Data access within the Information Warehouse framework

by J. P. Singleton
M. M. Schwartz

IBM's Information Warehouse™ framework provides a basis for satisfying enterprise requirements for effective use of business data resources. It includes an architecture that defines the structure and interfaces for integrated solutions and includes products and services that can be used to create solutions. This paper uses the Information Warehouse architecture as a context to describe software components that can be used for direct access to formatted business data in a heterogeneous systems environment. Concepts of independence between software components and how this independence can provide flexibility for change are discussed. The integration of software from multiple vendors to create effective solutions is a key emphasis of this paper.

The Information Warehouse* framework from IBM includes an architecture, software products, and consulting services to create software systems that allow organizations to locate, access, copy, and manage their data, even in today's complex, heterogeneous systems environments. Most companies have an abundance of data to support their business processes and yet they struggle to make effective use of the data. Diverse computer hardware and software systems and distributed networks are often used to satisfy the information technology requirements of a company, and this diversity can add to the difficulty that the overall organization has in making effective use of data. Even understanding the scope of the data resources of a company can be challenging. There are usually multiple copies of the same data for various reasons—sometimes to make operational data more usable for end users, and sometimes to place data for better data access performance. Many times individuals have copies unbeknown to administrators. In some cases, the same data are represented differently in copies because of different application conventions.

Virtually everyone in the organization who uses data can be affected by this complexity. Data administrators have difficulty knowing what data exist, where copies of data exist, and whether copies are current and consistent. Systems administrators have the complex job of managing the installation and support of multiple software packages to support program access to all the data. Application builders often have to write multiple programs, each having unique requests, in order to access all of the required data. End users have difficulty knowing what data are best suited for their purpose and how to locate and access the data. Often end users are presented with data descriptions that have an administrator's view of the data rather than a business view that employs terms they understand. All these factors make it difficult for an organization to make effective use of its data.

Information Warehouse Architecture I[1] defines a structure, components, and programming interfaces that can be used to build software systems so that organizations can locate, access, copy, and manage their data in an integrated and flexible manner. The architecture is published for the use of customers and as a guideline for the integration of IBM products and the products of other software vendors. This initial architecture is focused on the requirements of business professionals (and applications) accessing formatted business data for informational or decision support use. (Formatted data are sometimes called structured data and can be contrasted with text, image, and video data.)

In this paper, we use the Information Warehouse architecture as a context to describe logical software components that can comprise *data access solutions*—a name that we use to refer to software that provides direct access to data in a heterogeneous systems environment. We discuss data access solutions with reference to how they satisfy the application builders' requirements for data access. We also consider aspects related to system administration. A distributed data environment is assumed, and the terminology of client/server computing is used.

We use the term "data access" to refer to accessing (i.e., reading or updating) data at the location where the data are stored. It is generally a synchronous function, that is, after making a data access request, an application waits until a response is received for the request. Data copy or replication is the copying or staging of data to another store for subsequent access. Data copy is frequently an asynchronous process. Data access and data copy are complementary functions. Information Warehouse Architecture I defines a structure and interfaces to support both.

Application builders' requirements

The application builders' point of view is of particular interest because it is here that "the rubber meets the road" when it comes to satisfying the end users' requirements for consistent and complete access to data. End users want the applications they use to provide access to all interesting data regardless of the type of data, the location of the data, or the data management software used. Applications have the job of satisfying this requirement, but in reality the requirement is "pushed down" to a lower level and must be satisfied by the software that provides data access services to the application—the data access solutions.

Many data access solutions are available today. Virtually all data management vendors provide a data access solution for access to their data, and perhaps to other data management systems. Still other software vendors specialize in providing data access to a variety of data management systems even though they do not provide data management systems themselves. We can safely say that for any data source there is a way to access the data. That is the good news. The bad news is the problem of complexity that application builders and systems administrators must deal with because of the many data access solutions and the rapid change in data access technology. The challenge is to find methods to simplify this complexity in a way that provides flexibility to adapt to changes in technology and changes in the enterprise.

A single consistent interface can lessen the dependence of the application on specific data management software.

The problem, seen in more detail from an application builder's point of view, is that an application must often use multiple data access solutions, each having a different interface, to reach the required data. The multiple solutions introduce complexity and added costs to developing, testing, and installing applications. Their data access requirements call for a single, general, and consistent interface for accessing all types of data. Furthermore, this interface must be available to all the client systems in which the application runs. These requirements are the broadest for commercial application vendors since their products must operate in all of the prevalent client systems and access all prevalent data sources, not

just those of a single enterprise. These vendors can best realize a return on their investment by reducing the development effort with a single, consistent data access interface and by increasing the number of data sources that the application can reach.

In addition to the benefits of efficiency and cost, a single consistent interface can lessen the dependence of the application on specific data management software by enabling the application to be separated or isolated from the data management software. Having a degree of independence between the applications and data management software allows the enterprise more flexibility in choosing or changing the data management systems that it uses. Care must be taken to ensure that dependence on data management software is not traded for dependence on the consistent interface. Here standards can play an important role. When the consistent interface complies with formal standards, additional independence is provided since the application may be able to run with multiple product offerings that implement the interface. In contrast, use of a proprietary interface results in the application (and the enterprise) being "locked in" to the vendor whose software provides the interface.

Software standard specifications attempt to introduce consistency and homogeneity into the heterogeneous world of software. There are factors that can prevent them from being effective, however. For one, standards groups define specifications by consensus among the participating members. Often, divergent positions are represented. Because of this situation, the definition of a standard can be a lengthy and political process. The development of a standard can thus be outpaced by advances in technology. Second, software vendors naturally attempt to differentiate their products to show "value add" for their customers. This differentiation results in changes and extensions when compared to a standards definition. Although the changes and additions may provide added value, customer use of them results in being "locked in" as mentioned above. This condition benefits the vendors but not always the customers. The benefits of advanced function and "value added" extensions must be weighed against the benefits of efficiency and independence that standards provide. Customers who feel strongly that compliance to standards is necessary must make this compliance a clear requirement for software vendors.

We discuss more about standards as we describe logical components that can comprise data access solutions. We also consider how each component can help to meet the requirements of application builders and how the different components and approaches simplify or complicate systems administration for the enterprise. Finally, we look at how the overall structure or architecture of a data access solution can provide the benefits of openness, independence, and flexibility for the enterprise.

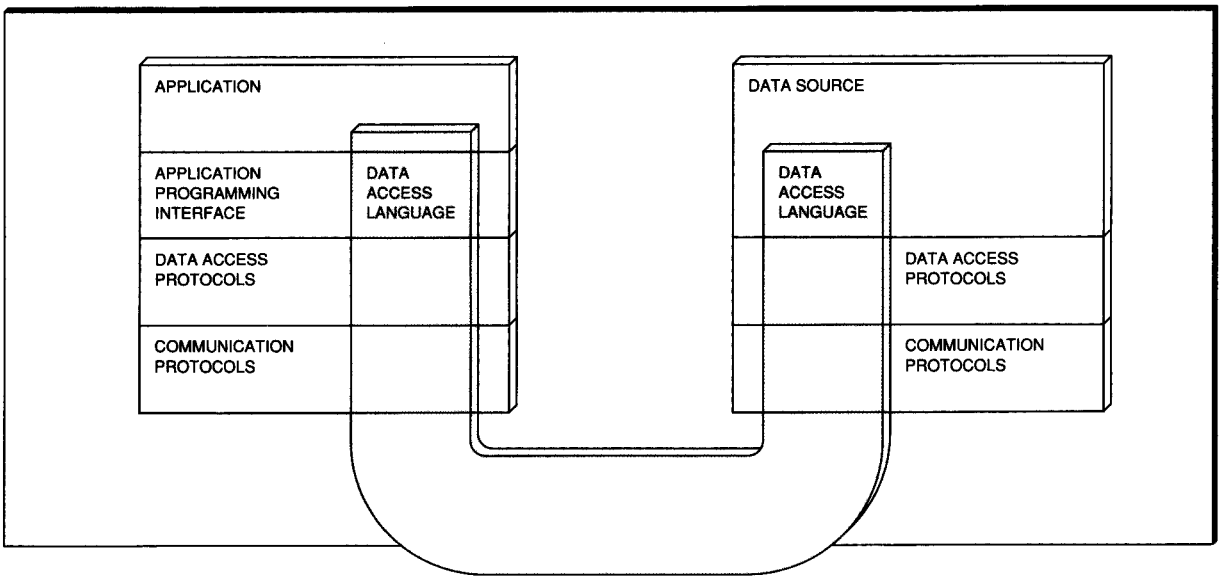
Components of a data access solution

The software components that we describe provide data access as a service for applications and tools, and are typically developed by data access vendors or data management system vendors. (We use the term *data access vendors* to refer to vendors that provide software for accessing data but do not provide data management system software.) Because the components are described from a logical view, the individual components identified do not necessarily exist as discrete product offerings. Generally, two or more components are combined in a product. The components we describe can be thought of as capabilities of a data access solution.

We start with the components that are "seen" by the application program. These components are, first, the *data access language* that is used to express requests passed to the data management software and, second, the *application programming interface* (API) that is used to incorporate these requests into the application program. The relationship of these two components can be seen in Figure 1. Although application builders benefit from all components in the data access solution, these first two should be the only components with which they directly interact.

Since we are assuming a distributed environment, data requests will frequently involve communication with remote data management software. We discuss this communication next in two contexts. First discussed are the *data access protocols* that define the content and meaning for the communications between data access clients and servers[2] on different systems. Then, we discuss how these data access protocols can be affected

Figure 1 Data access components



by *communication protocols*, the protocols that define how the data access communications are exchanged between nodes in a computer network. The four components that we have identified thus far define the basic flow of data access as shown in the figure. We describe them in detail and then describe two key additional components for solving the application builders' requirements.

In the area of data access, Information Warehouse Architecture I identifies interfaces for the data access language and the application programming interface. It also identifies a data access protocol. These interfaces are critical for integration—the interface between the application and the data access solution and the interface between the client and server systems. Although the architecture clearly indicates that data access should be provided through a single consistent interface, the architecture does not describe or mandate the "mechanics" of how to achieve it. Current software technology is very diverse in this area.

We discuss the interfaces identified by the architecture for data access in more detail later in this paper. In addition, we go beyond the architecture to describe some of the current software technology that might be used to provide these interfaces. In the course of this discussion, we introduce the possibility of additional software interfaces. We discuss these interfaces in the same context as those identified in the architecture. They provide additional opportunities for openness, independence, and flexibility in the structure of an overall data access solution.

