Interoperability, Distributed Applications and Distributed Databases: The Virtual Table Interface

Michael Stonebraker Paul Brown Martin Herbach
Informix Software, Inc.

Abstract

Users of distributed databases and of distributed application frameworks require interoperation of heterogeneous data and components, respectively. In this paper, we examine how an extensible, object-relational database system can integrate both modes of interoperability. We describe the Virtual Table Interface, a facility of the INFORMIX-Dynamic Server Universal Data Option, that provides simplified access to heterogeneous components and discuss the benefits of integrating data with application components.

1 Introduction

Software interoperability has many faces. Two of the most important are application and database interoperability. These are the respective domains of distributed application and distributed database technologies. First-generation products in these areas tended to support only homogeneous systems. Customer demand for open systems, however, has delivered a clear message to vendors about interoperability: "*distributed* must be *open*."

1.1 Incremental Migration of Legacy Systems

"Reuse of legacy applications" is the most frequently quoted requirement for application and data interoperability. The typical corporate information infrastructure consists of dozens, if not hundreds, of incompatible subsystems. These applications, built and purchased over decades (often via the acquisition of entire companies), are the lifeblood of all large companies.

Two goals of application architectures today are:

- making these legacy applications work together, and
- allowing IS departments to incrementally rewrite those which must be rewritten because of changing business needs.

Sophisticated IS architects realize that legacy reuse really means managing the gradual and incremental replacement of system components, at a measured pace. Some attention has been given to a strategy of incremental migration of legacy systems [BS95]. This is seen as the best methodology for controlling risk, limiting the scope

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

of any failure, providing long-lived benefits to the organization, and providing constantly measurable progress. This strategy involves migration in small incremental steps until the long-term objective is achieved. Planners can thus control risk at every point, by choosing the increment size. Interoperable distributed application and database management systems are important tools for practitioners of this approach.

1.2 Interoperability and Distributed Application Frameworks

Reuse of legacy applications is often a significant motivator for the adoption of distributed application frameworks, especially object request brokers, such as the Common Object Request Broker Architecture (CORBA). IT departments wish to rapidly assemble new applications from a mixture of encapsulated legacy components and newly engineered ones. ORB vendors meet this need with wide-ranging connectivity and encapsulation solutions that provide bridges to existing transactional applications. In addition, the ORB client of a component's services does not know where or how a component is implemented. This location transparency is an important enabler of reuse and migration because replacing components does not affect the operation of a system, as long as published interfaces are maintained. Thus, object orientation is seen to be an enabler of interoperation.

1.3 Interoperability and Distributed Database Systems

Most modern relational database systems also support data distribution with location transparency. The basic model of the distributed database, interconnected database instances with common (and transparent) transactional "plumbing," lends itself well to a gateway model of interoperation. A real-world information system includes many types of database system, from different vendors, and of different vintage and capability. Gateways successfully connect these database systems so location-transparent requests may address heterogeneous data sources.

1.4 Combining Distributed Applications and Distributed Data

Combining distributed application and distributed database capabilities into a single framework will maximize flexibility in interoperation, reuse, location transparency and component. Application developers need to invoke business logic without regard to the specifics of that logic's current implementation. They also need to be able to develop the logic without regard to the underlying data's current storage mechanism.

The database management system is ultimately responsible for transactional and data integrity. The services that distributed application frameworks define for transactional control (e.g. CORBA Object Transaction Service, Microsoft Transaction Service and Java Transaction Service) are a first-generation approach at integrating distributed applications with distributed data. The tighter integration between transactional systems and ORBs called Object Transaction Managers (an example is M3 from BEA Systems) evince recognition that guaranteeing transactional integrity in distributed-object applications is a formidable task.

The question is clearly not which type of distribution mechanism you need (ORB or distributed database), but how are they best combined. In the next section we discuss enhancements to relational database management systems (RDBMS) that provide additional means of merging distributed application and data facilities for enhanced interoperability.

2 Extensible Database Systems

2.1 SQL3

ANSI (X3H2) and ISO (ISO/IEC JTC1/SC21/WG3) SQL standardization committees are adding features to the Structured Query Language (SQL) specification to support object-oriented data management. A database management system that supports this enhanced SQL, referred to as SQL3, is called an object-relational database management system (ORDBMS). An ORDBMS can store and manipulate data of arbitrary richness, in contrast with a SQL2 database system that primarily manages numbers and character strings. SQL3 allows for user extension of both the type and operation complements of the database system. This enhancement to RDBMS technology provides several benefits, including:

- the ability to manage all enterprise data no matter how it's structured,
- co-location of data-intensive business logic with dependent data, for greater performance,
- sharing of enterprise-wide processes among all applications (another method of reuse),
- reduction of "impedance mismatch" between object/component application model and relational data model, and
- a vehicle for integration of database and distributed application framework.

