placement is another important feature of CLR integration. Both T-SQL and CLR-hosted code run inside the database engine on the server. Having .NET Framework functionality and the database close together allows you to take advantage of the processing power of a server machine. I recommend using the SQL Server Profiler to review the performance of your applications, and then make your choice based on the empirical results of your testing. The SQL Server Profiler has been enhanced to provide deep profiling of CLR performance within SQL Server, including a new graphical output format that can be used for comparison.

Now let's take a look at some of the capabilities of the .NET Framework and how they complement Yukon.

User-defined Types, Functions, and Aggregates

Yukon supports an extensible type system in the CLR. These types can be used as part of table definitions on the server side and can be manipulated as objects on the client side. UDTs allow you to extend the existing type system by creating new types whose implementation is in managed code. Some examples of UDTs are geospatial, custom financial, and specific date/time types. Think of UDTs as a contract between the server engine and the code.

In .NET terms, the UDT is a struct or a reference type, not a class or enum. This means that memory usage will be optimized. However, the UDT does not support inheritance and polymorphism. It can have both public and private functions. In fact, tasks such as constraint checking should be done as private classes. If your UDT is constructed of multiple pieces of information—for example, a geospatial type would contain longitude, latitude, and maybe altitude—you will need to provide private fields that contain those elements. Once you've created a UDT, you can use T-SQL to register the type in SQL Server. When this happens, the bits of the DLL are actually stored in the database.

UDFs come in two varieties: scalar-valued and table-valued. A scalar-valued function is one that returns a single value such as a string, integer, or bit. Table-valued functions return a resultset that can be comprised of one or more columns.

The ability to create additional aggregate functions beyond those already in the box was not available to developers using previous releases of SQL Server. With Yukon, developers can create custom aggregate functions in managed code and make these functions accessible to T-SQL or other managed code. The user-defined aggregate is also a .NET class that expects to reference an assembly already available in the database.

You can use a UDAgg to translate data stored in the database into mathematical values. Take, for example, a statistical function. You could store data containing results from a quantitative survey in a relation table. In order to figure out the weighted mean or standard deviation of these results, you could call a UDAgg that calculates and returns that information.

Managed Stored Procedures

Managed code routines that work as stored procedures are best used when you want to return a simple resultset to the caller with modifications, or if you want to simply execute a data manipulation language (DML) or data definition language (DDL) statement. Once registered on the server, taking advantage of the code is as simple as writing a

stored procedure.

To wrap a managed code method using a stored procedure, use the CREATE PROCEDURE statement. CREATE PROCEDURE uses the syntax shown in this prototype:

```
CREATE PROCEDURE mysproc (@username NVARCHAR(4000))
AS
EXTERNAL NAME YukonCLR:[MyNamespace.CLRCode]:: myfunction
```

XML and SQL Server Yukon

The XML story in SQL Server Yukon really begins with SQL Server 2000. That version of SQL Server introduced the ability to return relational data as XML, bulk load and shred XML documents, and expose database objects as XML-based Web services. The Web Services Toolkit, also known as SQLXML 3.0, provides Web services capability to stored procedures, XML templates, and SQL Server UDFs. Two important pieces were missing in the XML technology: a native XML storage mechanism—the XML data type—and a sophisticated query language that supports querying semi-structured data. Yukon delivers these two elements, plus more Web services enhancements and extensions to the T-SQL FOR XML statement. Let's start with the central enhancement to the native SQL Server data types—the XML data type.

XML Data Type

XML has evolved from a representational technology to a wire format and is now seen as a storage format. Persistent storage in XML has become an interesting story, and there are many possible applications of the XML data type. First, XML is useful when you don't know the schema of the object. Second, the XML data type is useful for dynamic schemas in situations where information is reshaped on a continual basis. Third, XML persistence is central to XML document applications. There isn't a single application scenario that best exemplifies proper XML data type usage. The programming example I'll present later in this article uses XML as a storage format for a dynamic schema that represents a customer order in the form of a message to the server.

The XML data type is a full-fledged data type. It has all the powers and capabilities of the other types found in SQL Server. The importance of this can't be overstated. As a full-fledged type, the XML column can be indexed, it can have row and column constraints via XSD schema (although an XSD schema is not needed), and it can be queried using an XQuery expression embedded in T-SQL. These functions are appended to the XQuery W3C specification. In addition, using the xmldt::modify method, the user can add subtrees, delete subtrees, and update scalar values.