In this paper, we use *data management system* as a term inclusive of database management systems and file systems. As we look at the components of a data access solution, we do not focus on data management software. We assume that the data management systems are a "given," that is, the data management systems that an enterprise owns are part of the definition of its requirement for data access. Access must be provided to the data with minimal impact on the function of the data management systems. We focus on how to reach the data wherever the data exist. The data to be accessed may be data that are used in the day-to-day operations of the enterprise or the data may be a copy of data made for informational use.

The term "middleware" has been in popular use in the computer software community for some time now. Data access middleware is software that provides an application with a consistent in-

terface to underlying (and often remote) services, insulating the application from the native interfaces and complexities required to execute the services directly. In this paper, all of the data access components could be considered to be part of middleware when, grouped as a data access solution, they provide a single consistent interface for access to heterogeneous data sources. Later in the paper we identify two components that provide key functions most strongly associated with data access middleware solutions.

SQL is currently the most popular and universal language for relational database access.

Data access language. For our purposes here, a data access language means the statements that are used to express what the application intends to do at the data management system. Examples are operations such as "fetch a record," "insert a record," and "create a table."

Many data access languages are available today. Examples include dBASE**, DL/I*, QUEL, and SQL (Structured Query Language). Some of these languages are designed to control navigation through the linked records of a hierarchical or network database. Others are designed to specify the desired result of a database operation, without specifying how the result is to be accomplished. There is a fundamental difference between these two classes of database language. One can be thought of as process-oriented, the other as "set-" or result-oriented. The process-oriented language allows the expression of detailed operations such as "get unique record," "get next record," and "get next record within parent and hold for update." The set-oriented language allows the expression of higher-level operations such as "select all unique rows where the value in the second column is greater than 2400."

Of the many available database languages, SQL is currently the most popular and universal language for relational database access. It is supported by nearly all of the major relational data management vendors. As its popularity has increased, even nonrelational data management vendors have adapted their products to accept it. As its use has become more and more prevalent, SQL has proven itself to be a very powerful, expressive query language, irrespective of the data store.

SQL was designed to be used with relational databases, i.e., where data are conceptually organized in tables and one table is related to another table by data values, not by linked records. It provides the capability to retrieve data, and to insert, update, and delete data as well. Elements of the language deal specifically with relational constructs such as tables, rows, and columns, and with operations such as the join and union of multiple sets of data. SQL is a set-oriented data language. An SQL statement expresses *what* is to be accomplished, not *how*. An application has no need to navigate through a database searching for records. Instead, the application describes *what* is to be retrieved (or changed), and the details of *how* are left to the database manager.

SQL offers a big advantage to the application that needs access to data in a heterogeneous environment. Because SQL can be used for both relational and nonrelational data access, and because it is supported in many of the leading database products, it is now possible to use a single data access language in conjunction with multiple different data management systems. SQL also has advantages in the client/server environment. Detailed database operations do not show through in the language, so implementation differences can be masked. The set orientation of SQL is a big advantage over distributed file access, as much more processing is possible at the remote site on behalf of one SQL statement. It also supports distributed data management systems more efficiently than navigational data languages because repeated function calls are not needed to navigate through a database.

The success of SQL as a *de facto* industry standard for database has prompted action by the recognized standards bodies ISO (International Organization for Standardization) and ANSI (American National Standards Institute), and the industry consortium X/Open, to write formal definitions[3] for the language. These groups have, for the most

part, cooperated with each other. The result is that a very significant amount of SQL has become standard in the industry, and portability of applications between different data management systems is becoming more and more achievable. The current definition of SQL from ISO and ANSI is popularly called *SQL92*. This definition is given with three levels of conformance: entry, intermediate, and full. In 1994, SQL implementations will be expected to conform to the entry level. Conformance to the remaining levels will be expected at some (as yet unspecified) time in the future.

Although SQL standards exist, they have, until very recently, lagged behind the full capability of implementing products. Database vendors, wanting to differentiate their products in the marketplace, have augmented them with extensions beyond the standard. The extensions are rarely compatible with any other vendor, so, even though there is a standard core to the language, it is very unlikely that the complete SQL set from any two vendors would be the same.

The differences between these sets of statements (or dialects) range from the support of completely unique SQL statements, to additional clauses on otherwise standard SQL statements, to just the minor inclusion of an extra keyword, and of course, might include unique semantics on a statement because of differences in default values or actions. Using nonstandard extensions in application development results in an application that is limited to a specific data management system, or to one that must test for a capability before using it—an added complexity, especially if testing for the capability must be done in a unique way. Thus, application developers interested in access to heterogeneous data management systems are advised to stay within the standard SQL.

There are other differences to consider beyond the language itself. For instance, the way in which data and status are presented to the application can vary. The basic interface between an application and an SQL-supporting data management system is one in which an SQL statement (or something representing the statement) is passed from the application to the data manager and the results of the SQL statement execution are passed back to the application in the form of data values, data descriptors, and status information. The way in which data and status information are passed between the application and the data manager varies considerably. A control block or descriptor is normally used, but the format of such a descriptor is not yet standardized. Formal standardization of status codes, data types, and data descriptors such as those defined in SQL92 will soon help in this area when data management systems implementing these features are released.

Until very recently, SQL standards have lagged behind the full capability of implementing products.

A final point on nonstandardization in SQL concerns the format and content of schema information. Most SQL implementations store schema information in "database catalogs" which appear to the user as normal tables accessible through SQL. These catalogs contain information about the tables, columns, data types, user authorities, and so on in the data management systems. It is probably safe to say that each of the vendor catalogs is different, at least in some respects. If the application needs information about the objects in a data management system, chances are that the application will have to ask each data manager for the information in a different way, by a different name, or will see results in different formats. Schema information covering tables, columns, and views is specified in the SQL92 intermediate level, but these tables are only a start for standardizing all of the information that is needed.

SQL is the data access language of choice for the Information Warehouse framework. The language has been chosen, but one of the many dialects must also be chosen. The SQL dialect identified in Information Warehouse Architecture I is specified in the ISO-ANSI SQL92 entry level standard. Using a standard dialect of SQL is very important if the enterprise needs access to multiple data management systems. The use of a nonstandard language tends to lock the application into a single vendor, thus reducing flexibility in choice of data management systems, and could

result in increased development and maintenance costs when the application must be changed to support a different data management system. The

The program preparation process has not been standardized.

trade-off is that the advanced functionality offered by a specific vendor will not be available to the application.

The best chance to achieve wide portability and connectivity for an application and to increase independence from any particular database management system (DBMS) is obtained by staying within the set of SQL statements specified by the Information Warehouse architecture (SQL92 entry level).

The implementation differences in control blocks and schema information can be circumvented by using a standard application programming interface discussed next.

Application programming interface. Technically, SQL is a data sublanguage with respect to an application programming language such as C or COBOL. SQL92 does not yet contain the logic constructs (for example, IF-THEN-ELSE, DO WHILE) that are provided by a procedural programming language. In contrast, the popular programming languages in use today do not contain anything close to the expressive power of SQL for manipulating databases. Thus SQL does not yet replace, but is complementary to, most programming languages at the functional level. For an application to gain the benefit of both a high-level programming language and SQL, a process is needed to combine the statements from the two languages in the appropriate logical sequence. The process used is an attribute of the application programming interface.

Information Warehouse Architecture I identifies two application programming interfaces for access to data: embedded SQL and callable SQL.

Embedded SQL. In embedded SQL, the SQL statements are interspersed directly into the application program in sequence with the procedural language statements. This style makes the program logic more straightforward for the programmer but requires additional source program processing. The programming language typically does not understand SQL statements, so something must be done to make the SQL statements acceptable to the programming language compiler. Embedded SQL is typically converted by a language preprocessor. The preprocessor converts SQL statements into a series of assignment statements and procedure calls that are compatible with the programming language and are then compiled along with the rest of the application program statements.

Program preparation is the process whereby SQL statements are preprocessed, the application program is compiled, and program variables and parameters are bound to the target data management system. For some vendors, this process is accomplished by a preprocessing utility. For others, there is no preprocessing step. Instead, the SQL statements are passed on to the data management system for interpretation.

The program preparation process has not been standardized. The differences in this process must be dealt with by the application builder, making it more difficult to write an application that is portable across database managers. This problem was not as big in the past when most application development was done in-house for the data management systems installed in an enterprise. One application generally ran with one data management system. The industry trend, however, is toward the use of applications and tools written by software development companies for general use. These application and tool builders have a different need. They are motivated to write their applications to run with as many data management systems as is practical. Application builders who want to ship "off-the-shelf" applications are forced to go through unique program preparation steps for each different data management system and ship multiple prepared modules, or they must provide relatively complex installation procedures and ask the customer to execute the program preparation steps during installation. The latter of these two choices may also force the application builder to

ship source code, making it more difficult to keep proprietary information out of competitors' hands.

Callable SQL. Callable SQL is an alternative API for using SQL with a data management system. In contrast to embedded SQL, the SQL statements are not embedded within the application as statements. Instead, the statements are passed as character strings through function calls to the data management system. These function calls provide the same capabilities as embedded SQL. Using a series of function calls, an application is able to submit an SQL statement for processing, retrieve the resulting data (if applicable), and inspect status information.

Callable SQL has no SQL program preparation process. Functions built into callable SQL replace the processing normally achieved during program preparation. Since the application code is not preprocessed, the application can be more independent of the data management system. As a result, it is a major advantage to be able to ship applications without including source code.

Builders interested in accessing multiple data management systems with their applications will probably find callable SQL the more flexible of the Information Warehouse API alternatives. Callable SQL functions allow the application to identify which data management systems are available in the run-time environment and some of the characteristics of each. Being able to determine these characteristics at run time means that the application builder can adapt the application to the capabilities of the data source. The information available includes data server names, server product identification, and SQL dialect conformance. From this information the application can infer the availability of specific functional support.

There are several callable interface implementations based on SQL in the marketplace today. Some of the more prominent products and interfaces include Q+E** Database Library, EDA/SQL**, Open Client**, Oracle Glue**, and ODBC. Each of these interfaces uses SQL to express the database operation, but each of them has its own different set of function calls and operational capabilities.

