SQL/CLI - A New Binding Style For SQL

Murali Venkatrao, Microsoft Corp.
email: muraliv@microsoft.com

Michael Pizzo, Microsoft Corp.
mikep@microsoft.com

1. Introduction

In July of this year, the American and International committees responsible for the SQL standard finalized the specification for new binding style called the *Call Level Interface* (SQL/CLI) [2]. This new binding style is an addendum to the existing SQL Standard [1], and will comprise a new part in the next version of that standard. This paper takes a comprehensive look at SQL/CLI and explains why it is important to today's applications.

The paper is divided into 4 sections. Section 2 explains the motivation behind providing a new binding style and its relationship to existing binding styles. It also discusses the history and evolution of SQL/CLI. Section 3 gives an overview of SQL/CLI, exploring its various aspects and functionality. Finally, Section 4 summarizes the main ideas of this paper and takes a look at the future.

2. Motivation For SQL/CLI

The SQL Standard defines a *data sublanguage* -- that is, it is used in the context of another language (called the *host language*) in order to perform special purpose database management tasks. The relationship between a host language and the SQL data sublanguage is referred to as a *binding style*.

2.1 Traditional Binding Styles

SQL-92[1] defines three binding styles – Embedded SQL, Module Language and Direct Invocation. A brief discussion of each method is provided, focusing only on aspects that are required to establish a motivation for SQL/CLI. The interested reader is referred to [1] for comprehensive information.

2.1.1 Embedded SQL

Embedded SQL is, by far, the most popular of the three traditional binding styles. In this method, SQL statements are embedded directly into the text of the host language program. Before application code containing embedded SQL can be compiled, it must be precompiled using a preprocessor provided by the DBMS vendor. This preprocessor "splits" the embedded SQL code between SQL statements and statements in the host language. The resulting source code is then compiled and linked in the usual way. The SQL statements are submitted to the *DataBase Management System* (DBMS) in some implementation defined fashion for execution.

Dynamic SQL is a variant of embedded SQL which allows the application to construct SQL statements during run time. However, these Dynamic statements are themselves executed by Static SQL statements (PREPARE, EXECUTE and EXECUTE IMMEDIATE) embedded in the host language and precompiled by the DBMS specific preprocessor.

2.1.2 Module Language

In Module Language, all the SQL statements are present in one or more modules (for example, a *compilation unit* of PL/1, an object file in C, etc.). Each module contains one or more procedures; a procedure has a set of parameters and an SQL statement. Each procedure may be invoked from other modules within the application. An SQL module is written using module language and compiled and associated through a mechanism defined by the individual DBMS.

2.1.3 Direct Invocation

Direct Invocation defines a set of SQL statements which can be executed directly. In direct invocation, each DBMS vendor defines the method of invoking these statements, returning results, raising conditions, and accessing diagnostic information.

2.2 Why a New Binding Style?

2.2.1 DBMS Independence

In early database systems, applications were designed and implemented to perform custom business functions. Such applications were tightly coupled to the DBMS itself – frequently, both DBMS and the application were run on the same machine.

In today's world, database systems are increasingly being structured as clients and servers; applications run on a client machine with the database running on a separate server machine. This environment has encouraged the development of generic database

[1] Throughout this paper, the term "SQL-92" refers to the (identical) documents ANSI X3.135-1992 and ISO/IEC 9075:1992, "Database Language SQL".

applications for which the target database may not be known during application development. Since embedded SQL requires an implementation specific preprocessor, an application must, at a minimum, be recompiled for each DBMS it is to work with.

Similarly, Module and Direct Invocation rely on DBMS defined mechanisms for executing the SQL, and applications written and compiled for one DBMS will not work with any other DBMS.

SQL/CLI provides DBMS independence by allowing the development of portable object modules that perform database operations via *function calls*. These function calls are implemented in a run time library, which is referred to as the *CLI implementation*. Thus, no implementation specific transformations on the source code (such as preprocessing) is required.

2.2.2 Concurrent Processing

Many applications have a need to perform concurrent operations, including concurrent database operations. The existence of global data areas in both embedded SQL and module language raises the question of scope and visibility of changes by concurrent operations to such data.

