White Paper

March 1994

FoxPro® Client-Server Architecture for Enterprise Database Connectivity

Client-Server application development using Microsoft® FoxPro and SQL Server.

Contents

Special thanks go to Yair Alan Griver of Flash Creative Management, Pat Adams of DB Unlimited, and Melissa Dunn of MicroEndeavors, Inc. for their valuable input and thorough review of the whitepaper.

Overview

The ability to access heterogeneous data that resides on different hardware platforms, different operating systems, different network operating systems, and different databases is a fundamental need for client-server computing. Client-server computing is beginning to move into the mainstream of corporate information systems. With this move comes the need for client-server applications that can access enterprise-wide data. Much of this data is currently stored in mainframe and mini-computer databases, and one of the challenges facing implementors of client-server technology today is how to bring this mission-critical data to the desktop and integrate it with the functional, easy-to-use graphical user interfaces (GUIs) that are associated with PC-based tools.

Each computer company, each corporation, and, in some cases, each individual user has their own definition of what client-server computing means. In order to eliminate confusion, when talking about the client-server architecture in this white paper it refers to distributing an application between a front-end client workstation component and a back-end server component. Ideally, the server has responsibilities for managing all the requests it receives from other processes, including request queue management, buffer management, execution of the service, results management, and notification of service completion. It is the client’s task to initiate communications, request specific services, acknowledge service completion notifications, and accept results from its server. User-intensive functions, such as handling input and displaying data, are left to the user’s FoxPro application. Data-intensive functions, such as file I/O and query processing, are left to the RDBMS.

In comparison, when an application running on a PC can transparently access data located on a file server, this is known as the file server architecture. Essentially, the PC application requests data from a shared file, the networking software automatically retrieves a block of the file from the server. However, in a scenario where FoxPro repeatedly requests blocks of data from the network server, heavy network traffic is produced.

Microsoft FoxPro and SQL Server provide the best of both worlds in client-server development. FoxPro has superior decision-support capabilities with its unsurpassed speed through the Rushmore technology. SQL Server provides a high level of security and data integrity at the database level for robust data entry systems. These products working together provide the capability to develop on-line transaction processing and decision support client-server applications.

The purpose of this white paper is to outline some of the basic issues involved in accessing heterogeneous databases, outline general approaches to achieving heterogeneous database access, and outline how FoxPro can access heterogeneous databases. The database connectivity solutions developed by Microsoft are discussed in depth, with an emphasis on how these products relate to each other. Finally, this white paper provides some general guidelines for designing applications for enterprise database connectivity using Microsoft FoxPro, SQL Server and Microsoft® database connectivity products.

Heterogeneous Database Access Issues

Think of accessing heterogeneous databases as a subset of using distributed databases. The technical challenges of delivering fully distributed database management systems (DBMS) in commercial products are difficult and have not yet been solved. These problems include distributed query processing, distributed transaction management, replication, location independence, as well as heterogeneous database access issues. The ability to access *heterogeneous databases* (that is, data that resides on different hardware platforms, different operating systems, different network operating systems, and different databases) is a fundamental need today, and it can be addressed without having to wait for fully distributed databases to arrive.

When thinking about the problems involved in accessing heterogeneous databases, it is useful to consider the problems at different levels. Figure 1 identifies some of the levels and interfaces encountered when accessing data in a client-server environment.

Figure 1

µ §

Levels and interfaces in a client-server environment

Some of the areas that need to be addressed when attempting to access heterogeneous databases are application programming interfaces (APIs), data stream protocols, interprocess communication (IPC) mechanisms, network protocols, system catalogs, and SQL syntax.

Application Programming Interfaces

Each back-end database typically has its own application programming interface (API), through which it communicates with clients. A client application that must access multiple back-end databases therefore requires the ability to transform requests and data transfers into the API interface supported by each back-end database it needs to access.

Client/server applications communicate with SQL Server for Windows NT™ through two application programming interfaces: Open Database Connectivity (ODBC) and DB-Library™.

ODBC is a C programming language interface for generic database connectivity. The ODBC interface permits maximum interoperability, allowing a single application to access diverse database management systems. The application developer can develop, compile, and ship an application without targeting a specific DBMS. ODBC achieves interoperability by forcing all clients to adhere to a standard interface. The ODBC driver will automatically interpret a command for a specific data source. ODBC has been designed to be a general purpose Call Level Interface (CLI) for any database backend, including non-relational DBMSs.

ODBC provides the following advantages:

- Microsoft Windows operating system universal data access: ODBC is the Microsoft strategic direction for access to relational databases from the Windows platform. New Windows-based client-server applications should use ODBC as their database access API. In the future, Microsoft will also support ODBC on the Macintosh® and other platforms.
- Flexible heterogeneous data access: ODBC was designed as an API for heterogeneous database access.
- ODBC preserves the semantics of the target DBMS data types.
- ODBC provides a connection model that is generic and extensible to allow for different networks, security systems, and DBMS options.
- Access to “local” data. ODBC enables easy access to local data such as Xbase or Microsoft Access®. It will treat local data that is not in a relational format as if it were a relational database. From a single FoxPro application you can access local and remote data through the same ODBC API.

DB-Library is an API designed specifically for Microsoft SQL Server or Sybase® SQL Server. DB-Library is a set of C functions and macros that allow an application to access and interact with SQL Server. DB-Library offers a full set of application programming interfaces (APIs) for: (1) opening SQL Server Connections, (2) formatting queries, (3) sending query batches to the server and retrieving the resulting data, (4) bulk-copying data from files or program variables to and from the server, (5) performing two-phase commit operations, and (6) executing stored procedures on remote servers.