In the rest of this section we discuss the framework of type extensibility that SQL3 provides, and some important database extensibility features beyond the scope of the SQL3 specification.

2.2 Extended SQL Types

The most fundamental enhancement of an ORDBMS is support for user-defined types. SQL3 provides new syntax for describing structured data types of various kinds. A UDT may be provided as part of a type library by the ORDBMS vendor, third-party suppliers, or by the customer. The UDT is the basis for a richer data model, as well as a data model that more closely maps to the real world (or that of OO analysis and design methodologies).

2.3 Extended SQL Operations

An ORDBMS also supports user-defined functions to operate on the new user-defined types. Rather than being limited to a proprietary interpreted stored-procedure language, state-of-the-art ORDBMS implementations allow user-defined functions (or methods) to be implemented in a variety of languages. A complete UDF facility will allow data-intensive functions to execute in the same address space as the query processor, so that the enterprise database methods may achieve the same performance levels as built-in aggregate functions.

2.4 Beyond SQL3

An RDBMS has many areas that could benefit from extensibility that are not addressed in SQL3. The programming environments that may be used to extend the ORDBMS is an important area not fully addressed by the SQL3 standard. The specification provides for the use of interpretive languages like SQL and Java; however, some types of system extension are only practical if a compiled language like C may be used for extensions. Allowing compiled extensions to run in the same address space as the vendor-supplied portions of the RDBMS provides some unique architectural challenges that are beyond the scope of this paper.

Vendors of object-relational systems can be expected to make their products extensible in other ways as well. For example, the INFORMIX-Data Server Universal Data Option (a commercially available ORDBMS) includes the ability to add index methods for optimized access to complex types. SQL3 specifies how a user-defined type may be constructed to support geospatial data, but not how a user-defined multi-dimensional indexing algorithm may be incorporated within the ORDBMS engine. Without such an indexing method, queries against two-dimensional geospatial data will not perform well. The ability to access existing applications and data from new applications that require richer query capabilities is an important interoperability concern, so the performance of type and operation extensions is always critical.

2.5 Extended Storage Management

The SQL3 standard specifies enhancements to the syntax and semantics of the *query engine* half of an RDBMS, but is silent on changes that would affect the *storage manager*. It has been common practice since the earliest days of relational database technology to build an RDBMS in two distinct parts. The query engine is the RDBMS front end, and is engineered to translate user queries into the optimal set of calls to the storage manager. The storage manager is the RDBMS back end, and is engineered to translate calls from the query manager into the optimal set of I/O operations. Extending the set of index methods, as discussed above, is an extensibility dimension that affects the storage manager.

In addition to indexing mechanisms, type extensibility challenges many other assumptions made by a typical RDBMS storage manager. Data of a user-defined type might be of much larger (or even worse, variable) size than classical data. Inter-object references can create very different data access patterns than will occur with classically normalized data. Even transaction mechanisms and cache algorithms may be impacted by different client usage regimes encouraged by large, complex objects. The ability to tailor portions of the ORDBMS storage manager is a very challenging requirement for vendors.

3 Virtual Table Interface

An example of a storage management extensibility mechanism with special significance for data and application interoperability may be found in the INFORMIX-Data Server Universal Data Option. The *Virtual Table Interface* (VTI) allows the user to extend the "back end" of the ORDBMS, to define tables with storage managed by user code. The query processor and other parts of the ORDBMS "front end" are unaware of the virtual table's special status.

The Virtual Table Interface is used to create new *access methods*. When you use VTI, the data stored in a user defined access method need not be located in the normal storage management sub-system. In fact, the data may be constructed on the fly, as by a call to an external application component, making VTI a useful interoperability paradigm.

3.1 How the Server Uses VTI Interfaces

To implement a new virtual table access method, one writes a set of user-defined functions that can substitute for the storage management routines implemented by the ORDBMS. To understand what we mean by this let's see how an ORDBMS works normally. Within the ORDBMS, SQL queries are decomposed into a schedule of operations called a query plan. Query plans typically include operations that scan a set of records and hand each record, one at a time, to another operation, perhaps after discarding some records or reducing the number of columns in each record.

For example, when a query like:

```
SELECT * FROM Movies;
```

is passed into the DBMS, a query plan is created that implements the following logic. In the pseudo-code example below, functions printed in **bold** type make up the interface that the query processing upper half of the DBMS uses to call the storage manager.