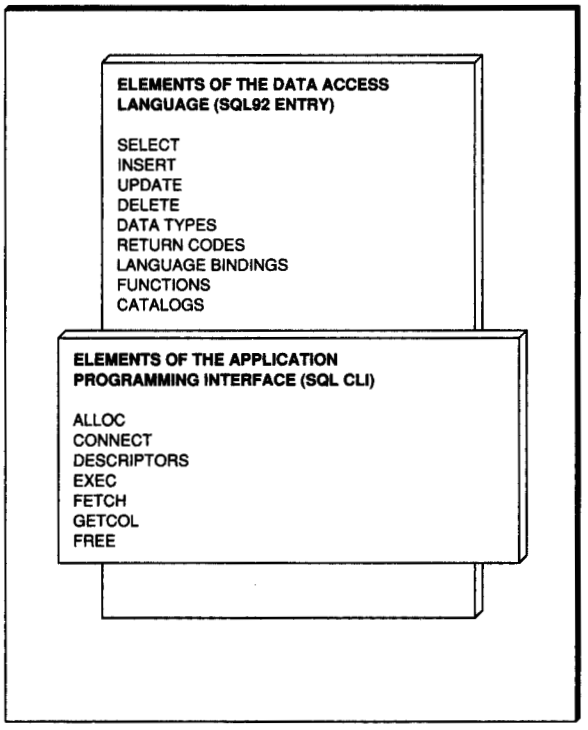
The most promising attempt at standardization for callable SQL is a definition originated by the SQL Access Group[4] that is being further refined and extended in cooperation with X/Open, ANSI, and ISO. This definition, the SQL Call Level Interface (SQL CLI), is expected to be the standard base for future implementations of callable SQL. Most of the leading data management vendors are supporting this work. The callable SQL interfaces that preceded this emerging standard must choose to remain different or to evolve to the standard. The same caveat that appears elsewhere in this paper applies here as well: Application builders will gain the most flexibility in choice of data management systems if the function calls used are those specified in the industry standard, not those of a specific vendor.

Two components to the data access API have been discussed here (Figure 2)—the function calls in callable SQL and SQL as the data access language that is passed through these function calls on to the data management system. These components are somewhat independent, but not entirely. The same function calls, for example, are used to prepare and execute any SQL statement, whether it be an UPDATE, CREATE, or SELECT. In some cases, however, a sequence of function calls might make sense only after execution of a particular SQL statement (a FETCH call would follow the execution of a SELECT but not a DELETE). At a lower level, there are dependencies between the function calls and SQL statements such as the compatibility of function call parameters with SQL data types and the matching of error codes with SQL operations.

The callable SQL identified in Information Warehouse Architecture I is the version defined by X/Open and adopted by ANSI and ISO—the SQL CLI.[5] Use of this API will increase the independence of the application from a particular interface provider.

The X/Open SQL CLI specification points to the X/Open SQL CAE (Common Applications Environment) as the definition of SQL to be used in conjunction with the SQL CLI. Other callable interfaces based on SQL also point to their respective SQL dialects. Note that it is possible for two implementations of the same CLI specification to support two different SQL dialects. Because the X/Open SQL CLI is, for the most part, just a carrier of SQL statement strings, the SQL dialect to be used can be specified independently from the SQL CLI.

Figure 2 Language and API standards



The SQL dialect identified in Information Warehouse Architecture I for both embedded and callable SQL is specified in the ISO-ANSI SQL92 entry level standard. By aligning with existing and upcoming international standards, the Information Warehouse framework callable SQL API can provide an open solution that avoids the pitfalls of most proprietary solutions.

For the user of Information Warehouse applications, the use of a standard data access language and API in this framework promises that it will finally be possible to buy a "shrink-wrapped" application off the shelf and simply install it, and that it will run with any data management system supporting the Information Warehouse framework callable SQL API.

To be realistic, however, one should realize that a standard (common) interface will almost always represent a subset of the function offered by any target data management system. It is impractical to attempt to coordinate competing vendors so that they all produce the same function at the same time. Differences will always exist between different products.

Application builders must deal with a real dilemma. The use of a common interface provides portability and data location transparency for the application—a degree of independence from the rest of the system. The common interface, however, prevents the application from using all of the function available at a data management system. The choice between portability and function belongs to the application builder.

An interface would be deficient if it arbitrarily constrained an application from using function available at a target data management system. Some callable SQL implementations provide an escape mechanism to allow nonstandard function to be used by the application. The mechanism is either a special function call or special data access language syntax. The escape mechanism separates or encapsulates the nonstandard request so that normal syntax and other error checking is bypassed. An application builder can choose to use such a mechanism at the expense of application portability.

A final point to be made here is that the Information Warehouse framework callable SQL API is not dependent on any particular communications protocol. This point can be quite significant whenever a network is migrated to a new protocol or upgraded to a new functional level.

Data access protocols. Thus far we have discussed the components with which the application directly interacts—the data access language and the application programming interface. These components rely on additional layers of software to connect to the appropriate data management system. When the data are local, that is, on the same system as the application that issues the request, the connection is ultimately made by using the data access API of the local data management system. In a distributed environment, the data may be on a remote system, in which case, distributed processing is involved in accomplishing the data access. A data access client on the application system must "talk to" a data access server on the system where the data reside. This communication requires a definition for how requests and information are exchanged between the software on the two systems. Such a definition is referred to as a protocol. A protocol can be more formally

defined as a "set of semantic and syntactic rules that determines the behavior of functional units in achieving communication."[6]

Typically, the protocols required to achieve communication on behalf of an application are actually multiple protocols existing at different levels or layers. Each layer defines a set of functions that are provided as services to upper layers, and each layer relies on services provided by lower layers. At each layer, one or more protocols define precisely how software on different systems interact to accomplish the functions for that layer. This layering notion has been formalized in many architectures. The most widely referenced is the reference model of Open Systems Interconnection (OSI),[7] defined by ISO and depicted in Figure 3. We show it here as an example of the relative responsibilities of different protocols in achieving communications for data access. The figure indicates that there is a peer-to-peer communication between software at each layer and a reliance on underlying layers for services to accomplish communication.

In this subsection, we discuss data access protocols that define the *content and meaning of requests and information* exchanged between data access clients and servers. (In the OSI reference model, data access protocols are a part of the application layer since they are considered an application process.)

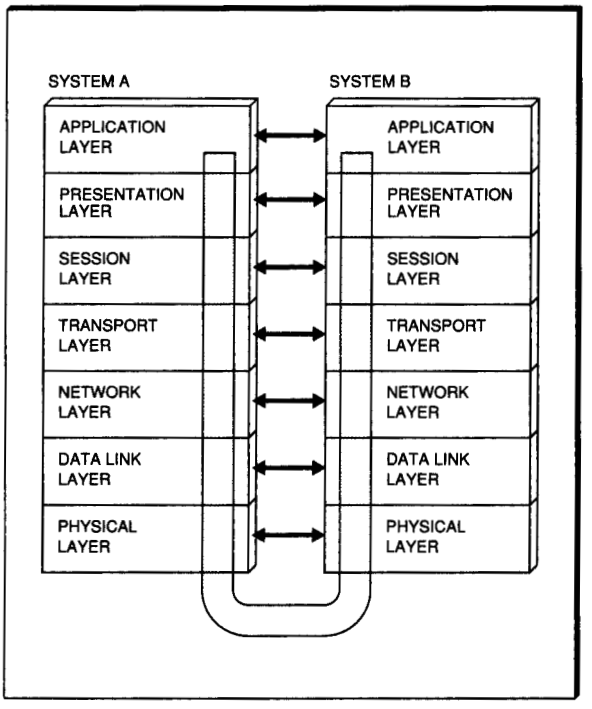
Data access protocols rely on underlying protocols to provide services for the *communication* of these requests and information. In this paper, we refer to these supporting protocols as communications protocols. We will discuss communications protocols briefly, noting implications that the protocols can have on data access. (In the OSI reference model, communications protocols span some or all of the layers of the model below the application layer, depending on the specific protocol and the range of the services it defines.)

We have said that data access protocols define the content and meaning of requests and information exchanged between data access clients and servers. We list some examples here of exchanges that might be defined by a data access protocol. These examples are for a "conversational" protocol which is a request and reply dialog where the data access client sends request messages and the server sends the related reply messages. The examples are:

- Initiation of a conversation
- Identification of the client and server and their capabilities
- Identification of the database to be accessed and its characteristics
- Preparation of an SQL statement for execution, either for application preprocessing or dynamically at application run time
- Processing a request for data
- Termination of the conversation

Data access clients and servers relying on protocols to define their communications is not so different from people relying on a language and associated rules of grammar to communicate verbally. And, just as there are many different spoken languages, there are also many different data access protocols defined. The products that implement these different data access protocols are distinct data access solutions, that is, they accomplish data access in the unique way defined

Figure 3 Reference model for Open Systems Interconnection



by the protocol. A data access client that implements one data access protocol cannot communicate directly with a data access server that implements another data access protocol. Enterprises often have multiple distinct data access solutions in house in order to address all of their data access requirements. This situation presents significant challenges for application use and systems administration and has become an inhibitor to the deployment of client/server technology. This will be discussed further in the subsection on programming and administration aspects.

Today, it is generally accepted that any data access solution must provide access to the data management systems of multiple vendors.

Evolution of data access protocols. In the software industry today most of the significant data management vendors and data access vendors have their own solutions for client/server data access. These solutions are based on data access protocols defined independently by the vendors. Today's data access protocols have evolved in concert with and as a reaction to the evolution of data access requirements and data access technology.

One of the most significant factors in the evolution of data access protocols has been the demand for openness and the support of the heterogeneous enterprise. Today, it is generally accepted that any data access solution must provide access to the data management systems of multiple vendors. However, data access protocols that satisfy this requirement differ greatly in function, topology, and the approach they use to incorporate multivendor data access. The diversity of the protocols results from the diversity of the requirements, independent designs, the starting objectives of a data access protocol, and its "history" or evolution. Here is an example of different starting objectives and evolution: Some data access protocols were developed to support decision support applications and were initially targeted at read access for use with a very wide set of data management systems. Other data access protocols were developed by database vendors for transaction (and decision support) processing within the product sets of the vendors. Both can evolve to satisfy today's data access requirements, but they will be very different from each other.

We noted that today's data access protocols differ in the approach used to incorporate access to the data management systems of multiple vendors. A basic difference in approaches is whether the data access software of different vendors interoperates to create a solution (referred to as "multivendor interoperability" in this paper) or whether a single vendor creates the total solution. For many enterprises, multivendor interoperability is specifically called for in their requirements for multivendor data access. For many other enterprises, it is not a requirement, and either approach is acceptable. Both types of protocols exist today, with the single vendor solution being more predominant.

Design approaches for multivendor data access. When considering approaches for data access protocols that allow access to the data management systems of multiple vendors, two significant approaches stand out: open, common protocols and database gateways. We describe these approaches in this section and point out which ones incorporate multivendor interoperability. Another approach that accomplishes heterogeneous data access outside of the data access protocol will be described later. Microsoft's Open Database Connectivity (ODBC) falls into this last category.