Data Stream Protocols

Every DBMS uses a data stream protocol that enables the transfer of requests, data, status, error messages, etc. between the DBMS and its clients. Think of this as a “logical” protocol. The API uses interprocess communication (IPC) mechanisms supported by the operating system and network to package and transport this logical protocol. The Microsoft SQL Server data stream protocol is called Tabular Data Stream (TDS). Each database’s data stream protocol is typically a proprietary one that has been developed and optimized to work exclusively with that DBMS. This means that an application accessing multiple databases must have the ability to use multiple data stream protocols. Using ODBC helps resolve this problem for application developers.

Interprocess Communication Mechanisms

Depending on the operating system and network it is running on, different interprocess communication (IPC) mechanisms might be used to transfer requests and data between a DBMS and its clients. For example, Microsoft SQL Server on OS/2® uses named pipes as its IPC mechanism, Sybase SQL Server on UNIX® uses TCP/IP sockets, and Sybase on VMS® uses DECnet™ sockets. The choice of IPC mechanism is constrained by the operating system and network being used. In a heterogeneous environment, multiple IPC mechanisms may be involved.

SQL Server for Windows NT has the ability to communicate over multiple Interprocess Communication Mechanisms. SQL Server communicates on named pipes (over either Netbeui or TCP/IP network protocols) with clients running Windows, Windows NT, MS-DOS®, and OS/2 operating systems. It can also simultaneously support TCP/IP Sockets for communication with Macintosh, UNIX, or VMS clients and SPX sockets for communications in a Novell® Netware® environment. As the networking components for Banyan® VINES® become available for Windows NT, it will be supported as well.

Network Protocols

A network protocol is used to transport the data stream protocol over a network. It can be considered the “plumbing” that supports the IPC mechanisms used to implement the data stream protocol, as well as supporting basic network operations such as file transfers and print sharing. Popular network protocols include NetBEUI, TCP/IP, DECnet, and SPX/IPX.

Back-end databases can reside on a local-area network (LAN) that connects it with the client application, or it can reside at a remote site, connected via a wide-area network (WAN) and/or gateway. In both cases, it is possible that the network protocol(s) and/or physical network supported by the various back-end databases are different from that supported by the client or each other. In these cases, a client application must use different network protocols to communicate with various back-end databases.

System Catalogs

A relational database management system (RDBMS) uses system catalogs to hold information, or metadata, about the data being stored. Typically, system catalogs hold information about objects, permissions, data types, and so on. Each RDBMS product has an incompatible set of system catalogs with inconsistent table names and definitions. Many client tools and applications use system catalog information for displaying or processing data. For example, system catalog information can be used to offer a list of available tables, or to build forms based on the data types of the columns in a table. An application that makes specific reference to the SQL Server system catalog tables will not work with another RDBMS such as DB2® or Oracle®.

SQL Syntax and Semantics

Structured Query Language (SQL) is the standard way to communicate with relational databases. In a heterogeneous environment, two main problems arise with respect to SQL syntax and semantics. First, different database management systems can have different implementations of the same SQL functionality, both syntactically and semantically (for example, data retrieved by a SQL statement might be sorted using ASCII in one DBMS and EBCDIC in another; or the implementation of the UNION operator in different database management systems might yield different result sets). Second, each implementation of SQL has its own extensions and/or deficiencies with respect to the ANSI/ISO SQL standards. This includes support for different data types, referential integrity, stored procedures, and so on. An application that needs to access multiple back-end databases must implement a lowest common denominator of SQL, or it must determine what back-end it is connected to so that it can exploit the full functionality supported.

When developing client-server applications in a heterogeneous environment, it is important to first understand the different approaches to accessing databases. These database access approaches can be classified into three possible classes: the common interface approach, the common gateway approach, and the common protocol approach, as defined by R.D. Hackathorn in his article “Emerging Architecture for Database Connectivity” in *InfoDB.*

Common Interface Architecture

A common interface architecture, shown in Figure 2, focuses on providing a common API at the client side that enables access to multiple back-end databases. Client applications rely on the API to manage the heterogeneous data access issues discussed earlier. Typically, a common API would load back-end–specific drivers to obtain access to different databases. An example of a common interface architecture is Microsoft Open Database Connectivity (ODBC), discussed later in this technical note.

Figure 2

µ §

Common interface architecture

Common Gateway Architecture

A common gateway architecture, shown in Figure 3, relies on a gateway to manage the communication with multiple back-end databases.

An example of a common gateway architecture are gateways based on Microsoft Open Data Services, discussed later in this technical note.

Figure 3

µ §

Common gateway architecture

In his book *Introduction to Database Systems*, C.J. Date states: "... there are clearly significant problems involved in providing satisfactory gateways, especially if the target system is not relational. However, the potential payoff is dramatic, even if the solutions are less than perfect. We can therefore expect to see gateway technology become a major force in the marketplace over the next few years." (page 635)

Common Protocol Architecture

The common protocol approach, shown in Figure 4, focuses on a common data protocol between the client and server interfaces. Conceptually, this is perhaps the most elegant way of addressing the problem of heterogeneous data access.

Figure 4

µ §

Common protocol architecture

Two common data protocol architectures are the proposed ANSI/ISO Relational Data Access (RDA) standard, and the IBM® Distributed Relational Database Architecture (DRDA™). Both of these architectures are in their infancy, and it is too early to determine how well they will function as commercial products.

It is important to note that these approaches to enabling heterogeneous database access are not exclusive. For example, an ODBC driver might connect through an Open Data Services gateway to a back-end database. Alternatively, an ODBC driver or Open Data Services gateway that "speaks" DRDA or RDA is possible.