The XML data type is flexible. You can choose whether you want to associate an XML Schema Definition (XSD) schema with a column. When a column has an XSD schema association, it is known as typed XML. Without the XSD association, it's untyped XML. An XSD schema can be used to type an XML column after it's imported into the database. It can then be used to create indexes and other metadata for the system catalog. Typed XML columns are significantly faster and more flexible than untyped columns when you're looking to query them. The usage scenario will dictate the XML column type, though most applications will appreciate an XSD schema reference.

Note also that you can store both XML documents and XML fragments in the same column. In addition, you have to create a special "XML" index on XML columns. Doing this creates indexes for tags, values, and paths in the XML value.

XSD and XML Data Type

When creating your XML column, you should add an XSD document and associate it with the column. The XSD schema association is executed from within the SQL Server Workbench. The XSD association is also possible using DDL:

```
CREATE TABLE T (pk INT, xCol('mySchema'))
```

When a schema is imported into the database, it is parsed into various types of components, including ELEMENT, ATTRIBUTE, TYPE (for simple or complex types), ATTRIBUTEGROUP, and MODELGROUP. These components, and any associated namespaces, are then stored in various system tables. You can view these components using dedicated system views. In other words, the schema is not stored intact. This has at least two ramifications. First, the schema tag and its associated attributes are not stored. Instead, the XML tag attributes—targetNamespace, attributeFormDefault, elementFormDefault, and so on—are reassigned to the schema's components. Second, SQL Server makes no provision for recovering and viewing the original schema document once it is imported. It is

recommended that you keep a copy of all schemas or store their contents in a dedicated database table. For this, an XML column will suffice. Alternatively, you can retrieve XML schemas using the built-in function `XML_SCHEMA_NAMESPACE ('uri')`. This returns the content of namespace 'uri'.

When the XML column and an XSD schema are combined, the query capabilities for XML data are complementary to normal relational data and queries. While I don't recommend storing all your data in XML, the ability to take advantage of XML in the database makes XML document management and other application scenarios a breeze.

Now that the data has been stored, I'll move on to XQuery, as my discussion is incomplete without it.

XML Query

The XML Query Language, commonly referred to as XQuery, is an intelligent and robust language optimized for querying all types of XML data. With XQuery, you can run queries against variables and columns of the XML data type using the type's associated methods. With the invocation of the query method, you can query across relational and XML data in the same batch.

As with many of the XML standards, the development of XQuery, which is currently a working draft, is overseen by the W3C. Yukon supports a statically typed subset of the XQuery 1.0 Working Drafts in the range of December 20, 2001 to November 15, 2002. See <http://www.w3c.org> for all of the drafts.

XQuery evolved from a query language called Quilt, which was based on a variety of other languages such as XPath 1.0, XQL, and SQL. It also contains a subset of XPath 2.0. You can use your existing skills with XPath and not have to learn an entirely new language. However, SQL Server Yukon contains significant enhancements that go beyond XPath, such as special functions and support for better iteration, sorting of results, and construction.

XQuery statements consist of a prologue and a body. The prologue contains one or more namespace declarations and/or schema imports that create the context for query processing. The body contains a sequence of expressions that specify the query criteria. These statements are then used within one of the aforementioned methods of the XML data type (query, value, exist, or modify).

XQuery is capable of querying typed XML (data that is associated with an XML schema) as well as an XML document.

Meet Microsoft XQuery Designer

XQuery Designer is a new tool, integrated into the new SQL Server Workbench, which makes working with XML data easy. XQuery Designer helps you write queries that manipulate and retrieve data from both XML columns and XML documents. The central development theme for the XQuery Designer was to empower XQuery development without making users learn the ins and outs of the XML Query Language. For a peek at the XQuery Designer in action see [Figure 1](#).

Up to this point, I've talked about the new .NET functionality and XML functionality of Yukon as separate entities. The combination of these technologies creates new opportunities to solve application scenarios in popular areas like e-commerce. SQL Server development is newly focused on distributed applications, where business workflow and logic is represented in the database application. Using the powerful support for Web-based applications that's built into Yukon, let's delve deeper into the new functionality I've presented here.