TABLE_DESCRIPTION ∗ Table;
SCAN_DESCRIPTION ∗ Scan;
ROW ∗ Record;
Table := **Open_Table**(Movies);
Scan := **Begin_Scan**(Table);

```
while (( Record := Get_Next(Scan)) != END_OF_SCAN) {
      Process(Record);
}
End_Scan(Scan);
Close_Table(Table);
```

Developing a new access method requires that you write your own versions of each of these highlighted functions. Third party vendors may use this interface to write their own storage management extensions for the ORDBMS: gateways and adapters to exotic systems, interfaces to large object data types (like Timeseries and Video data), and facilities to tie the ORDBMS environment into other infrastructures.

The DBMS's built-in implementations of these functions are very sophisticated. They interact with the locking system to guarantee transaction isolation. They understand how to make the best use of the memory cache and chunk I/O for optimal efficiency. User defined access methods – written by application developers rather than database engineers – are usually much simpler than the built in storage manager functions because their functionality is more specialized. Quite often read-only access to an external object is sufficient. When you write a new read-only VTI access method there is rarely any need to implement a locking or logging system. An access method may be as simple as the implementation of a single function (**Get_Next**), although enhancements to improve performance could complicate things. We will discuss query optimization and virtual tables in a later section.

Furthermore, to support INSERT, UPDATE and DELETE queries over a data source adds other complexity to an access manager, which we will also discuss in a later section.

3.2 Creating a New Storage Manager

To use the virtual table interface, you need to:

1. Create a set of user defined functions implementing some subset of the interface (for example, the five highlighted functions in the above example).
2. Combine these functions into an *access method* using the CREATE ACCESS METHOD statement.
3. Create a table that uses the new access method.

When the query processor encounters a table on a query, it looks up the system catalogs to see whether or not that table is defined as an internal table, in which case it uses the internal routines. If the table is created with a user defined access method the ORDBMS creates a query plan that calls the user defined functions associated with that access method in place of the built-in functions when running the query.

The set of functions developed in step (1.) consists of a single mandatory function, **Get_Next**, and zero or more additional functions (Table 1). **Get_Next** is called by the query processor to fetch a row from the table. Some of the other functions may be implemented to optimize table scanning by isolating start-up and teardown overhead (the **Open_Table**, **Begin_Scan**, **End_Scan**, **Close_Scan** and **Rescan** functions). Others are used for table modifications (**Insert**, **Update**, **Delete**), maintenance (**Drop_Table**, **Stats**, **Check**), query optimization (**Scan_Cost**) or to push query predicates down to query-capable external components (**Connect**, **Disconnect**, **Prepare**, **Free**, **Execute** and **Transact**). Another set of functions that manages virtual table indexes is also called at appropriate times (such as create/drop index, insert, delete, update). In the interest of brevity, these functions are not explicitly treated here. Any unimplemented member functions of the access method are treated as no-ops.

3.3 Optimizing Virtual Table Queries

Although at its simplest a VTI access method may be a single function implementation that returns a single row, simple optimizations may yield considerable performance improvements at the cost of some design complexity.

Function	*Category*	*Description*
Get_Next	Mandatory	Primary scan function. Returns reference to next record.
Open_Table	Setup	Called at beginning of query to perform any initialization.
Begin_Scan	Setup	Called at beginning of each scan if query requires it (as when virtual table is part of nested loop join).
End_Scan	Teardown	Called to perform end-of-scan cleanup.
Close_Table	Teardown	Called to perform end-of-query cleanup.
Rescan	Teardown/setup	If this function is defined, query processor will call it instead of an End_Scan/Begin_Scan sequence.
Insert	Table modification	Called by SQL insert.
Update	Table modification	Called by SQL update.
Delete	Table modification	Called by SQL delete.
Scan_Cost	Optimization	Provides optimizer with information about query expense.
Drop_Table	Table modification	Called to drop virtual table.
Stats	Statistics maintenance	Called to build statistics about virtual table for optimizer.
Check	Table verification	Called to perform any of several validity checks.
Connect	Subquery propagation	Establish association with an external, SQL-aware data source.
Disconnect	Subquery propagation	Terminate association with external data source.
Prepare	Subquery propagation	Notify external data source of relevant subquery or other SQL.
Free	Subquery propagation	Called to deallocate resource associated with the external query.
Execute	Subquery propagation	Request external data source to execute prepared SQL.
Transact	Subquery propagation	Called at transaction demarcation points to alert the external data source of transaction begin, commit or abort request.