It is the nature of data access protocols to have a given protocol common to the set of data access clients and servers that implement the protocol. The protocol must be general and complete enough to support the requirements of the clients and servers that participate in the protocol. As we have just discussed, the evolution of data access protocols resulted in many diverse protocols. Most of these protocols are "closed" or private, that is, the protocols are used only by the product set of a given vendor. Some protocols are published for the purpose of multivendor participation as clients or servers in a data access solution. As a result, a data access client implemented by

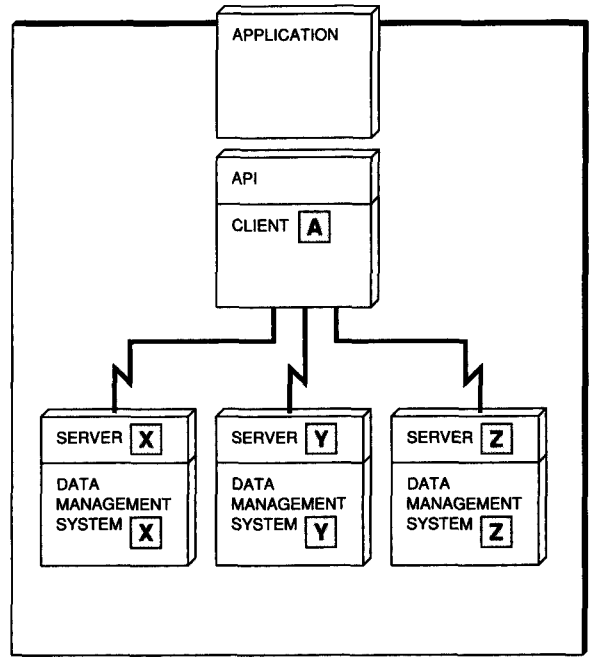
one vendor can interoperate with a data access server implemented by another vendor. We refer to such a protocol as an open, common protocol—open because any vendor can participate, and common because the protocol is common to the set of data access clients and servers that implement the protocol. (Implicit in this use of the term "open" is that the protocol is technically suitable for the use of other vendors.) To be effective as a heterogeneous solution, an open, common protocol must be designed for use across diverse machine architectures and operating systems. The protocol must be very general and complete in order to support data access to heterogeneous data management systems. No assumptions can be made nor any knowledge presumed between the data access clients and servers except what is defined via the protocol. In private protocols, assumptions can be made and knowledge can be built into the data access client and server to support the communication. For example, a private protocol might assume that both client and server software know the format of a particular data structure, and therefore the format does not need to be defined in the communication. Or assumptions can be made that floating point data are represented in a single format rather than in the diverse formats that could exist in a more heterogeneous environment. Such assumptions can result in more efficient but less general communication.

"Opening" the protocol allows, but does not require, the data management system of any vendor to participate directly in the dialog of the protocol. This factor can be very important for performance, data integrity, and security.

Figure 4 illustrates an open, common protocol by showing a data access client communicating with the servers of multiple vendors via a single data access protocol. ISO RDA,[8] X/Open RDA,[9] and DRDA*[10] (Distributed Relational Database Architecture*) are all examples of open, common protocols.[11]

The second approach for multivendor data access is a gateway. A *gateway* is software that has a dual role of both server and client in data access.[11] It acts as a server within a given environment and then acts as a client to pass the application request to the target data management system located in a different environment. In this way, a gateway can be thought of as extending or

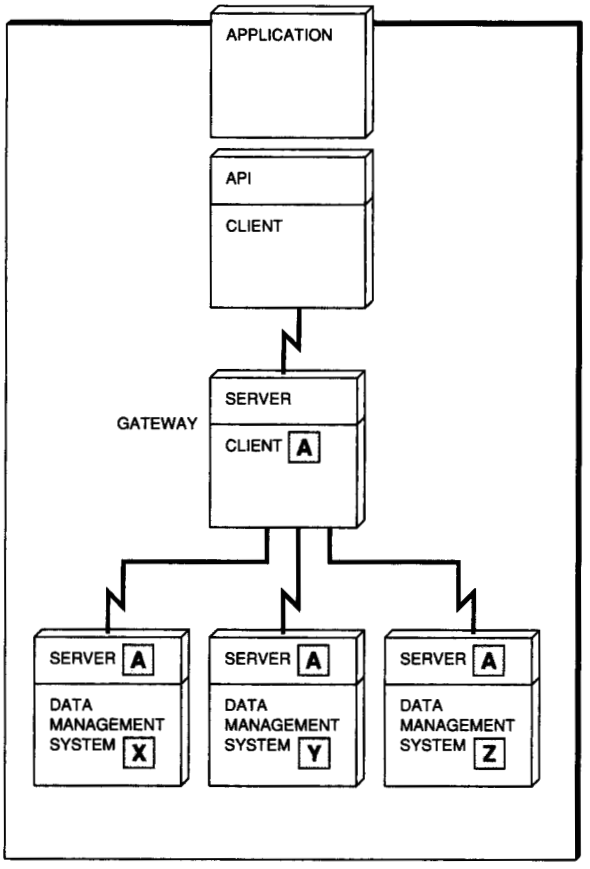
Figure 4 Clients and servers using open, common protocol



complementing the environment with access to additional data sources. For example, a gateway can extend a LAN (local area network) environment by providing access to data sources that are on other LANs or on host systems. Or a gateway can extend the private protocol data access solution of a vendor by providing access to the data management systems of other vendors. When a gateway is used, an application connects to (or is connected to) the gateway as though the gateway were a server. Typically, the application is unaware of the role of the gateway in handling the data access request.

The placement of gateways within a network varies. They can be on the same system as the client, on the same system as the target server, or on a separate system. Where a gateway is placed affects the topology or configuration of the network. In some cases, the architecture of the gateway determines the placement. In others, the placement can be determined by the enterprise on the basis of systems administration considerations. Figure 5 shows a gateway that is on a system separate from both the client and server. The gateway in Figure 6 is on the same system as the target data management system.

Figure 5 Gateway on system separate from client and server



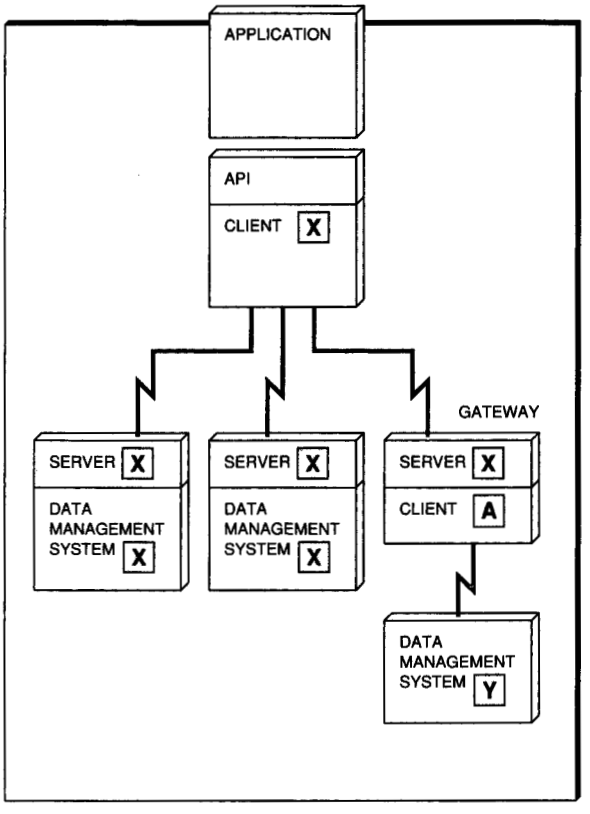
Gateways vary in terms of the functions they perform. Some possible gateway functions are:[12]

- Translating SQL syntax
- Detecting semantic differences
- Converting data types
- Accessing generic system catalogs
- Maintaining transaction boundaries
- Converting status codes and messages
- Mapping user identification and security checks
- Balancing load and limiting the server
- Providing manageable control points for large networks
- Mapping LAN communications protocols to WAN (wide area network) communications protocols

As can be seen from this list, gateways can have an "all encompassing" role in data access, and their responsibilities can span some or all of the logical software components that we discuss in this paper. In this section, however, we are concerned primarily with their role in simply communicating a data access request to the target data management system.

An important measure of the effectiveness of a gateway is the accuracy and correctness with which its functions are provided. The most basic concern is handling a transaction so that data integrity is preserved. Security is another important requirement. Also, when transforming requests for one type of server into requests for another type of server, it is a challenge to correctly represent both types of servers. Unavoidable mismatches in function sometimes occur. It is important to understand how the mismatches are handled and whether there is any loss of function relative to either server.

Figure 6 Private protocol connection to gateway at target data management system



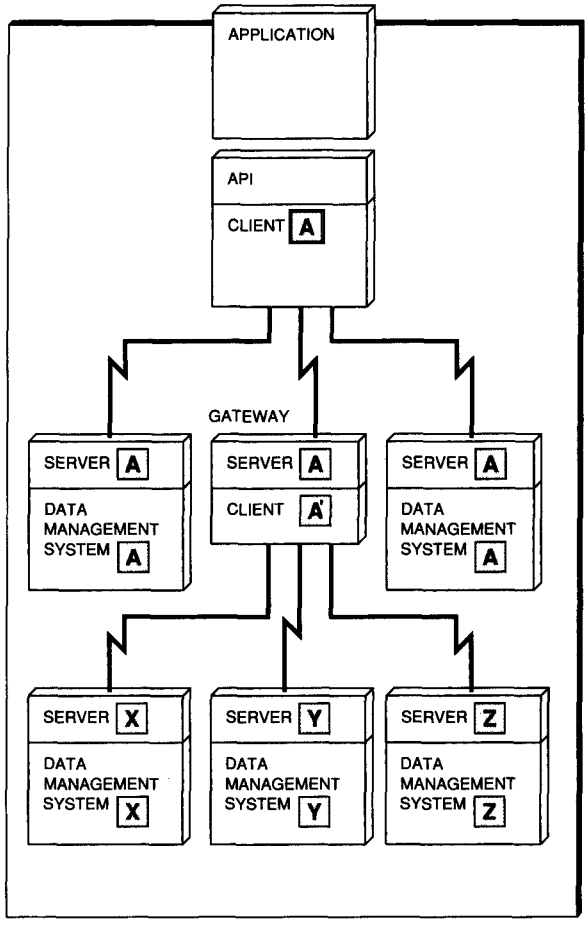
Gateway interfaces are sometimes open, that is, they are published for the purpose of multivendor interoperability. Figure 6 is an example of where the client portion that directly interacts with the target data management system has been implemented by a vendor different from the vendor implementing the server portion of the gateway. The client could be implemented by the data management vendor, although it is more typical for this implementation to be done by a third-party vendor. Gateways that have published interfaces can also be used by an enterprise to develop customized access to a unique or unusual data source.