Meet the SQL Server Service Broker

Over the last 10 years, the proliferation of e-commerce applications has created a need for increased process management across database applications. If you've built an order entry system or made an online purchase, you are familiar with the workflow model. When a customer places an order for a book, a transaction must be committed into the inventory, the shipping, and the credit card systems, and an order confirmation must be sent through another Web application. Waiting for each of these processes to happen synchronously doesn't scale well. In the past, developers had to write complicated stored procedures and use remote procedure call code to queue messages. Yukon provides a new, scalable architecture for building asynchronous message routing.

The SQL Server Service Broker is a Yukon technology that allows internal or external processes to send and receive guaranteed, asynchronous messages using extensions to normal T-SQL DML. Messages can be sent to a queue in a database shared with the sender, to another database in the same SQL Server instance, or to another SQL Server instance either on the same server machine or on a remote server.

In traditional messaging systems, the application is responsible for ordering and coordinating messages. Message duplication, synchronization, and order are all difficult problems enterprise data systems must deal with.

SQL Server Service Broker solves these problems by automatically handling message order, unique delivery, and conversation identification. Once a conversation is established between two service broker endpoints, an application

receives each message only once, in the order in which the message was sent. Your application can process messages exactly once, in order, without additional code. Service Broker automatically includes an identifier in every message. An application can always tell which conversation a particular message belongs to.

Service Broker queues messages for delivery. If the target service is not immediately available, the message remains with the sending service and delivery attempts are repeated until the message is sent successfully. This allows a conversation to continue reliably between two services, even if one service is temporarily unavailable at some point during the conversation.

SQL Server Service Broker provides loose coupling between the initiating service and the target service. A service can put a message on a queue and then continue with its application processing tasks, relying on SQL Server Service Broker to ensure that the message reaches its destination. This loose coupling enables scheduling flexibility. The initiator can send out multiple messages, and multiple target services can process them in parallel. Each target service processes messages at its own pace, depending on the system's current workload.

Queuing also allows systems to distribute processing more evenly, reducing the peak capacity required by a server. This can improve overall throughput and performance in database applications. For example, an order entry application might see an increase in requests at noon every day, resulting in high resource usage and slow response times. With Service Broker, the order entry application need not perform all of the processing for an order when the application receives it. Instead, the application can enter the order and submit requests for background processing such as billing, shipping, and inventory management. Background applications perform the processing reliably over a period of time, while the order entry application continues to receive new orders.

One of the most difficult things to accomplish in a messaging application is to allow multiple programs to read in parallel from the same queue. Situations like this can cause messages to be processed out of order, even if they're received in order.

Consider a traditional order processing application. Message A, containing instructions about creating the order header, and Message B, containing instructions about creating the order line items, are both received on the queue. If both of these messages are dequeued by separate service program instances and processed at the same time, it is possible that the order line item transaction will attempt to commit first, and fail because the order does not yet exist. The failure causes the transaction to roll back and the message to be requeued and processed again, wasting resources. Traditionally, this problem was solved by combining the information from Message A and Message B in a single message. While this approach is straightforward for two messages, it scales poorly to systems that involve coordinating dozens or hundreds of messages.

Service Broker solves this problem by automatically locking all messages related to the same task, so that these messages can only be received and processed by one service program instance. Meanwhile, other service program instances can continue to dequeue and process messages related to other tasks. This allows multiple parallel service programs to work reliably and efficiently.

One of the most useful features of Service Broker is activation. Activation automatically starts a service program to read messages from a queue as they arrive. If messages are arriving faster than they are being handled, additional instances of the service program are started, up to a configured maximum. If the message arrival rate is reduced and an active service program checks the queue and finds that there are no messages available for processing, the service program shuts down. This allows the number of service program instances to grow and shrink dynamically as the load on the service changes. If the system goes down or is rebooted, service programs are automatically started to read the messages in the queue when the system comes back up. Traditional messaging systems lack this behavior, and frequently end up having either too many or too few resources dedicated to a particular queue at any given time.

Database Integration

The integrated design of Service Broker provides benefits for application performance and administration. Integration with SQL Server allows transactional messaging; that is, each loop of a service program—receiving a message, processing the message, and sending a reply message—can be enclosed in a single database transaction. If the transaction fails, all work rolls back, and the received message is requeued so that another attempt can be made to process it. No actions take effect until the application commits the transaction. The application remains in a consistent state, without the added resource overhead and complexity of a distributed transaction coordinator.