We have looked at the basic issues involved in accessing heterogeneous databases, and generalized ways of approaching solutions. We will now look at specific connectivity products from Microsoft that enable heterogeneous data access. The SQL Server building blocks to data access, Tabular Data Stream (TDS) and the Net-Library architecture, are an integral part of products enabling connectivity to heterogeneous databases. We then discuss Microsoft Open Database Connectivity (ODBC) and the FoxPro Connectivity Kit. Finally, we make recommendations to help you decide which API, DB-Library or ODBC, to use and identify considerations that you should be aware of when developing client-server applications.

SQL Server Building Blocks (TDS and Net-Library)

Tabular Data Stream (TDS) and Net-Library are part of the core SQL Server

Note§ *The SQL Server Driver for ODBC also uses Net-Libraries and the TDS protocol to communicate with SQL Server and Open Data Services.*

technology that Microsoft connectivity products build on to integrate SQL Server–based applications into heterogeneous environments. Figure 5 shows how TDS and Net-Library fit into the client-server architecture of SQL Server–based applications.

Figure 5

µ §

FoxPro and SQL Server building blocks

TDS is the data stream protocol used by Microsoft SQL Server, Open Data Services, and SYBASE software to transfer requests and responses between the client and the server. Because TDS can be considered a logical data stream protocol, it must be supported by a physical network interprocess communication mechanism (IPC) which is where the Net-Library architecture comes in. A DB-Library application makes calls to the generic Net-Library interface. Depending on which Net-Library is loaded, communication with SQL Server is achieved using named pipes, TCP/IP sockets, DECnet sockets, SPX, and so on.

The Net-Library architecture provides a method of sending TDS across a physical network connection, as well as a transparent interface to the DB-Library application programming interface (API) and the SQL Server driver for ODBC. Net-Libraries are linked in dynamically at runtime. With the Microsoft Windows NT, Windows, and OS/2 operating systems, Net-Libraries are implemented as dynamic-link libraries (DLLs), and multiple Net-Libraries can be loaded simultaneously. With the MS-DOS operating system, Net-Libraries are implemented as terminate-and-stay-resident (TSR) and only one can be loaded at any given time.

Microsoft Open Database Connectivity

Open Database Connectivity (ODBC) is a universal database connectivity API that enables applications to access data in a heterogeneous environment of relational and non-relational database management systems. Based on the SQL Access Group's Call Level Interface (CLI) specification, ODBC is an open, vendor-neutral way to access data stored in a wide range of proprietary databases. ODBC takes the "common API" approach, discussed earlier, to achieving heterogeneous data access.

The ODBC architecture consists of three components:

- **Application**. Calls ODBC functions to connect to a data source, send and receive data, and disconnect.
- **Driver Manager**. Provides information to an application such as a list of available data sources; loads drivers dynamically as they are needed; and provides argument and state transition checking.
- **Driver**. A DLL that processes ODBC function calls and manages all exchanges between an application and a specific DBMS. If necessary, the driver may translate the standard SQL syntax into the native SQL of the target data source. All translations are the

responsibility of the driver developer.

The Driver Manager and driver appear to an application as one unit that processes ODBC function calls. Applications are not limited to communication with one driver. A single application cam make multiple connections, each through a different driver, or multiple connections to similar sources through a single driver.

Figure 6 shows the components of the ODBC architecture.

Figure 6

µ §

The ODBC model

Each ODBC driver supports a set of core ODBC functions and data types and, optionally, one or more extended functions or data types, defined as extensions:

- Core functions and data types are based on the X/Open and SQL Access Group CLI specification. If a driver supports all core functions, it is said to conform to X/Open and SQL Access Group core functionality.
- Extended functions and data types support additional features, including date, time, and timestamp literals, scrollable cursors, and asynchronous execution of function calls. Extended functions might not be supported by a specific driver. Extended functions are divided into two conformance designations, Level 1 and Level 2, each of which is a superset of the core functions.

ODBC can be used in different configurations, depending on the database being accessed. It can be used in one-, two-, or three- tier implementations. Microsoft and SYBASE SQL Servers and Open Data Services ODBC drivers conform to the highest level of ODBC extended functionality (level 2), supporting scrollable cursors and asynchronous communication.

For additional information about ODBC, see the *Microsoft ODBC Application Developer’s Guide* and the *Microsoft ODBC Driver Developer’s Guide.* A complete list of ODBC drivers can be found in the *Microsoft ODBC Driver Catalog*. Microsoft FoxPro can connect to any of the ODBC drivers listed in the catalog.

Microsoft Open Data Services

Microsoft Open Data Services is a server-side development platform that provides application services to complement the client-side APIs discussed earlier. Open Data Services provides the foundation for multithreaded server applications to communicate with DB-Library or ODBC clients over the network. When the client application requests data, Open Data Services passes the request to user-defined routines, and then routes the reply back to the client application over the network. The reply looks to the client as if the data were coming from SQL Server. Figure 7 illustrates how Open Data Services integrates into an enterprise.

Figure 7

µ §

Open Data Services and an enterprise

Open Data Services allow you to extend your FoxPro applications to reach enterprise data. The most common use of Open Data Services is as a gateway to data sources which may not have ODBC drivers available. There is an ODBC driver for Open Data Services which would provide you with open connectivity to any data source. Your FoxPro application would connect to the Open Data Services ODBC driver as if you were connecting to a SQL Server or Oracle database. Two types of gateways are:

· General-purpose Gateways - that can handle any ad hoc SQL request from a DB-Library or ODBC client. The Database Gateway from Micro Decisionware, for example, implements a general-purpose gateway into DB2. A component that understands the SQL language and can act on SQL requests is essential to the operation of a general-purpose gateway. This SQL interpreter usually resides in the back-end database itself (as is the case with DB2), but it can also be implemented in the gateway.