SQL/CLI eliminates global data areas by introducing the concept of a *handle*. A handle identifies a data structure that resides in the CLI implementation. All data in the CLI implementation that is accessible to the application, is associated with a specific handle. Concurrent database operations occur on different handles.

2.2.3 Multiple Transactions

The traditional binding styles use COMMIT and ROLLBACK statements to delimit transactions. A transaction begins implicitly when the application operates on a database. This model works well in the case of a single transaction active on a single connection. It is difficult to extend to the case where the application wants to have multiple transactions active on multiple connections.

In SQL/CLI, there is an environment handle that provides a global context for database access. Within an environment, there can be one or more connection handles, each of which pertains to an active transaction. While the existence of multiple active transactions is implementation dependent in SQL/CLI, the model is designed to be extensible to allow for such functionality.

2.3 Evolution of SQL/CLI

During the late 1980's, several DBMS and application vendors began to recognize the need for a way to access data in heterogeneous data sources. Many vendors had generic applications that required access to more than one data store. Although the SQL language was relatively portable and interoperable, application developers needed a model in which there was no need to recompile the application every time access to a new data store was needed.

By late 1989 Microsoft, Lotus, Sybase and DEC had jointly developed a specification called *SQL Connectivity* that defined a set of functions for database access. Around the same time, an industry consortium of database vendors had formed the *SQL Access Group* (SAG) whose charter was to promote an interoperable SQL model. In June 1990, SAG accepted the *SQL Connectivity* specification as their base document for a *call level interface* (or CLI). For the next two years, SAG worked assiduously on CLI and released a preliminary specification in late 1992. This was adopted by ANSI SQL and ISO DBL committees as the base document for a new binding style for SQL.

Meanwhile, Microsoft Corp. released a software development kit based on their extended version of the CLI, called *Open DataBase Connectivity* (or ODBC). By 1993, a large number of applications and database vendors were using the tools provided in the ODBC SDK to implement this early version of the CLI.

By the end of 1994, SAG merged into X/Open and CLI activities became a working group within X/Open. The X/Open CLI was published in April 1995. In July 1995, SQL/CLI was approved as an international standard, officially called "ISO/IEC 9075-3:1995, Information Technology - Database Languages - SQL - Part 3, Call-Level Interface (SQL/CLI)". ISO plans to publish the standard by the end of this year.

3. Overview of SQL/CLI

SQL/CLI is a set of functions that an application can use to access SQL databases. It contains functions for allocating and deallocating resources, connecting and disconnecting from SQL-servers,[2] executing SQL statements, obtaining diagnostic information, controlling transaction termination and obtaining information about the implementation.

[2] Throughout this paper, the term "SQL-server" is used to indicate a source of data for which there exists a CLI implementation.

3.1 System Model

The SQL/CLI system model consists of three components – an Application, a CLI Implementation, and an SQL-server. The application makes calls to functions defined by SQL/CLI. The CLI Implementation is a run time library that implements the CLI functions, and is linked to the application. The CLI Implementation talks to an SQL-server, which processes the SQL statements.

3.2 Argument Types

Each parameter to a CLI function is one of the following four types:

1. *Input* arguments let the application pass data to the CLI Implementation.
2. *Output* arguments let the CLI Implementation return data to the application.
3. *Deferred Input* arguments are used to pass values to dynamic parameters in an SQL statement. When a function is called with a deferred input argument, the CLI Implementation saves the address of the argument, and retrieves the argument value at a later time.
4. *Deferred Output* arguments are used to retrieve column values. The CLI Implementation writes to the deferred output argument at a time other than when the function is called.

3.3 Handles and Attributes

A handle identifies a data structure within the CLI Implementation. There are four types of handles in SQL/CLI - environment handle, connection handle, statement handle and descriptor handle. The function *AllocHandle()* is used to allocate a handle of any type. For example, the following call in C allocates a connection handle:

```
SQLAllocHandle(SQL_HANDLE_DBC,
EnvHandle, &ConnectionHandle);
```

The first argument indicates the type of handle desired, the second argument is the handle in whose context the desired handle is to be allocated, and the third parameter is an output parameter in which the CLI Implementation will return the desired handle.

An environment is a global context for database access. It contains any data that must be global to the entire application such as current values of environment attributes, defined connections and information on which connection is current.