Table 1: Virtual Table Interface User-Defined Routine Set

Techniques such as blocking rows into the server a set at a time and caching the connection to the external data source between calls to **Get_Next** are typical.

Caching data within the access method is possible as well, but this greatly complicates the design if transactional integrity is to be maintained. In practice, VTI access methods typically depend on the ORDBMS query engine to avoid unnecessary calls to **Get_Next** (natural intra-query caching) and do not cache data across statements.

For a normal table, the DBA gathers statistics about the size and distribution of data in a table's columns as part of database administration. The access method can gather similar information about the virtual table. The query engine calls two access method functions, **Stats** and **Scan_Cost** while determining a query plan to get this information. The **Stats** function provides table row count, page count, selectivity information and other data used during query optimization. Sophisticated data sources (other database systems) typically provide interfaces for determining this information (e.g. the ODBC SQLStatistics interface) which may be called from the access method. For other external data sources, the access method author determines how complete a set of statistics is worth providing the query engine.

While these statistics are typically fairly static in practice, experience has shown us that other dynamic factors may have a significant effect on query planning. For instance, if a hierarchical storage manager physically manages the external table, the data may become orders of magnitude more expensive when it moves from disk to tape. The **Scan_Cost** function allows the access method to convey a weight to the query engine to reflect the effect of such volatile factors. For instance, the optimizer might have decided that an index was normally not worth using if it would return more than 50% of a table's rows, but a sufficiently high scan-cost factor might alter that decision point to 90%.

We note that the functionality of **Get_Next** is relatively self-contained. Because the results of this operation can be used in other processes like sorts, summaries and joins, there are requirements that scan positions be markable and that scan operations be able to skip back or forward. This means that a function that initiates a re-scan of the data, rather than simply calling a combination of the **End_Scan** and another **Begin_Scan** can improve the efficiency of the access method. For example, consider a nest-loop join. On completion of the 'outer' loop of the join, it is necessary to reset the inner scan to its begin point. If the remote data source provides cursor functionality it is more efficient simply to reset the cursor to the start, rather than close the cursor and re-issue the query. This resetting logic can be handled in the **Rescan** function, rather than complicate **Get_Next**.

3.4 A Non-deterministic Approach

Informix has also extended cost based query optimization to queries involving external data sources when it is impractical to gather normal statistics. We use a neural-network-based technique for estimating selectivities of predicates involving these virtual tables [LZ98]. A system table holds a neural network representation of a selectivity function for predicates involving specified external tables. Given the input parameters in the predicate, the neural network returns an estimate of the selectivity. The access method invokes the appropriate neural network to obtain selectivity estimates, which are then used in determining the network cost, predicate push down strategy, and the order in which different predicates are evaluated. The neural networks are trained off-line using data gathered from actually executing the predicates with different input parameters. Experience to date indicates that this heuristic approach may out-perform deterministic optimization algorithms even when traditional statistics are available (and accurate). It is of course particularly appropriate for external tables.

3.5 Accessing Other SQL-aware Data Sources

The subquery-propagation category of functions listed in Table 1 enable the access method to interoperate intelligently with other SQL-aware data providers. The presence of implementations of these functions signals the query engine that subqueries involving these virtual tables can be handled by the access method. Typically this means that the access method will dispatch subqueries to an external database system.

Because the optimizer has the same information available to it regarding the virtual table as it has about an internal table, it is free to make proper decisions during query planning. If the costs are appropriate, a join will be pushed (via the **Prepare** and **Execute** functions) to the access method and hence to the external database. In this way, intelligent gateways become instances of VTI access methods.

3.6 Supporting Insert, Delete and Update Operations

The access method includes functions to support insert, delete and update operations. In the current implementation, certain transactional limitations are imposed. If the external data source supports ACID transaction facilities, then the access method can use this to provide transactions over the local representation in the absence of local data. (The ORDBMS is simply a sophisticated client of the remote database system.) When the application calls for integrity rules between data stored locally and data stored remotely, the remote system needs to support a two-phase commit protocol. The introduction of means for the remote system to send transactional messages to the ORDBMS, and the management of local data caches in the ORDBMS are subjects of continuing research.

3.7 Interoperability and VTI

Because a virtual table is implemented procedurally, the managed data returned by the access method may (i) originate from within the ORDBMS storage manager (perhaps with the access method adding some transformation before forwarding the data to the query processor), (ii) reside externally to the database system or (iii) be computed by the access method itself. In case (ii), the data may be stored in the file system or in another relational database system invoked by the access method. (In other words, VTI is a convenient and flexible way to write specialized distributed database gateways.) If the access method invokes the services of an external component (a CORBA object, say) then that external component will look to the ORDBMS client just like another table.