Administration is easier when your data, messages, and application logic are all in the database. Only one item—instead of three or four separate components—requires maintenance for disaster recovery, clustering, security, backup, and so on. For instance, with traditional messaging systems, the message store and the database can become out of sync. For example, when one component is restored from a backup, the other component must also

be restored from a backup taken at the same time, or the message store and the database may be inconsistent. With a single database for both messages and data, this is no longer an issue.

Using a common development environment is also a benefit. The messaging and data parts of an application can use the same languages and tools in a Service Broker application. This extends the developer's familiarity with database programming techniques to encompass message-based programming. Stored procedures that implement a Service Broker service can be written in either T-SQL or one of the CLR-targeted languages.

Additionally, database integration makes automatic resource management possible. Service Broker runs in the context of the SQL Server instance, so the broker can maintain an aggregate view of all messages ready to be transmitted from all databases in the instance. This allows each database to maintain its own queues while still maintaining fairness in resource usage across the entire SQL Server instance.

SQL Server Yukon Programmability Scenario

Let's see how these concepts come together in an order entry application where a customer places orders on a Web site. The messages sent between the services are stored in an XML column. For illustration purposes, I'll use an ASP.NET-based Web service that's registered and hosted by an instance of the CLR within SQL Server. This service communicates with a partner to send and receive purchase order information. Using an XML column for the messages allows for schema encapsulation and simplification.

The customer places an order on the Web site. When the order transaction commits, a message containing the order information is put in the Order Entry Service Queue. The order is sent to the Order Entry Service as an XML column. This allows all the order information to be consolidated into one column. For a full view of the process, see [Figure 2](#).

The Order Entry Service starts a transaction, receives the message, and processes it. The Order Entry Service then sends a request on to the Credit Limit Service to verify credit status. If it is okay, the Order Entry Service moves to the next step.

The Order Entry Service checks the inventory for each item in the order. An XQuery processes the inventory check. At this point, if the item is available for shipping, a message is sent to the Shipping Service Queue.

The Shipping Service, a T-SQL stored procedure, uses XQuery to parse the XML message. The customer information is used to generate an order shipment. When the shipment is complete, a message is sent to the Billing Service Queue. When the Billing Service Queue receives the "ship complete" message, it updates the order status in the database to indicate the order has been shipped. All messages to and from the Order Entry Service are written to an audit table for later analysis and problem resolution.

You can see from this simple example how the new Yukon programmability features—XML, CLR, Service Broker, and T-SQL enhancements—can be used to build reliable, scalable, and complete database applications.

Conclusion

I've touched on just a few of the new and exciting architectural capabilities in Yukon. Database developers have never had so many choices for data access and storage. The SQL Server Service Broker example highlights only a fraction of the capabilities found when combining the new capabilities of SQL Server. Start working with the SQL Server Yukon beta and stay tuned to *MSDN® Magazine* for future coverage. Before you know it, you'll be using Yukon and never looking back.

Eric Brown joined the SQL Server team at Microsoft almost three years ago. Previously, he worked in data development at greatfood.com and other small companies.

From the [February 2004](#) issue of [MSDN Magazine](#).
Get it at your local newsstand, or better yet, [subscribe](#).