A connection represents an association between the application and the SQL-server. Associated with a connection is any data that must be global to the connection as a whole, such as transaction state information, certain pieces of diagnostic information, and the current values of connection attributes. A connection is allocated in the context of an environment. An application can be connected to several SQL-servers at the same time and/or may establish more than one connection to the same SQL-server.

A statement represents the state of a single SQL statement. Associated with a statement handle is information about the statement's execution such as dynamic parameters, the current values of statement attributes, result values and status information. A statement is allocated in the context of a connection, and a connection can contain one or more statements.

A descriptor holds information about either columns or dynamic parameters. A CLI descriptor is analogous to the SQL descriptor area in embedded SQL. Descriptors are discussed in more detail in Section 3.8.

Environments, connections and statements are characterized by certain *attributes*. The function *SetEnvAttr()* is used to set an environment attribute. The function *GetEnvAttr()* is used to obtain the current value of an environment attribute. *SetConnectAttr()* and *GetConnectAttr()* perform similar functions for connection attributes, as do *SetStmtAttr()* and *GetStmtAttr()* for statement attributes.

3.4 Establishing Connections

Prior to establishing a connection, the application must allocate an environment handle and then allocate a connection handle in the context of that environment. The *Connect()* function establishes a connection to an SQL-server. Consider the following call, made in C:

```
SQLConnect(ConnHandle,SvrName,
Len1,UsrName,Len2,Auth,Len3)
```

ConnHandle is the connection handle. Every function operates in the context of one of the 4 types of handles, and this handle is passed in as an input argument. SvrName is a character buffer containing the server name, and Len1 is the length of this buffer, in octets. Len1 can be set to the manifest constant SQL_NTS to indicate that SvrName is null-terminated. In general, any string input argument to a function always has an associated length argument, which can either contain the actual length, or can contain SQL_NTS indicating null termination.

SvrName usually contains a "friendly name", such as "Payroll Database" or "Accounts Database". This name maps to the actual physical server address in an implementation defined fashion. As a result, the application is protected from the complexities of networks and server addresses. UsrName and Auth provide logon information which the SQL-server uses to authenticate the user. Len2 and Len3 indicate the length of these buffers, or contain SQL_NTS.

The application can allocate another connection handle and call *Connect()* to establish a second connection to the same or different SQL-server. The number of such connections is implementation defined.

3.5 Executing SQL Statements

Once a connection is established, the application allocates a statement handle within the context of an established connection. SQL/CLI requires that a connection first be established before allocating a statement handle.

SQL Statements can be executed in one of two ways:

1. Call *Prepare()* and then call *Execute()*. *Prepare()* takes three arguments: the statement handle, a character buffer that holds the SQL statement to be prepared and the associated length argument. The CLI Implementation submits the statement to the SQL-server, which prepares it for execution. The application then calls *Execute()* using the same statement handle as it did when calling *Prepare()* to execute the SQL statement. After preparing the statement, the application can call *Execute()* as many times as it wants for repeated execution. This is equivalent to the PREPARE/EXECUTE statements in embedded SQL.

2. Call *ExecDirect()*. This function takes as arguments the statement handle, the SQL statement to be executed, and its length. It is used to execute the statement once. It is similar to the EXECUTE IMMEDIATE method of Embedded SQL, except that it can support dynamic parameters just like the Prepare/Execute model.

The SQL statements that can be executed are any SQL statement that can be prepared using dynamic SQL, as specified in SQL-92: CREATE, *cursor-specification*, searched DELETE, dynamic positioned DELETE, DROP, GRANT, INSERT, REVOKE, searched UPDATE, dynamic positioned UPDATE. COMMIT and ROLLBACK statements are specifically excluded from execution using CLI since other methods of transaction delimitation are available (see Section 3.11).

3.6 Parameterized Execution

Statements executed through SQL/CLI may contain *dynamic parameter markers*. Dynamic parameter markers appear as question marks (?) within a statement, and the SQL/CLI implementation substitutes the current values from application memory locations at execute time. The association of parameter markers with application memory locations is called *parameter binding*. As an alternative to binding, the application can specify parameter data during execute time using the *PutData()* routine.

Dynamic Parameters are useful for executing a single prepared statement multiple times with different parameter values.