Any given vendor's data access solution or product set will probably incorporate a combination of common protocols and gateways. The example given earlier of a database vendor using a private (common) protocol for data access within its own product set and incorporating gateways for access to other vendors' data management systems is a case of combined use. SYBASE Open Server** provides an example of this combination that would look like Figure 6. Data access requests are communicated from a SYBASE client ("client X") to a SYBASE server ("server X") using the SYBASE private protocol. The gateway is an open gateway since the Open Server interface is published for the use of other vendors and for customers. In the figure, the client portion of the gateway is implemented by "vendor A" for access to the data management system of "vendor Y." SYBASE also uses this interface to provide its own gateways for access to data management systems like IBM's DATABASE 2* (DB2*) and Oracle**.

INGRES**/Gateway *to* DB2 is another example of a private protocol connecting to a gateway, except in this case the gateway can be connected to an open, common protocol. (Refer to Figure 7.) An INGRES client communicates a data access request to an INGRES server over INGRES Net, a private protocol. The client portion of the gateway is an application that passes the request to IBM's DB2. The DRDA client of DB2 can then communicate the request to any DRDA-capable server, for example, IBM's DB2 on MVS, VM, or OS/400* (Operating System/400*).

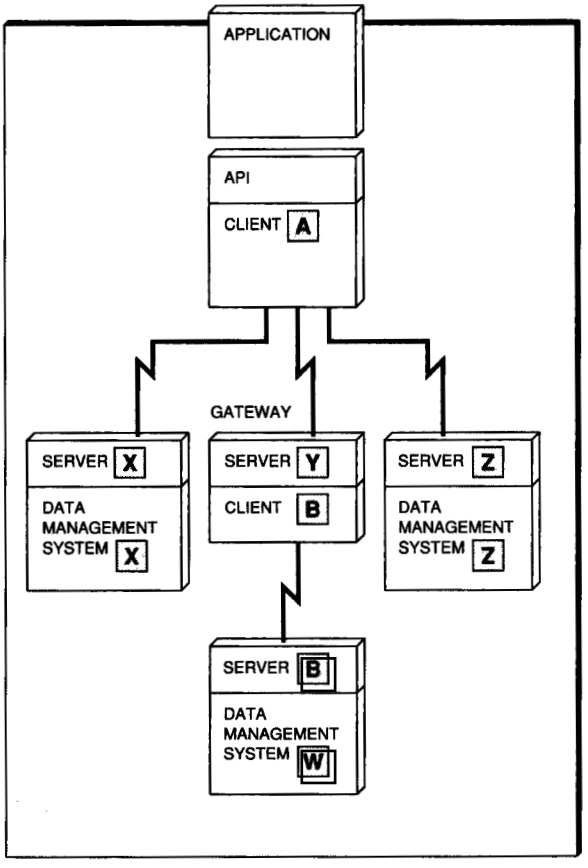
It is also possible, although more unusual, for a common protocol to connect to a gateway. EDA/SQL Server Engine *for* DB2 for DRDA from Information Builders, Inc., is an example. (Refer to Figure 8.) Client A could be any DRDA-capable client connecting to the DB2 DRDA server. (IBM's Distributed Database Connection Services [DDCS/2 and DDCS/6000], Informix** Gateway *with* DRDA, Micro Decisionware Database Gateways *for* DB2, SQL/DS, and OS/400, and XDB Link are examples of DRDA-capable clients.) The EDA/SQL Server Engine *for* DB2 for DRDA provides the client portion of the gateway using exits provided in DB2. The gateway can provide access to IMS (Information Management System) or VSAM (Virtual Storage Access Method) data available on the same system or can provide access to any other data sources available through EDA/SQL.

Figure 7 Gateway connecting to open, common protocol



We have discussed two design approaches that allow access to the data management systems of multiple vendors. In summary, gateways and

Figure 8 Open, common protocol connecting to gateway



open, common protocols are methods for gaining access to the data management systems of multiple vendors. Only open gateways and open, common protocols obtain this access using multivendor interoperability, with the latter being the more common approach since it tends to be a more "level playing field." Open gateways allow a vendor to participate as a server. Open, common protocols allow any vendor to participate as a client, as a server, or as both. The data access solution of an enterprise can then be comprised of data access clients and servers from one or many vendors in a combination decided by the enterprise.

Programming and administration aspects. Because of the diversity of application requirements and the diversity of the capabilities of data access solutions, an enterprise frequently has more than one data access solution in use. This use has implications for programming and administration.

Thus far, we have discussed data access protocols as though they were independent components in a data access solution. In reality, most products that implement data access protocols also provide a programming interface for use by an application. So, when an enterprise has multiple distinct data access solutions, each presents its own API, and each is likely to have some differences when compared to the others. When an application must use multiple data access solutions in order to meet its requirements for data access, the work for the application builder increases as we discussed previously. One factor is skills; the application builder must learn the nuances of multiple APIs. Another factor is that, regardless of whether the APIs are different or the same in style, the application must use each API separately as shown in Figure 9, because the data access solutions are distinct and separate. This has an effect on the design or structure of the application program. It also has a significant impact on data transparency since the application must know, or must be "told" in some manner, which API to use to access the data.

An important implication results from the fact that the API (and often, the data access language) is integrated with the data access protocol. When an application directly uses the API of such a data access solution, and if the application becomes dependent on the API, implicitly, the application (and the enterprise) also becomes dependent on the associated data access implementation. There are also implications for administration. For example, an enterprise may have chosen an open, common protocol as the basis for data access. Theoretically, only one data access client would be required for each client system. Let us say the enterprise has chosen the data access client provided by vendor X. If an application in use at the enterprise is dependent on the API of vendor Y which is integrated with vendor Y's data access client, then the data access client of vendor Y would also be required even though both clients implement the same open, common protocol.

For some applications, a solution such as we describe later in the subsection on connection management software can provide independence between the API and the data access protocol and can also lessen the complexity of dealing with

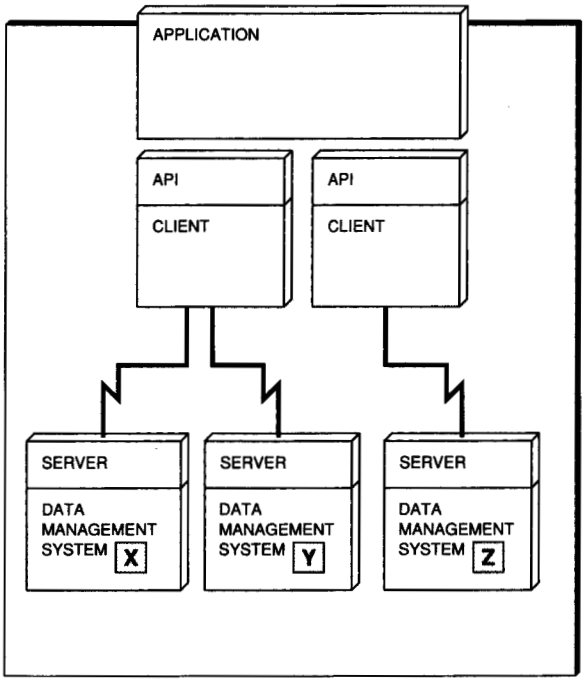
multiple APIs. However, it does not improve the complexities of administering data access solutions in the enterprise since it does not decrease the number of data access solutions required. Having multiple distinct data access solutions in use at an enterprise presents a challenge for systems administrators. The problems are related to installation, maintenance, and problem determination.

Any data access solution is comprised of data access client and server software components and supporting definitional information that exist at the data access client and server systems. The definitions are information related to the network, systems, security, and data management products and databases that can be accessed. Some data access solutions also require their own schema information similar to that in a relational database catalog for any data that are to be accessed. Such definitions are unique to each distinct data access solution. The systems administrators must set up and maintain these definitions along with the data access software components. When an installation has multiple data access solutions in use, the work of the systems administrators generally increases. There are more components (i.e., software components and definitional information) to administer because the solutions are very likely to overlap. There is more complexity because different data access solutions will have their own unique procedures, configuration, and support requirements.

Another type of complexity can occur for systems administrators. Setting up any one data access solution can be complex in a distributed environment because of the layers of software used to accomplish communications. When multiple data access solutions coexist (or try to) in an environment, systems administrators must deal at times with incompatibilities and errors that arise at installation and run time.

From an administration point of view, minimizing the number of components in use for data access and minimizing the number of distinct data access solutions in use reduces systems administration. These factors need to be considered as an enterprise evaluates data access solutions. Some single-vendor "all-in-one" solutions can reduce the number of components, depending on the topology of the design approach. An open, common protocol solution can theoretically reduce the number of data