Figure 1 The XQuery Designer in Action

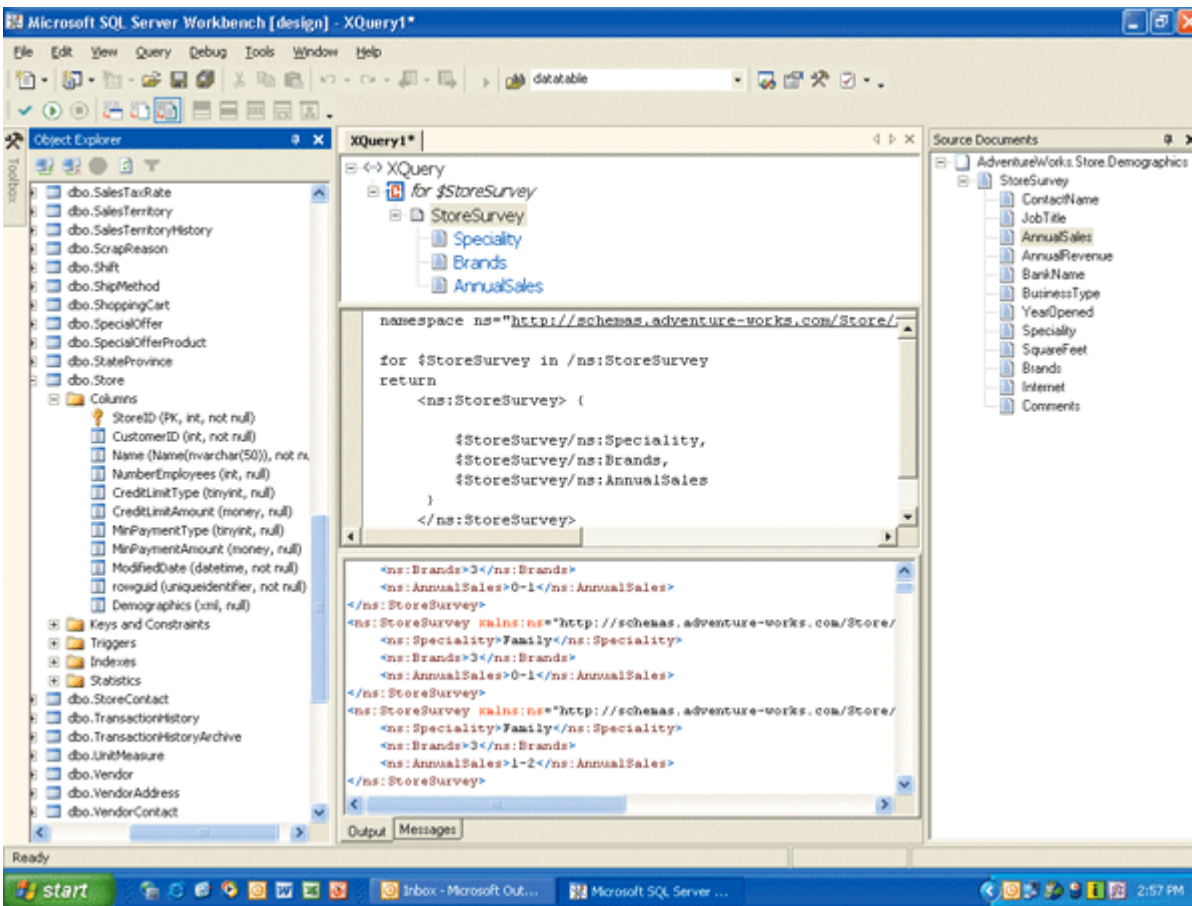
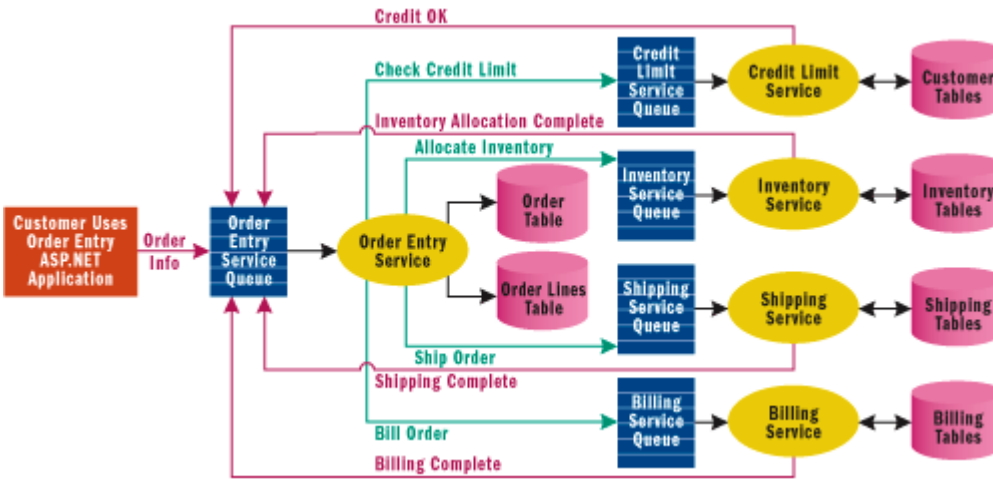


Figure 2 An Order Entry Application



© 2003 Microsoft Corporation. All rights reserved.