3.7 CLI Cursor Model

Many SQL statements return a set of rows from the SQL-server. Traditional programming languages are not well equipped to handle such sets of rows. Embedded SQL and module language use the concept of a *cursor* to facilitate this task. A cursor is like a pointer that is positioned on a given row in the returned set of rows (sometimes called a *result set*). The cursor moves (or *scrolls*) through the result set, enabling the application to retrieve one row at a time. An SQL statement that can generate a result set is called a *cursor specification*. Embedded SQL and module language require that *cursor specification* be executed by the OPEN statement (which opens a cursor) instead of the EXECUTE or EXECUTE IMMEDIATE statements.

SQL/CLI also uses the concept of a cursor. However, unlike embedded SQL or module language, a *cursor specification* can be executed just like any other SQL statement – by *Prepare()/Execute()* or *ExecDirect()*. A cursor is automatically opened by the CLI Implementation. The function *Fetch()* moves the cursor in the forward direction – each call to *Fetch()* moves the cursor to the next row in the result set.

By default, cursors are forward-only. That is, they can move only in the forward direction. The application can set the statement attribute SQL_ATTR_CURSOR_SCROLLABLE to SQL_SCROLLABLE, to enable *scrollable cursors*. This enables the application to move the cursor both forwards and backwards. In SQL-92, scrollable cursors are required to be supported at the Intermediate SQL level [1]. The function

FetchScroll() is used to move the cursor in any direction. This function takes as arguments the statement handle, a flag indicating the direction to move, and an offset. The flag and offset arguments together determine the row on which the cursor is positioned. For example, if flag is SQL_FETCH_RELATIVE and offset is 10, then the cursor is moved 10 rows forward.

By associating application memory areas with result set columns (known as *column binding*), the application can also retrieve a row into its buffers when it calls *Fetch()* or *FetchScroll()*. We will examine this, and other ways of retrieving result set columns, in the next section.

3.8 CLI Descriptor Areas

SQL/CLI exposes functions for accessing the descriptor areas defined in Dynamic SQL. These descriptor areas are used to describe dynamic parameters and result columns. Descriptors are accessed through descriptor handles. A descriptor is describes one of the following:

1. A set of zero or more dynamic parameters. There are 2 types of parameter descriptors:
    - The *application parameter descriptor* or APD, which holds information about the application variables that supply dynamic parameter values.
    - The *implementation parameter descriptor* or IPD, which holds information about the same parameters after any specified data conversions.
2. A single row of result data. There are 2 types of row descriptors:
    - The *implementation row descriptor* or IRD, which contains a description of a row from the database.
    - The *application row descriptor* or ARD, which contains the row, after any data conversions that the application may specify.

A descriptor is a *conceptual table* with a header record and one or more detail records. The header record contains fields which describe the table as a whole, such as a COUNT field indicating the number of detail records available. For ARD and IRD, this is the number of columns in the result set. For APD and IPD, this is the number dynamic parameters.

Each detail record describes one result set column or dynamic parameter. There are many fields in a detail record, and we will not attempt to cover all of them here. The interested reader is referred to [2]. The TYPE field is an integer field which contains the data type code. In the IRD, this field indicates the data type of the column on the SQL-server. In the ARD, this indicates the data type of the application buffer that will hold this column. By specifying a different type in the ARD, the application can perform data conversion. Similarly, the TYPE field in IPD contains the data type of a dynamic parameter as it appears on the SQL-server. The application can specify a different TYPE field in APD to indicate that the CLI Implementation has to convert data before performing the execution.

Other fields in a detail record describe a dynamic parameter or a result set column in greater detail. For example, the LENGTH field in IRD or IPD indicates the maximum length, in characters, of a character dynamic parameter or result set column. The OCTET_LENGTH field in the ARD is set to the length of the buffer that is bound to a column.

The descriptor areas contain 3 *deferred fields* – DATA_PTR, INDICATOR_PTR and OCTET_LENGTH_PTR. When an application wants to bind to a result set column, it sets the DATA_PTR field in ARD to the address of a variable which is to hold the data. When *Fetch()* or *FetchScroll()* is called, the CLI Implementation populates this buffer with column data. Similarly, the application can set the DATA_PTR field of the APD to the address of a buffer which will hold the parameter data. Prior to calling *Execute()* or *ExecDirect()*, the application populates this buffer with the actual parameter data.