· Custom Gateways - Not all data server applications need to understand and respond to SQL requests (for example, a data server application that returns the contents of a specific flat file as a results set). This type of application could be designed to respond to only one particular procedure call (such as **GetFileA**). The Open Data Services application would define the column names and the data types of the fields in the flat file, and then return the records in the file to the requesting client as rows of data. Because this results set would look exactly like a SQL Server results set, the client could process it. This approach works when the information required from the existing system is well-defined, not ad hoc in nature. For ad hoc queries, the better approach is to extract the data from the existing system and load it into a relational database.

This section gives general guidelines to follow when developing applications for enterprise database connectivity using Microsoft FoxPro, Microsoft SQL Server and the Microsoft database connectivity products discussed in the previous section.

FoxPro Connectivity Kit Architecture

Microsoft FoxPro provides immediate and direct access to heterogeneous data through the architecture of the FoxPro Connectivity Kit. The Microsoft FoxPro Connectivity Kit consists of a set of libraries and drivers that enable developers to build client-server applications using FoxPro for MS-DOS or FoxPro for Windows. The Connectivity Kit gives the developer the ability to query external databases, send and retrieve data, update external databases, administer external databases, and execute DBMS specific features such as stored procedures. Essentially, through the FPSQL libraries in the Connectivity Kit, there is a channel opened directly to the external data source. (See Figure 8 on the following page.) The SQL syntax is transmitted directly without being interpreted.

Applications created with FoxPro for MS-DOS can access SQL Server data through the FPSQL library included in the Connectivity Kit. The FPSQL library, FPSQL.PLB, is bound to DB-Library, the API which allows connectivity to Microsoft SQL Server and Sybase SQL Server.

In a Windows environment, the Connectivity Kit includes the FPSQL library that provides ODBC connectivity. Using this library, applications created with FoxPro for Windows can connect to any external database, provided that there is an ODBC driver for that particular DBMS. The ODBC drivers that ship with the Connectivity Kit are for Microsoft SQL Server and Oracle. For a comprehensive list of ODBC drivers that FoxPro can connect to, refer to the *Microsoft ODBC Driver Catalog*.

Figure 8

Microsoft FoxPro client-server architecture.

FoxPro 2.6 Professional Edition Connectivity Updates

Microsoft FoxPro 2.6 adds several new ease-of-use enhancements and dBASE compatibility extensions. In addition, programmability enhancements have been made to client-server connectivity. The Connectivity Kit is now included within FoxPro 2.6 Professional Edition.

Client-server connectivity enhancements include:

- A client-server wizard to assist developers in generating the code necessary to connect to ODBC data sources. (Further details are provided in the Processing/Managing Data section.)
- Improved support for handling NULL values from ODBC data sources. (Further details are provided in the Processing/Managing Data section.)
- Ability to return multiple error messages back to the FoxPro client application.

  The Connectivity Kit will now be able to return a list of up to 5 error messages per connection. They are displayed in the error message window and can be obtained by calling DBError(). DBError now has a fourth optional parameter with values from 1 to 5 that indicates which message is being requested from the list. If the parameter is not specified, the latest error message is returned.
- New DBVersion() function
  This function will send back the version number of the connectivity kit that is currently installed on the FoxPro client. No parameters are required to execute this function.
- Logical values return (T,F).
- Date fields from the ODBC data source will automatically be converted to character fields within FoxPro.

FoxPro Using the Gateway Approach

FoxPro for Windows applications are able to access a back-end database through a direct-connect (two-tier) ODBC driver loaded at the workstation, or by connecting to an Open Data Services–based gateway using ODBC (a three-tier solution). FoxPro for DOS applications will be able to access a back-end database through

DB-Library, or by connecting to an Open Data Services–based gateway via DB-Library.

Figure 9

µ §

FoxPro applications using DB-Library or ODBC can connect to SQL Server and Open Data Services.

SQL Syntax and Semantics

SQL is a widely accepted industry standard for data definition, data manipulation, data management, access protection, and transaction control. SQL originated from the concept of relational databases using tables, indexes, keys, rows, and columns to identify storage locations. SQL is different from the FoxPro language which is a record oriented data manipulation language. FoxPro excels in locating specific rows of data. SQL excels in locating records of data..

Microsoft FoxPro natively supports some standard SQL commands against FoxPro data only. The SQL commands supported by FoxPro are INSERT INTO, CREATE TABLE, and SELECT.

ODBC defines a core grammar level that corresponds to the X/Open and SQL Access Group CAE SQL draft specification. The Core SQL grammar provides the following:

· Minimum SQL grammar.
  - Þ Data Definition Language (DDL): CREATE TABLE and DROP TABLE
  - Þ Data Manipulation Language (DML): simple SELECT, INSERT, UPDATE, SEARCHED, and DELETE SEARCHED
  - Þ Expressions: simple (such as A>B+C)
  - Þ Data types: CHAR

· DDL: ALTER TABLE, CREATE INDEX, DROP INDEX, CREATE VIEW, DROP VIEW, GRANT, and REVOKE

· DML: full SELECT, positioned UPDATE, and positioned DELETE

· Expressions: subquery, set functions such as SUM and MIN

· Data types: VARCHAR, DECIMAL, NUMERIC, SMALLINT, INTEGER, REAL, FLOAT, DOUBLE PRECISION

Beyond the FoxPro SQL and the ODBC SQL implementation, you can use the SQL language specific to the external database server. The decision as to whether to use “generic” SQL that is common to all databases being accessed, or to “sense” the back-end being accessed and make use of SQL extensions such as stored procedures, depends on the type of application being developed. Using ODBC as the client API, you can rely on the ODBC driver to take care of some of the differences in SQL syntax and semantics.

The decision on the level of “generic” versus “specific” SQL to use depends, among other things, on:

· The set of features you want to access from your applications, including features that may not be available from all data sources.