Figure 9 Application use of multiple APIs



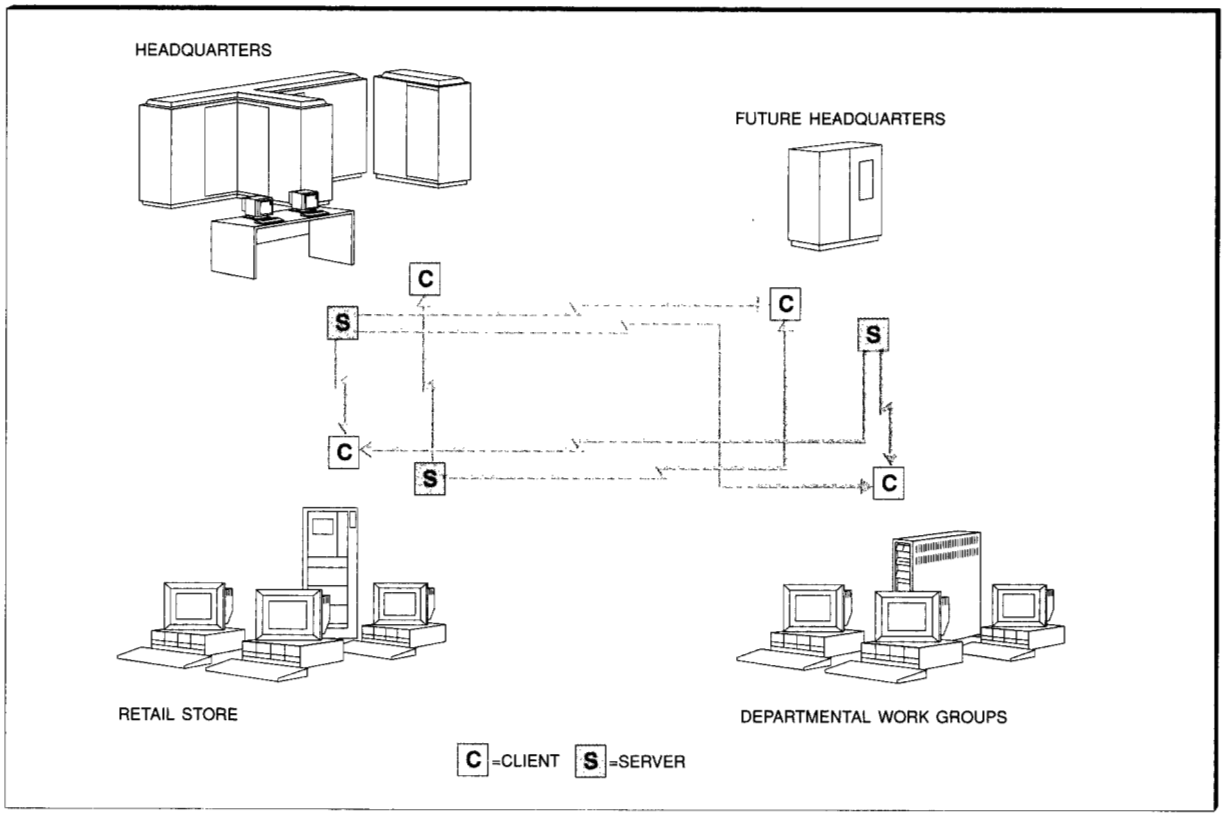
access solutions to one. The number of components can also be reduced to a minimum because typically only one data access client is required per application system and only one data access server is required for each data management system. Definitional information would be required at each client and server. The complexities of problem determination in a multivendor environment can be reduced when the common protocol includes trace and accounting information and system alerts, as is the case for DRDA.

Here is an example of how an open, common protocol could enable data access and data migration in a very heterogeneous environment while minimizing the number of components. Consider the large retailer shown in Figure 10. The retailer has the following systems:

- A mainframe computer at headquarters with enterprise inventory and sales data
- Workstations at each retail store for store inventory and sales data
- Multiple LANs for departmental work groups

Each system is from a different hardware vendor as is each database management system. The re-

Figure 10 Enterprise use of open, common protocol solution



tailer requires the mainframe computer applications to be able to "pull" data from the retail stores for consolidation at headquarters and requires the retail stores and departmental work groups to have access to the consolidated data of the mainframe. Furthermore, the retailer is considering downsizing the mainframe computer to a smaller system from another vendor. Downsizing will require, over time, that the mainframe applications and data be moved to the new system. Eventually, the retail stores and departmental work groups will access the consolidated data on the new system. During a transition period, data will be copied periodically from the mainframe to the new system. With an open, common protocol, it would be possible to enable all of the data "paths" between heterogeneous systems and data management systems having no more than one client and one server on each system.

Open, common protocol solutions have had some success to date, but single-vendor solutions are by far more predominant. Presently, product support is probably the most limited for ISO RDA, perhaps because of the requirement for OSI communications. Some support exists for X/Open RDA, and this support may grow, depending on whether communication protocols other than OSI are supported in the future. Support is the greatest for DRDA, with a mixture of software vendors providing client implementations. Server implementations are provided predominantly by IBM, although some announcements and statements of intent have been made by other software vendors. For an open, common protocol to provide the flexible solution shown in Figure 10, a heterogeneous mix of clients and servers implementing the protocol must be available.

The continuing evolution of data access protocols. We have discussed a few aspects of data access protocols. But enterprise requirements for data access solutions are much broader, including requirements for availability, data integrity, per-

formance, function, and specific communications protocols in addition to openness, flexibility, manageability, and cost that we have discussed. Today, enterprises frequently cannot find a single data access solution that meets all of their requirements. However, enterprise requirements are the most significant determinant, and data access solutions (and their related protocols) will continue to evolve to meet these requirements.

Technology changes can play a role in the evolution of data access protocols. In the subsection on enhancement facility, we discuss a component that, as one of its responsibilities, takes on the role of coordinating or implementing data access operations that combine data (multisite join), optimize distributed operations (query decomposition and global optimization), and coordinate distributed data management operations (multisite update) across multiple heterogeneous data management systems. These data access operations can also be implemented through data access protocols. Multisite join and multisite update have been implemented in data access protocols to varying degrees today. It is possible for an open, common protocol to provide robust, global optimization through the direct participation of data management systems. The protocol would define how data management systems share information and perform tasks such as moving data in order to accomplish the most efficient joins. Enhancement layers introduce a pragmatic "let me do it for you" approach that provides the function independent of the data access protocols and does not require multivendor participation. Whether open, common data access protocols evolve to provide global optimization of queries could be affected by the acceptance and effectiveness of enhancement layer function.

Economics could also be a determinant in the evolution of data access protocols. Data access solutions that are based on open, common protocols can have a lower purchase cost since fewer components are needed. Also, vendors can price their components lower since fewer development resources are required overall; for example, a data access provider would only need to develop a data access client for each client system it supports, assuming that data management vendors provided data access servers. However, today both software vendors and enterprises have made significant investments in data access solutions that are based on private protocols. These investments will affect the acceptance of open, common protocols at least for the near term.

To be effective, an open, common protocol must allow vendors to participate in the definition of the protocol. Earlier we discussed the difficulty of reaching consensus in standards groups. This problem applies here, and, as a result, the definition of an open, common protocol can lag behind advances in technology.

In the end, how much importance enterprises assign to each of their requirements for data access will determine the rate of evolution. Most enterprises struggle with the balance of their requirements. It is a very complex formula.

An enterprise has much to consider to make effective use of what is available and to plan for change. Chief among these considerations is the fact that independence between components in a data access solution can provide flexibility for change. Information Warehouse Architecture I identifies DRDA as a protocol for data access. An open, common protocol such as DRDA provides independence between data access clients and servers.

Communications protocols and their implications. We have stated that data access protocols rely on communications protocols to define services that accomplish the exchange of data access messages between nodes in a computer network. Here we discuss the relationship between data access protocols and communications protocols.

As is true for data access protocols, there are multiple communications protocols in the industry today. Most notable are ISO Open Systems Interconnection (OSI), Systems Network Architecture (SNA), Transmission Control Protocol/Internet Protocol (TCP/IP), and Internet Packet Exchange (IPX**). The fact that multiple communications protocols exist results from diversity in network environments, processor resources, and requirements. An enterprise might use one or more communications protocols within its computer network, depending on the overall configuration requirements. Often, communication solutions are already in place when a data access solution is selected by an enterprise. Thus, use of the predominant, existing communication solutions becomes a requirement for data access solutions.

Having considered the relationship between applications and data access protocols, and the effect of applications having to use multiple interfaces, we can "see the writing on the wall." When a data access solution has to use multiple communication solutions with differing interfaces and functions, it encounters the same types of complexity. Data access solutions can become tied to specific communication solutions because of their unique interfaces and services. For example, ISO RDA and X/Open RDA currently use only OSI communications, and DRDA uses only SNA. Data access solutions can also be limited or enhanced in function depending on the services available through a given communications protocol. For example, SNA session outage notification to the client and server programs makes it possible for DRDA to initiate transaction backout in the case of communication line failure. All of these factors define a relationship between data access solutions and communication solutions that is similar in many ways to the relationship between applications and data access solutions. Obviously, data access solutions could benefit if this complexity were reduced. Not surprisingly, in addition to the software development concerns that we have just discussed, there is an accompanying set of administrative concerns for communication solutions, analogous to what we described earlier for data access solutions.

There are significant differences between communications protocols, both in the amount of services that are provided and in the individual services. A simple way to get an idea of the differences is to discuss the major protocols in the context of the OSI reference model (refer to Figure 3). The OSI protocol and SNA define services over the full range of the stack. However, the two protocols are different in approach and organization. TCP/IP, which was designed for the interconnection of networks with dissimilar communications protocols rather than for application use, defines protocols starting at the transport layer. Examples of services in the presentation layer and session layer are user authentication; session outage notification; and sync point management, which is used to coordinate updates to multiple data access servers.

As enterprises balance their requirements and search for ways to manage this complexity, the same considerations for independence apply. Flexibility can be gained by achieving some independence between data access protocols (or other applications that require communications) and the communication solutions. One approach that simplifies the software development aspects is to allow applications that were implemented for use with one communication interface to use other communication networks with no change to the application. This use can be accomplished by providing underlying software that maps requests for communication services made by the application (in this case, a data access solution) to appropriate services of the alternate communication solution. This approach must provide function compensation when the service requested by the application is not provided by the alternate communication solution. IBM's Multiprotocol Transport Networking (MPTN) architecture[13] defines such an approach. As an example, the MPTN technology enables implementations of DRDA, which was originally designed for SNA LU 6.2 communications, to communicate via TCP/IP.

Connection management software. We started with a premise that enterprise data are often located in multiple places and that applications might require access to any part of the data. Given that applications need to connect to more than one data source, some problems must be solved if these connections are to be made efficiently. The problems associated with connection management include:

- Identifying potential data sources
- Connecting to selected data management systems
- Understanding the capabilities and requirements of a data management system
- Interacting with the data management system
- Providing the necessary control to coordinate multiple connections

It is very inefficient to include all of this capability in each application. It becomes much easier for the application builder if these problems are addressed in a generic way and are provided as a service to the application.

In a popular solution, a vendor provides a single application programming interface and multiple back-end adapters, one for each type of data management system. The solution involves the mapping, by the adapters, from the single API to the requirements of each unique data management system. This solution allows the application to be

insulated from the characteristics of each data management system; they all look the same through the single API. An important objective is to provide a high degree of independence between the API and the adapters (and thus, the data management systems). When this independence is obtained, the overall system gains flexibility. More applications can access a greater number of different data management systems.