The application sets the INDICATOR_PTR field in the ARD or APD to the address of a variable which will contain *indicator information.* An indicator variable is used to transmit the SQL NULL value to and from host languages that may not have representations for NULL character or numeric values. The application sets the OCTET_LENGTH_PTR field in the ARD or APD to the address of a variable which will contain the actual number of octets for a result set column or dynamic parameter.

When a statement handle is allocated, the CLI Implementation automatically allocates 4 descriptors – the IRD, IPD, ARD and APD – and associates them with the statement handle. The application can also call *AllocHandle()* to explicitly allocate a descriptor. Explicitly allocated descriptors can be associated as the ARD or APD on a statement handle through *SetStmtAttr()*. The IRD and the IPD cannot be replaced. The *CopyDesc()* function copies fields from one descriptor to another.

3.9 Concise Functions

The operation of binding to a result set column using *SetDescField()* requires obtaining a descriptor handle and making multiple calls to set the individual descriptor fields. The function *BindCol()* specifies all of these values on a statement without obtaining the descriptor handle. Similarly *BindParam()* specifies multiple descriptor fields associated with parameter binding in a single call.

SetDescRec() and *GetDescRec()* set and get multiple descriptor fields in a single call, but require a descriptor handle.

Note that while these concise functions are a useful shorthand, they are not extensible. This is because, the fields that they can read are hard coded in their parameter lists. *GetDescField()* and *SetDescField()* however, are extensible.

3.10 Diagnostics

SQL/CLI has a rich diagnostic model to handle error conditions. Every function returns a return code of SQL_SUCCESS when it was executed successfully. If a function executes successfully, but some warnings are generated (for example, when *Fetch()* results in column data being truncated), the function returns with SQL_SUCCESS_WITH_INFO. When there is an error in executing the function, it returns SQL_ERROR.

Associated with each handle in SQL/CLI is a *diagnostics area.* This is a conceptual table that is equivalent to the diagnostics area used in embedded SQL. When a function that uses a given handle is called, the diagnostics area associated with this handle is cleared and new diagnostic information is written into the diagnostic area.

The diagnostics area has a header record and one or more detail records. The header record has several fields which return information such as the return code of the underlying function, the number of detail records, and the number of rows affected by an SQL Update or Delete function. The fields in the header record are available regardless of what the return code of the underlying function is.

When the underlying function returns a code of SQL_SUCCESS_WITH_INFO or SQL_ERROR, additional diagnostic information is available in one or more detail records of the associated diagnostics area. Each detail record contains several fields, including the 5 character SQLSTATE[3], MESSAGE_TEXT and the implementation dependent error or warning text.

The application can call *GetDiagField()* to read an individual field in the diagnostics area, or *GetDiagRec()* to read multiple fields in a single call.

3.11 Transaction Demarcation

When an application connects to an SQL-server and performs work, a transaction is implicitly started. The application commits or rolls back a transaction by calling the *EndTran()* function, which operates on a connection handle. It is implementation defined whether it is possible to have more than one connection, each of which has an active transaction.

4. Summary and Future Directions

SQL/CLI defines a programming interface that consists of function calls used to do database work. The handle based design of SQL/CLI means that there is no need for global data areas - the application gains access to all information through a handle. This binding style does not need a pre-processor and hence is useful for developing portable applications.

This binding style has gained wide industry acceptance. Microsoft has announced plans for a new version of ODBC to align with the SQL/CLI Standard. Several new enhancements are under active development for the next version of SQL/CLI, such as the efficient handling of large data through *locators,* and improved support in the CLI for *persistent stored modules,* which provide performance gains by avoiding recompilation of SQL code.

References

[1] IS 9075 *International Standard for Database Language SQL,* document ISO/IEC 9075:1992, equivalent to American National Standard ANSI X3.135-1992, J.Melton, Editor. October 1992

[2] IS 9075-3 *International Standard for Database Language SQL - Part 3: Call Level Interface,* document ISO/IEC 9075-3:1995, equivalent to American National Standard ANSI/ISO/IEC 9075-3:1995, J.Melton, Editor. Publication expected Fall 1995.

[3] SQLSTATE refers to 5 character codes that denote standard error/warning situations. They are defined in SQL-92.