· How much interoperability you want to provide.

· How much conditional code you want to include to determine whether a function or data type is supported by the data source.

· Performance requirements. In general, performance is increased through the use of specific back-end data source features, and it can be adversely affected if you use a minimum set of SQL common to all databases.

Using Microsoft FoxPro, a developer has the best of both worlds. SQL Server (or any ODBC database) can be utilized to locate sets of information and FoxPro's Data Manipulation Language can be used to manipulate the rows of the result sets from the SQL Server query.

When developing applications using Microsoft FoxPro the question comes up about when to use Microsoft FoxPro and when to move the data to SQL Server. It is important to understand when to use both and what the differences are. The following sections outline some of the advantages to storing data on SQL Server and how FoxPro can leverage those features.

Processing/Managing Data

Using SQL Server for data storage provides you with advantages such as built in data validation, referential integrity, user and data security, and the ability to store larger amounts of data than in FoxPro. SQL Server on the Windows NT platform can store and manage *terabytes* worth of data whereas FoxPro can handle 2 GB or a billion records in a single database, which ever comes first. On the other hand, FoxPro's use of the Rushmore™ technology to execute queries against such large quantities of data is very efficient and therefore can be faster that native database server queries. With this in mind, a developer may want to split an application up between SQL Server and FoxPro. For example, a decision support application that does not modify any data and involves complex queries which slow server processing, may be best suited to run under FoxPro. Using the speed of Rushmore, immediate decisions can be made due to the quick response time. The sensitive data or data that is being dynamically updated should be kept on the backend SQL Server.

When designing a FoxPro client-server application, the FPSQL function library requires that a connection handle be acquired in order to communicated with an external database. Once a connection has been established, this connection handle will be used for all future FPSQL function calls.

Establishing a connection is the same whether you are connecting to SQL Server, Oracle, or another ODBC database. Using FoxPro for DOS, the connection is always to a SQL Server database due to DB-Library. DBConnect() is used to establish the connection handle. In the event that your application requires connections to multiple data sources, you can open multiple connections. Each connection is a separate data stream. In order to establish a connection to a SQL Server called "foxsqlnt" you would type in the following:

```
handle = DBConnect("foxsqlnt","sa","","")
```

In the example above, the server name is "foxsqlnt", the user name is "sa", and there isn't any password.

Querying

In FoxPro, a query extracts information from your tables and places it into another, temporary table called a cursor. Using the FoxPro Connectivity Kit, FoxPro can query SQL Server data and place the result set into a FoxPro cursor or table. Once the data is retrieved, you can use FoxPro to browse, query, analyze, and report data stored on external data sources. The query that FoxPro uses to request data from SQL Server must be contained within a DBExec() command written in Transact-SQL which is the SQL implementation that SQL Server understands.

Using the ODBC API in FoxPro for Windows, ODBC extensions to SQL can be used through the DBExec() command. ODBC defines 4 different extensions to ODBC:

1. Date, time and timestamp
2. Scalar functions: numeric, string, and data type conversion functions
3. Outer joins
4. Procedures

The SQL statement syntax is based upon the SQL Access Group's standard escape clause to cover vendor specific extensions to SQL. The format is: "--*(vendor(vendor-name),product(product-name)SQL extension--*)".

The following examples create the same result set of upper case employee names. The first statement below uses the escape clause syntax for a scalar function. The second statement uses the native syntax for SQL Server.

```
select=--*(vendor(Microsoft),product(ODBC) fn UCASE(NAME)--*)
   from employee

select upper(NAME)
   from employee
```

Results that are returned to the Fox application can be returned in a cursor or table. The cursor is similar to any other table in FoxPro, however it is temporary and the data can not be modified unless a read/write cursor is created. Cursors can be fast to work with because they can be held in RAM memory. The cursor that the results are sent back to can be given a specific alias name or it will default to the alias name of *dbresult*. When you want to call the cursor in your application it is difficult to find out what name has been assigned to the cursor. If you want to modify the data then the result set should be put into a Fox table in which you assign a name to. Note that, by putting the result set into a table, it then becomes a FoxPro table or cursor which is not linked back to SQL Server. When you modify the table, it is the local table that is changed. When you want to modify the SQL Server data, you must send an UPDATE request to the server.

The following example modifies the *ytd_sales* column to reflect the most recent sales recorded in the *sales* table in a SQL Server database. This assumes that only one set of sales is recorded for a given title on a given date and that updates are current.

```
handle = DBConnect("foxsqlnt","sa","","")
```

```
IF handle<0
    WAIT WINDOW “Not Connected”
ELSE
    = DBExec (handle,"update titles;
        set ytd_sales = ytd_sales + qty;
        from titles, sales;
        where titles.title_id = sales.title_id;
        and sales.date in (select max(sales.date)from sales)")
ENDIF
```

Complex queries may take time to process, in which case, you may want to have more control of the type of processing that is done. This can be done using the DBSetOpt() function to specify the type of result set processing you want to have in the application. FPSQL functions default to synchronous processing unless otherwise specified. This means that the client will not have control of the application back until the processing of the result set is complete. The other option, asynchronous processing, is when the control is given back to the client application while the result set is still being processed. However, each function must be called in the application repeatedly until it returns a value other than 0. A 0 value indicates that the database server is still executing the query.