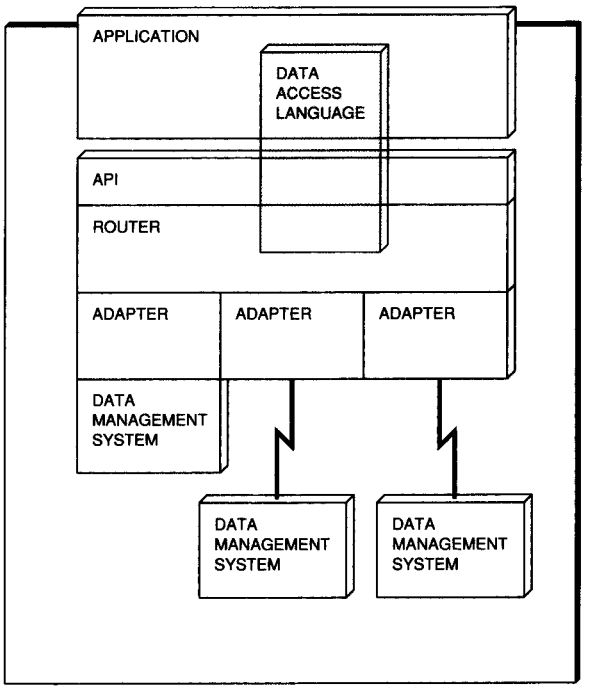
Solutions of this type available today typically define an API and a data access protocol that are designed to work together. Each of the solutions has a different API. The problem here is that applications using any of these solutions are automatically locked into a proprietary API and can reach only the data targets supported by the particular vendor.

A more open solution is to provide a facility that includes a standard data access API, a standard data access adapter interface, a standard data access protocol, and a switching mechanism (a *router*) that controls the connections between applications and data access adapters (see Figure 11). The difference here is that the components are designed as independent, pluggable units, and the emphasis is on the use of open industry standards in the definition of the interfaces and protocols. If the components are independent and the design is open, any vendor can participate by providing a component in the system. This solution could allow any application that uses the standard interface to connect to any data management system for which a suitable data access adapter is provided. Solutions of this type play a key role often associated with data access middleware, as defined earlier in the section on components of a data access solution.

Five important factors contribute to the success of this solution:

1. The design is implemented on all of the required operating system platforms.
2. The supported API contains sufficient function to satisfy application requirements.
3. The design incorporates applicable industry standards.
4. The design is open for any vendor to participate.
5. Application builders and data management vendors are motivated to provide the support.

Figure 11 Connection to multiple data management systems



Many data management vendors in the industry are currently interested in such a design because of its inherent efficiencies, and solutions are appearing, generally based on some form of callable SQL. An application builder needs to write to and test only one interface to gain access to many different data management systems. However, a data management vendor needs to provide only one adapter that will enable attachment by numerous industry applications. Even independent software vendors (those supplying neither the major applications nor the data management systems) are building adapters on behalf of several different data management system products.

Two components of the design have been introduced: the router and the data access adapter. Let us look more closely at the functions of each.

An example may be useful to illustrate the characteristics of the two components. Suppose we have a general-purpose report writing and business graphics application. The application is meant to be used in a generalized way, i.e., the application is to be developed with little knowl-

edge of the data management system with which it will operate. The application only assumes that it can operate using a standard interface (e.g., the SQL CLI). The general design of the application is

The router is positioned between the API and the various data access adapters.

to allow interaction with a user in a dialog that prompts the user for information to tailor a report.

First, the application presents a list of the available data sources (usually expressed in business terms such as "inventory data" or "daily sales summaries") to the user so that one of the sources can be selected. Upon selection, the application connects to the appropriate data management system, then presents a list of information subjects (names representing data) available at that system. The user then specifies the reports to be generated and identifies the data to be used and the functions to be applied to the data. The application determines whether the specified data are available, the data manipulation functions are supported, and the user is authorized to access these database resources. The application also defines the data types expected for the report so that data conversion can be performed if needed. The application then controls the fetching of the data.

Router. Among the first activities in our scenario are determining the possible data sources and making the connection to the selected one. These activities are accomplished using the router.

The router is positioned between the API and the various data access adapters. The router has access to information about each of the potential data management systems and about which adapter is used to gain access to a particular data management system.

When a data access adapter is installed in the system, information about the adapter and the associated data management system are "registered" in the system. Registration information includes identification of the adapter and a list of pertinent characteristics. This information is saved in a configuration file or directory and is used by the router. The information is made available to an application through function calls to the router. It enables an application to interrogate the system about the accessible data.

When the user chooses particular data to work with, the router finds and loads the proper data access adapter and establishes the connection between the application and adapter.

Because of its position between the API and the data access adapters, the router accepts function calls from the application, acts on certain ones, and passes the rest on to the appropriate adapter. Requests and actions associated with the operating environment, connection management, and transaction state management are handled by the router. Other requests and actions such as basic syntax checking, language mapping, data retrieval, buffering, and error code generation are handled by the data access adapter.

In systems that support multiple applications running at the same time, the router also provides traffic control between multiple applications and multiple data access adapters so that each function call is routed to the proper adapter and data flowing back through the adapter is returned to the proper application. Status information must be maintained for each instance of a running application, not unlike the task management required in a multitasking operating system.

The routing described here is not limited to a single node in the network. The routing can be cascaded through as many nodes as necessary. In other words, there can be multiple layers of routers, and when a connection is made, the routing information is passed from router to router until the final destination is reached.

The model we have used thus far has assumed that there is only one active connection at a time between the application and the data management system. Obviously, a router could be provided that supports multiple simultaneous connections. If this were done, the routers would have to take on additional responsibilities with regard to coordination between the connections. We will save

these considerations for a later discussion on the enhancement facility. The delineation between router and enhancement facility is an arbitrary one for the purposes of discussion in this paper. In a given implementation, the boundary between these two components could disappear.

Data access adapter. One of the most important components of the design is the data access adapter. It is this component that hides all of the real differences between the data management system and the standard API. These differences can range from the relatively simple, e.g., transforming a character string from fixed length to varying length, to the relatively complex, e.g., transforming a complex SQL request into the equivalent flat file record requests. It is the responsibility of the data access adapter to make the actual data management system appear to conform to the semantics of the standard data access API.

The data access adapter receives data-related calls from the application via the router and usually translates them into the native language required by the target data source. This operation is most efficient if the target system understands SQL. It is important to note here that differences between SQL dialects, or even the differences between two distinct data access languages, can be effectively masked in the data access adapter in many cases. The degree to which such masking is possible has a direct effect on the consistency and usefulness of the API and the amount of function available to the application for a given data management system. Any mismatch between the API and the native language mapping constrains the application in what can be accomplished at the data management system. In practical implementations today, complete masking of DBMS characteristics may not be possible. Some underlying DBMS capabilities such as multiple connection support or isolation levels may still be exposed. Applications may still need to be aware of and exploit these aspects of the DBMS. Our ideal, however, is to have these considerations minimized at the API. Minimization can often be accomplished by selecting appropriate run-time options or other such DBMS controls that can be manipulated outside the application program logic.

For remote data, the data adapter also invokes any data access protocols necessary to reach the target data management system. These protocols can be private or they can be open, common protocols. The protocols can be independent from the data access API. This independence, provided by the connection management software between the API and the data access implementation, provides additional flexibility to the enterprise in configuring the data access components.

ODBC is a commonly known example of the router and adapter function of connection management software. DRDA is an established distributed data access protocol. These independent architectures have been combined in adapter implementations. ODBC adapters are available that invoke a DRDA data access client to provide access to any available DRDA server.

Note that one data adapter can serve one or multiple data management systems. There are no bounds in the architecture for what an adapter might do to match a data management system with a data request from the API.

The adapter is responsible for conversion between data types if necessary. Data types can vary between hardware and operating system platforms. For example, there are at least three popular variations on floating point number representation. Data management systems may also implement specialized data types such as graphic (multibyte) character fields, or very long binary fields such as the multimedia types for voice, image, or video. We recognize that it is most efficient, and perhaps even necessary, to perform data conversion at the data source, where information about the data is known. If this support is not available, however, many types of data conversions can be performed at the data access adapter.

The adapter may also provide performance enhancements such as blocking of data buffers to reduce communication line overhead.

Each data management system running today almost certainly has its own set of error codes and messages. The adapter translates these specific codes into a standard set of status codes that are defined for the API. Note that whenever error codes are translated to a standard set, there is the potential for loss of information. The native error codes often provide more detailed or specialized information than a general set of codes. To offset this possibility, the data access adapter can sup-

port a diagnostic function where native (unmapped) error codes can be returned to the application.

A final consideration for data access adapters is the positioning of the function within the enterprise network. Depending on the client machine and operating system, the installation of a router and several data access adapters could easily overwhelm the processing and storage capability of the client system. Enterprises that find they have the need for several data access adapters may want to configure a network topology that puts the router and adapter functions on an intermediate server.

The most advantageous place for the enhancement facility is within the router or inserted between the API and the router.

Enhancement facility. So far, we have presented a high-level design that helps universal connectivity between applications and data management systems. We have stated the benefits of standardization, both at the API and the protocol level. Still, we realize that the complexity associated with heterogeneity will remain in the industry. Multiple data management systems and data access protocols are already in use. The functional capability in these systems varies considerably. Natural market forces will generate additional functional extensions. These extensions will not be developed by all vendors in the same way or at the same time. It is not rational to expect all of the database vendors to converge on a single standard in the near future. The cost would be high and the benefit to a particular vendor uncertain.

What seems more rational is to find a solution that would provide application connectivity to multiple data management systems while adapting to the differences in functionality. Several software developers have seen this opportunity and are providing products that map a standard API to many widely varying data management systems.

In basic form, this scheme does the same thing as the design we have already described: a standard API, connectivity to multiple data sources, agreement on a standard core of functionality, and a fair degree of transparency with regard to the data sources. There is a set of problems, however, that our design will not handle without additional function. These problems stem from the need to combine data from multiple sources (multisite joins), to coordinate distributed data management operations (multisite update), and to optimize distributed operations (query decomposition and global optimization). A coordination agent strategically placed in the design can address these problems and thereby provide significant advantages to the application builder and the using enterprise. We call this coordination agent an *enhancement facility* (Figure 12). It plays a key role often associated with data access middleware solutions, as defined earlier in this paper.

Some reasons for adding yet another layer to the design are:

- It is extremely inefficient to put this enhancement function in every application. Duplicating complex function in each application is unnecessary and goes against providing a single consistent application interface.
- There is no provision thus far for one data access adapter to communicate with another in a cooperative way.
- Data management systems from separate vendors do not cooperate in any significant way.

The most advantageous place for the enhancement facility is within the router or inserted between the API and the router. This placement allows the enhancement facility to use the services of the router and the data access adapters.

Function provided in the enhancement facility can solve many of the difficult problems that are nagging the industry today. The two major categories for these problems are:

- Coordination between multiple data management systems—Some data management vendors are supplying distributed systems in which the separate nodes can communicate and cooperate, but only if they are implementations

from the same vendor. Multivendor interoperability is starting to appear via RDA and DRDA, and these protocols can be a basis for coordination; however, multiple distinct data access solutions will continue to exist. We foresee a need, at least temporarily, for a control point that can adapt and coordinate the various data access solutions.