Here is an example of how to change from synchronous (Batch) processing to asynchronous (Batch) processing and display the result set in a browse window:

```
handle = DBConnect(“foxsqlnt”,”sa”,””,””)
IF handle>0
    WAIT WINDOW “Successfully connected”
    result = DBSetOpt(handle,”Browse”,”ON”)
    result = DBSetOpt(handle,”Asynchronous”,1)
    IF result>0
        retcode=0
        DO WHILE (retcode=0)
            retcode=DBExec(handle,”select * from bigtable”)
        ENDDO
    ENDIF
ELSE
    WAIT WINDOW “Unable to connect.”
ENDIF
```

In addition to the type of processing you can request, multiple result sets can be handled in a batch mode or a non-batch mode. Batch mode processing, which is the default setting, will only return results from the DBExec() call once all of the individual result sets have been received. When Non-Batch mode processing is selected, the first result set is returned by the DBExec() call. In order to receive the rest of the results, your FoxPro application must call DBMoreRes() continuously until there are no additional results available. No more results are available when a value of 2 is returned.

Transaction Processing

Transaction processing guarantees the consistency and recoverability of SQL Server databases. A transaction typically consists of several SQL commands that read and update the database, but isn’t actually executed until a commit command is issued.

By definition, transaction processing guarantees either that an entire transaction is completed and all resulting changes are reflected in the database or that the transaction is rolled back to a predetermined save-point without changing the database. Transactions can even span multiple servers.

Transaction processing assures that all transactions are performed as a single unit of work - even in the presence of a hardware problem or general system crash. For example, in the scenario below it is crucial that the user-defined transaction be processed in its entirety or not at all:

```
BEGIN TRANsaction X
    Debit savings account $1,000
    Credit checking account $1,000
COMMIT TRANsaction X
```

User-defined transactions are created by surrounding SQL data modification statements with BEGIN TRANsaction and COMMIT TRANsaction commands. Without these commands, SQL Server treats each SQL command it receives as a single transaction. Uncommitted transactions can be canceled by rolling them back, ROLLBACK TRANsaction.

The key component to transaction processing is the write-ahead transaction log that is maintained by SQL Server. This log ensures that data can be recovered. When a request is made to modify a database is received, a copy of both old and new states of the database's affected portions is recorded in the transaction log. These changes are always made *before* they are made to the database itself. At any point in time, SQL Server knows which transactions are in progress and which have been committed.

During recovery from a system failure, SQL Server uses the transaction log to restore that database to a consistent state by backing out incomplete transactions. The log is also used to ensure that all changes associated with committed transactions are fully reflected in the database.

Using the FoxPro Connectivity Kit, the need for using BEGIN TRANsaction and END TRANsaction has been simplified. You can specify how you want FPSQL to manage the transaction processing within the application. Using the DBSetOpt() command, the execution of DBExec() and DBMoreRes() functions is modified based upon the mode of processing you request. You can specify the following modes of processing:

· Auto: every SQL statement is considered a complete transaction that is automatically committed.

· Manual: for each SQL statement, if no transaction is open, the driver begins a transaction which will remain open until the application commits or rolls back a transaction using DBTransact().

In the example below, each DBExec() command will be embedded in a transaction.

```
handle = DBConnect("foxsqlnt","sa","","")
IF handle>0
   WAIT WINDOW "Successfully connected"
   result = DBSetOpt(handle,"Transact","A")
ENDIF
```

Support for NULL Fields

SQL Server has a facility built in that prevents NULLs from slipping into columns where they do not belong.   A NULL in SQL Server indicates that the user has not made an entry into a field. The value is unknown rather than blank or 0. Essentially, a NULL indicates that if the user does not make an entry at insert time and there is no default entry for this column, SQL Server assigns the value NULL.

FoxPro does not know the concept of a NULL value when passed from SQL Server and will show inconsistent results. Close equivalents in FoxPro are completely blank strings (""), completely blank dates ({_/_/_}), and zero in the case of a numeric field. However, there is a way to work around the issue of accepting NULL data.

When designing a FoxPro application that queries SQL Server tables, it is helpful to know whether or not the columns in the table allow NULLs. You can get this information by typing in the stored procedure *sp_help* from the FoxPro command window or ISQL/W, the query tool that ships with SQL Server.

For example, *sp_help authors* brings back the following information on the authors table:

```
Name                              Owner                         Type
---------------------------------------------------------------------
authors                           dbo                           user table

Data_located_on_segment                          When_created
-------------------------------------------------------
default                                          Jul 27 1993  9:54AM

Column_name        Type         Length     Nulls    Default_name     Rule_name
---------------------------------------------------------------------
au_id              id           11         0        (null)           (null)
au_lname           varchar      40         0        (null)           (null)
au_fname           varchar      20         0        (null)           (null)
phone              char         12         0        phonedflt        (null)
address            varchar      40         1        (null)           (null)
city               varchar      20         1        (null)           (null)
state              char         2          1        (null)           (null)
zip                char         5          1        (null)           ziprule
contract           bit          1          0        (null)           (null)
```

In the table above, there are four columns that will allow NULL values; address, city, state, and zip. The "1" in the NULL column indicates that NULLs are allowed. You will also notice that this same command lists the defaults and rules that are bound to a particular column.

There are several ways to handle NULL values from a backend data source once the columns have been identified that may contain NULL values. The recommended option is to use the new feature within FoxPro 2.6 Professional Edition that allows constants for NULL values from the external data source to be defined by a user.

Users can define four types of NULL values:

· "CharNull"	size 30
· "IntNull"	size 20
· "FloatNull"	size 20
· "DateNull"	size 23

The values can be set/get using the DBSetOpt() and DBGetOpt() functions:

```
handle = DBConnect("foxsqlnt","sa","","")
IF handle>0
    WAIT WINDOW "Successfully connected"
    result = DBSetOpt(handle,"FloatNull","9999999.9999999")
ENDIF
```

If a constant for NULL values is not specified, the default for all cases is an empty string, NULLs become spaces in FoxPro.

The second option is to use the ISNULL function within SQL Server to change the value. The ISNULL function will replace each NULL entry it finds with a value that you specify. For example:

This SELECT statement will not catch any NULL entries:

```
handle = DBConnect("foxsqlnt","sa","","")
IF handle>0
    WAIT WINDOW "Successfully connected"
    = DBExec (handle,"select au_lname, phone, state from authors")
```

```
ELSE
   WAIT WINDOW “Unable to Connect”
ENDIF
```

This SELECT statement will catch NULL entries for the state column and change the NULL value to an asterisk (*):

```
handle = DBConnect(“foxsqlnt”,”sa”,””,””)
IF handle>0
   WAIT WINDOW “Successfully connected”
   = DBExec (handle,"select au_lname, phone, isnull(state, *)")
ELSE
   WAIT WINDOW “Unable to Connect”
ENDIF
```

For columns with a character datatype, you may want to substitute an asterisk (*) or a word, such as “unknown”, for a NULL value. Numeric columns with NULLs could be replaced with a 0.

Client-Server Query Wizard

Microsoft FoxPro 2.6 Professional Edition includes powerful new wizards, designed to make everyday database tasks easier for users and developers. In order to simplify connectivity to heterogeneous databases, a Client-Server Wizard has been included. The Client-Server Wizard will generate the code necessary to connect to a remote database and execute a query against the data. Once the connection code has been generated it can be reused within other FoxPro applications.

The wizard stores the connectivity code in a program file with a .CSQ extension instead of a .PRG extension. Here is an example of the code generated:

```
m.passwd=""
WAIT WINDOW NOWAIT "Connecting..."
m.Handle=DBCONNECT("FOXSQLNT","sa",m.passwd)
WAIT CLEAR
m.RetVal=DBExec(m.Handle,"use pubs")
mFldLst='titles.title,titles.pub_id,titles.type,titles.price'
mFrom='titles'
mWhere=""
mOrderBy=' ORDER BY titles.title'
mGroupby=""
m.RetVal=DBExec(m.Handle,"SELECT "+mFldLst+" FROM ;
   "+mFrom+mWhere+mGroupBy+mOrderBy,"Result")
```

Each of the variables represents a step of the wizard. For example, m.Handle represents Step 1 that asks which ODBC data source do you want to connect to.

Data Integrity

Relational databases organize data in a simple, tabular form, and provide many advantages over other databases. One of the key advantages is the ability of an external database to automatically maintain integrity between entities. This data integrity is initially set up by the database administrator, the application developer does not have to worry about programming this into the database application. FoxPro does not have the ability to enforce data integrity.

It is possible however to provide some forms of data integrity in FoxPro. However, this is done programmatically, it can be changed at any time, and the integrity relationships must be put into each FoxPro application.

When designing applications that access company data, it is important that the data is protected and consistent. Incorporating data integrity rules and business policies with your data will ensure that the data does not become corrupt or disorganized.

FoxPro can maintain the integrity of the data programmatically. SQL Server enforces data integrity within the database itself, guaranteeing that complex business policies will be followed by all client-server applications. By storing the data on SQL Server, all of your FoxPro applications can take advantage of advanced data integrity features such as user-defined data types, defaults, rules, stored procedures, and triggers.

User-Defined Data Types

SQL Server provides an extensive list of pre-defined system datatypes for developers to use, such as: char, int, varchar, etc. In addition, FoxPro developers have the ability to create their own datatypes to supplement the system datatypes.

For example, a *state_code* datatype could be defined as two characters (char[2]). The Transact-SQL code is:

```
sp_addtype state_code, 'char(2)'
```

User-defined datatypes can be created within FoxPro using a VALID or WHEN clause in an application. These datatypes can be shared among multiple applications by copying the code from application to application. The user-defined datatypes in SQL Server are created once, stored in one central location, and can be shared throughout the database. The integrity of your data is consistent with minimal programming. Another advantage of user-defined datatypes is that rules and defaults can be bound to them for use in multiple tables, and tailored to specific applications.

Defaults

Defaults allows that application develop to specify a value that SQL Server inserts if no explicit field value is entered into a particular column. For example, the current date could be set as a default value for a *purchase_date* field in a customer purchase record. If a user or the FoxPro application doesn't make an entry in the *purchase_date* field, SQL Server will automatically insert the current date.

Rules

In FoxPro you can create a PICTURE clause to specify how fields, memory variables, and arrays are edited and displayed. Rules are similar to the PICTURE clause in an @...GET command. However, rules also provide integrity constraints that go beyond the column datatype parameters to enforce business policies. Whenever values are entered into a database SQL Server checks it against any rule that has been bound to that column. A rule can require that a value must match a particular pattern, match one of the entries in a specified list, or fall within a particular range.

Triggers

Triggers enforce referential integrity at the table and view level to supply cascading deletions and to supply cascading updates. In essence, triggers are a special type of stored procedure that are explicitly called for execution and triggers are automatically invoked by SQL Server whenever an attempt is made to modify the data that they protect. Triggers are invoked when an INSERT, UPDATE, or DELETE action is called.

Triggers can be nested 16 levels deep for a cascading integrity check of the database tables. If a trigger changes a table on which there is another trigger, the second trigger activates and can then call a third trigger, and so on.

The FoxPro VALID and WHEN clauses are similar to triggers. VALID and WHEN clauses are used for data validation but can be used in different ways throughout an application which could lead to gaps in your code. These clauses also can be very precise for a particular field, allowing modification of the data one field at a time. For integrity checks among tables, this get very complex and affect productivity.

Although integrity checks may be faster within FoxPro, you will have to copy common code to each application. By keeping integrity checks within the external database, the integrity rules and triggers are maintained with the data in one place. When changes are made to your corporation's integrity rules, they are made at the server in one place. The FoxPro client application will not have to be modified. It is best, when accessing SQL Server data from Fox, to leave the integrity of the SQL Server tables up to that database engine and use the VALID and WHEN clauses for field validation within the FoxPro client applications.