Functions such as distributed update, complex query optimization, and coordinated recovery all require cooperation between the distributed nodes. These operations require complex communication, transaction processing controls, and protection from a vast array of possible failure conditions.

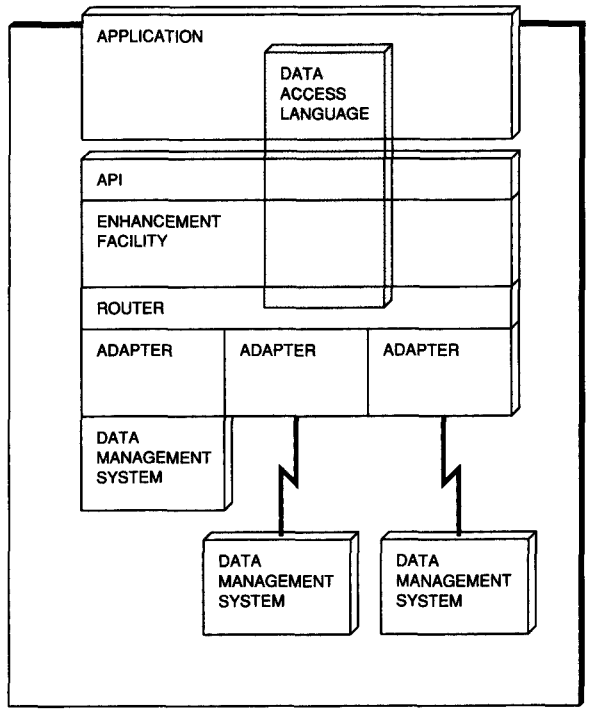
- Masking of functional deficiencies in the data management systems—An enhancement facility can emulate function that might not be available in the target data management system. There is a huge opportunity to equalize the disparate systems. The trade-offs are data location transparency versus performance and network complexity.

The enhancement facility can compensate for functional deficiencies by supplying surrogate function on behalf of the target data management system. Applications are often caught in a trade-off between complete application portability, which requires adherence to a common SQL subset, and being able to exploit the features of an advanced database manager. This trade-off might be resolved in favor of advanced features if the application could team up with an enhancement facility that can hide the differences between data management systems.

Here are some examples:

- Scrolling cursors can be simulated by caching the records in an intermediate buffer.
- Heterogeneous joins can be accomplished by decomposing the query, issuing separate queries (different syntax if necessary) to the target data management systems, and then joining the resulting sets in the enhancement facility.
- Some distributed systems require that the user log on at every node touched by the data requests. This inconvenience can be eliminated if the enhancement facility can authenticate the user one time, then propagate the authentication as necessary to any other node.

Figure 12 Enhancement facility



- Standard schema information (common catalog views) can be presented to the application while working with each data management system in its own language and protocol. Alternatively, a global data catalog can be built at the enhancement facility node.
- Remote data can be cached at the enhancement facility for improved performance.
- Multiple site update with two-phase commit integrity can be achieved by controlling each target site individually. This is possible where each target site is capable of committing an update.
- Integrity constraints can be enforced across heterogeneous target data management systems.
- System management functions can be supplied on behalf of a set of clients.

Note that the enhancement facility functions are logical building blocks that can be placed on the client platform, an intermediate server, or with the target DBMS. The functions can be combined with routing or data access adapter functions. No distinct interfaces are defined for these functions. The net effect of these functions is to increase consistency of services to the application and to

decrease dependency on a particular data access protocol or target data management system.

The complexity of the enhancement facility functions will tend to force the implementation away from the relatively small client machines toward larger and faster machines acting as servers or bridges in the network.

Conclusions

We have described logical building blocks that comprise solutions to address requirements for data access. In summary, the requirements of the application builder, both those in the enterprise and application software vendors, can be satisfied to a significant degree. Satisfying enterprise requirements for efficient systems administration are more difficult, however. Improvements are identified in some cases. In other cases, trade-offs are described where application builder requirements are met at the expense of increased systems administration.

We suggest two basic approaches for maximizing and protecting the investments of application builders and the enterprise—aligning with standards for the data access language and API, and structuring the data access solution such that components are interchangeable and the dependency of one component on another is minimized.

Use of a standard data access language and application programming interface by an application provides efficiency in developing the application and limits the dependency that the application has on the provider of an application programming interface and on individual data access management software. Limiting language and API use to what is defined by standards is difficult, however. As we stated earlier, extensions beyond the standard often have significant value. Application software vendors may be pressured to exploit nonstandard extensions in order to differentiate their products through specialized support. Because use of a consistent data access language can limit function, an application builder might choose to use a consistent language (or dialect) for a broad set of data management systems and then use server-dependent language as an exception for a limited set of strategic data management systems.

A data access solution can be structured in several ways to make components independent. Some make it easier for the application builder to use a standard data access language and API. Each is summarized below.

When a consistent data access language is presented to an application and then translated as necessary to the dialects of specific data management systems, it results in a larger base of consistency in the language and increases the independence between the application and the data management system. It is ideal when the data access language used by the application conforms to a recognized standard. If the data access language is not standard, the application becomes dependent on the provider of the translation rather than on multiple specific data management systems. When language translation is performed, unavoidable mismatches sometimes occur. It is important to understand how the mismatches are handled and whether there is function loss relative to the target data management system.

Information Warehouse Architecture I identifies ISO SQL92 as the data access language for the Information Warehouse framework.

When a router is included, independence is introduced between the application and the individual data access solutions. It is ideal when the API of the router conforms to a recognized standard. If it is not standard, the application becomes dependent on the provider of the router rather than on the individual data access solutions. For an enterprise that has multiple data access solutions, application development effort is reduced through the use of a single API, yet the application is enabled to use many individual data access solutions, thereby reaching many data sources.

Information Warehouse Architecture I identifies X/Open SQL CLI as the callable SQL API for the Information Warehouse framework. X/Open SQL CLI is the API that would be associated with a router component within the Information Warehouse framework.

When open, common protocols are used, this introduces independence between the data access clients and servers. This method allows an enterprise flexibility in choosing or interchanging the components of the data access solution. Solutions

based on open, common protocols can also provide benefits in reduced cost and administration.

Information Warehouse Architecture I identifies DRDA as an open, common protocol for data access in the Information Warehouse framework.

When an enhancement facility is included, this lessens the dependence that an application has on individual data management systems since there is more consistency of function and the application is less aware of differences between the data management systems.

Because Information Warehouse Architecture I identifies open and standard language, interfaces, and protocols, it offers an opportunity for the enterprise to obtain flexibility in choice of data access components, easier future expansion without perturbing current applications and components, reduced cost through simplified networks, and reduced administrative workload.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Borland International, Q+E Software, Information Builders, Inc., SYBASE, Inc., Oracle Corporation, Relational Technology, Informix Software, Inc., or Novell, Inc.

Cited references and notes

1. *Information Warehouse Architecture I*, SC26-3244, IBM Corporation (April 1993); available through IBM branch offices.
2. We use the term *data access client* to refer to software that executes in the operating system where the application runs and makes requests for data on behalf of the application. The term *data access server* is used for software that executes in the system where the data reside and responds to these requests.
3. *Database Language SQL*, ISO 9075-1992, International Organization for Standardization, Geneva (1992); *Database Language SQL*, ANSI X3.135-1992, American National Standards Institute, New York (1992); *Common Applications Environment (CAE) for SQL*, X/Open XPG4-SQL, X/Open Company Limited, 1010 El Camino Real, Suite 380, Menlo Park, CA 94025 (August 1992).
4. The SQL Access Group is a consortium of software companies interested in promoting a set of SQL standards that will allow heterogeneous connectivity between their data management systems.
5. This same SQL CLI will be the basis for other implementations of an SQL call level interface. Microsoft Corp. has stated the intent to support the SQL CLI in a future version of ODBC.
6. *IBM Dictionary of Computing*, Tenth Edition, McGraw-Hill, Inc., New York (August 1993), p. 542.
7. *Information Processing Systems—Open Systems Interconnection—Basic Reference Model*, ISO 7498-1984, International Organization for Standardization, Geneva (1984).
8. *Information Technology—Open Systems Interconnection—Remote Database Access, Part 2: SQL Specialization*, ISO/IEC 9579-2:1993, International Organization for Standardization, Geneva (1993).
9. *X/Open CAE Specification, SQL Remote Database Access*, Doc. No. C307, ISBN 1-872630-98-7, X/Open Company Limited, 1010 El Camino Real, Suite 380, Menlo Park, CA 94025 (July 1993).
10. *Distributed Relational Database Architecture Reference*, SC26-4651-01, IBM Corporation (March 1993); available through IBM branch offices.
11. M. Zimowski, "DRDA and RDA: A Comparison," *Database Programming and Design* **7**, No. 6, 54–61 (June 1994).
12. R. D. Hackathorn, *Enterprise Database Connectivity*, John Wiley & Sons, Inc., New York (1993).
13. *Multiprotocol Transport Networking: Technical Overview*, GC31-7073, IBM Corporation (1993); available through IBM branch offices.

Accepted for publication February 7, 1994.

J. Phil Singleton *IBM Software Solutions, Santa Teresa Laboratory, 555 Bailey Avenue, San Jose, California 95141 (electronic mail: phil@stlvm6.vnet.ibm.com).* Mr. Singleton is a senior programmer in the Information Warehouse Distributed Data Strategy and Architecture group. He is chairman of IBM's SQL Language Council and coordinator for the Information Warehouse SQL call-level interface. Prior to his current work in strategy and architecture, he helped develop the database query strategy for IBM's application enabling layer products and coordinated the development of the SAA Query Interface and the SAA Database Interface. Previous experience includes positions in technical planning, field introduction programs, management, system verification testing, programming assurance, and field engineering. He has worked for IBM since 1965.

M. Michele Schwartz *IBM Software Solutions, Santa Teresa Laboratory, 555 Bailey Avenue, San Jose, California 95141 (electronic mail: mschwartz@vnet.ibm.com).* Ms. Schwartz has been involved in various aspects of IBM's relational database product family since 1981. Currently, she is working on the data access strategy for IBM's data management products as an advisory programmer in Information Warehouse Distributed Data Strategy and Architecture. She has also participated in the definition of Information Warehouse Architecture I, distributed database planning for DB2, and the definition of IBM's query strategy. She was the lead programmer for Query Management Facility (QMF), a relational database query application. Prior to her experience with IBM's relational product family, Ms. Schwartz lived in Houston, Texas, and was a software developer on a flight design system for the Johnson Spacecraft Center and on real-time utility and traffic control systems.

Reprint Order No. G321-5544.