Stored Procedures

Every time a SQL command is sent to SQL Server for processing, the server must first parse the command, check to make sure that the syntax is correct, check to see if the client has the correct permissions necessary to execute the command, and create a query execution plan to process the request. For complex queries this process can take some time to process. Stored procedures ensure consistent access to data resources and increase the speed of query execution. Essentially, stored procedures are groups of compiled SQL statements that are stored on SQL Server for later recall.

Stored procedures have already been through the parser, the query optimizer, and pre-compiled in the procedure cache waiting to be executed. (See Figure X below.) SQL Server stores this compiled version in cache and uses it to process subsequent calls. As a result of being precompiled, stored procedures will dramatically increase the execution speed of your query. Stored procedures will also reduce your network traffic because you are sending a small data stream.

Figure 10

µ §

> Stored procedures will be recompiled for efficiency whenever changes are made to objects that they affect. In addition, stored procedures will accept parameters, so a single procedure could be used by multiple applications using different input data.

FoxPro does not have an equivalent function to a stored procedure. However, FoxPro developers can code complex queries and transactions into stored procedures and then invoke them **directly** from any FoxPro application, whether that be in Windows or MS-DOS. In addition, using stored procedures on the server side can reduce the amount of Transact-SQL code required in the client application.

For example:

The query below finds all *au_id*s in the *titleauthor* table for authors who make less than 50 percent of the royalty on any one book, and then selects from the *authors* table all author names with the *au_id*s that match the results from the *titleauthor* query. The following FoxPro statement will have to be added to your application:

```
handle = DBConnect(“foxsqlnt”,”sa”,””,””)
IF handle>0
   WAIT WINDOW “Successfully connected”
   = DBExec (handle,"select au_lname, au_fname from authors;
      where au_id in (select au_id from titleauthor;
      where royaltyper < 50")
ELSE
   WAIT WINDOW “Unable to Connect”)
ENDIF
```

To simplify the FoxPro code within your application, create a stored procedure on SQL Server called *get_authinfo* that executes a query to retrieve the same data from the customers and orders tables:

```
create proc get_authinfo as

select au_lname, au_fname
      from authors
      where au_id in
         (select au_id
         from titleauthor
         where royaltyper < 50)
```

When using the stored procedure, the FoxPro code will be the following:

```
handle = DBConnect(“foxsqlnt”,”sa”,””,””)
IF handle>0
   WAIT WINDOW “Successfully connected”
   = DBExec (handle,"exec get_authinfo")
ELSE
   WAIT WINDOW “Unable to Connect”)
ENDIF
```

Extended stored procedures

SQL Server provides a way to dynamically load and execute a function within a DLL in a manner identical to a stored procedure. Actions external to SQL Server can be easily triggered and external information returned. Both return status codes and output parameters identical to their counterparts in regular stored procedures are also supported.

Examples of extended stored procedures are supplied with SQL Server. One is *xp_cmdshell*. This function allows any Windows NT command or process to be executed from within SQL Server. For example, you can use xp_cmdshell from within a trigger to send a broadcast on the network about changes that have been made to the data. Another example would be a trigger that could test to see if the inventory has fallen below a certain level, automatically execute a reorder transaction, and send a Microsoft Mail message to the purchasing manager via MAPI.

Data Security

SQL Server implements comprehensive user-level security protections on database objects (tables, records, views, and so forth) and SQL commands. It also supports column-level security, where access to particular columns in a database can be restricted to certain users. Stored procedures can also be used to permit certain users to execute specific operations, without giving them permissions to access the underlying data.

All security information and logic is stored in the data dictionary, where it can be accessed and updated by the system administrator. Since all security is handled by SQL Server, FoxPro client-server applications can safely ignore these issues. This

Summary

Additional Information

References

security scheme is in addition to that imposed by Microsoft Windows NT™ Advanced Server.

Views

In SQL Server, views allow users to see and modify a subset of information contained within an existing database table or tables. For example, a company employee database might contain name, department, supervisor, performance rating, and salary columns (fields). A view of the table may contain only the name, department, and supervisor columns. Employees who do not need access to salary information can be given access to this predefined view rather than the actual table which contains sensitive information.

Views are created dynamically and behave like other tables - they can be displayed and operations performed on them. When data seen through a view is modified, the data in the underlying table(s) is modified as well. Conversely, changes to data in the underlying table(s) are automatically reflected in the views derived from them.

Users can only query and modify the data they see. The rest of the database is not accessible. By defining different views and selectively granting permissions on them, a user, or any combination of users, can be restricted to different subsets of the data.

This white paper has addressed some of the issues involved in enabling FoxPro client-server applications to access enterprise data stored in a wide variety of heterogeneous databases. FoxPro has powerful database functionality within, but, combined with the security, transaction processing, and data integrity of SQL Server, you can develop powerful client-server applications that access and maintain mission critical data.

Date, C.J. *An Introduction to Database Systems,* Volume 1 (5th edition). Addison-Wesley, 1990.

Hackathorne, R.D. **“**Emerging Architectures for Database Connectivity.” *InfoDB*, January 1991.

To receive more information about Microsoft FoxPro or Microsoft SQL Server§, contact Microsoft Inside Sales, Systems Software, at 1-800-227-4679.

FoxPro Connectivity Kit User’s Guide

FoxPro Goes Client-Server: DB-Library programming techniques for client applications
part number 098-30194

Microsoft SQL Server Transact-SQL Reference manual

Microsoft Open Data Services: Application source book
part number 098-32078

Discussion of the ANSI SQL Standard and Microsoft SQL Server
part number 098-34656