The Virtual Table Interface mechanism adds considerable value to the CORBA framework. If CORBA is used to encapsulate a legacy application, the incremental development cost of a few ORDBMS-resident functions will make that application available to any client application or tool that can access a relational database. No matter that the legacy application generates a batch report, was written 30 years ago, in COBOL, by a long-forgotten employee of an acquired company, and with the last known source code on a 9-track tape that was misplaced in 1979.

Because extensible database systems will all support Java as a user-defined function language, and because the links between ORBs and Java are growing ever stronger, much of this legacy-bridging will be done in this universal language. Not only does Java provide a simplified (and standard) development environment, this language actually has some positive performance implications. A single Java Virtual Machine can support the ORB, ORDBMS, business components and legacy wrappers with no extraneous inter-process context switching. For example, the Informix Dynamic Server shares a process with the installed JVM, so user-defined functions such as VTI access methods can execute with very good performance. ORBs will also recognize and optimize calls between co-located client and server.

3.8 Example: Enhancing a Legacy Application

To illustrate how an extensible database, a virtual table facility and a distributed application framework can all contribute towards a reuse platform, we consider The XYZ Corporation, a telecommunications service provider. XYZ has an extensive set of legacy mainframe applications which it must continue to deploy. In order to offer customers new features more rapidly (an absolute business necessity), XYZ has decided to interconnect all of their information systems via a CORBA ORB. XYZ is developing a suite of new customer network administration tools. This tool suite is being developed with a commercial object-oriented development toolset, and incorporates an ORDBMS. The ORDBMS includes a number of types and operations that represent XYZ business entities and procedures. In addition, the ORDBMS is extended with a number of generic type extensions, including extensive geographical and spatial operations.

One of XYZ's legacy applications is FAILREP. Run nightly, FAILREP produces a report listing all network nodes with detected error rates above a standard threshold. Nodes are listed by a proprietary network addressing scheme.

XYZ has decided to create a new ORB service which encapsulates the current FAILREP. Among the operations of this service are a number of scanning functions which are called by a VTI access method that allows the FAILREP report to be "seen" by the ORDBMS as a table. Developers of new applications may now reuse FAILREP in innovative ways. For example: "show me all groups of failing nodes, where a group is ten or more nodes within one mile of each other." Or "where is the center of the 50-mile radius circle with the greatest number of failing nodes?" Or fewest? Or, using additional operations of the ORDBMS time-series type, "what 5 states contain the highest number of 20-mile radius circles containing more than 100 nodes that have failed at least 5 times over the last 90 days."

These queries utilize existing enterprise applications as well as new corporate processes and data, and may be issued from standard database development tools. An extensible database enables this interoperation, and a virtual table capability allows for a convenient "contract" between developer and RDBMS. The CORBA infrastructure provides a similar contract between legacy application and developer.

3.9 The Next Step: Generic CORBA Access Method

It is even possible to eliminate the incremental development cost of the VTI access method. By defining CORBA interfaces that are isomorphic with the various groups of VTI functions (read, read/write, query, etc.), a generic CORBA access method may delegate the various functions to the CORBA object. Merely implementing one or more interfaces in the COBOL application's CORBA wrapper will make that application available to any database client. The ORDBMS will not need to be touched in any way (Figure 2).

The CORBA/VTI interfaces comprise an extensible gateway. This gateway uses the power of CORBA to connect a canonical interface to an arbitrary implementation (across many system architectures) and the power of an extensible database to connect many kinds of clients to a data source using a canonical query language. The result is the ability to painlessly view data, hitherto locked in a proprietary system, with virtually any data-aware development tool.

Although not yet available commercially from any vendor, the ORB/Virtual-Table approach reduces the complexity of database and component interoperability to a design pattern, and will certainly appear in products as extensible databases become mainstream technology.

4 Conclusions

The inexorable drive to distributed object (or component) application development is fueled largely by interoperability requirements. As middleware solutions proliferate, terms like "openness" and "bridge" are used by vendors to describe technologies that are perhaps a bit too proprietary. Software makers are also investing heavily in truly interoperable designs. The history of CORBA itself is illustrative. Whereas early OMG specifications did not include interoperability (and the first generation of ORB products were not open), the Internet Inter-ORB Protocol has made CORBA a real interoperation mechanism, and all ORB vendors have adopted IIOP.

When we consider distributed application and data frameworks as interoperability enablers, we enter the Panglossian world of Tanenbaum's Observation: "The nice thing about standards is that there are so many of them to choose from."[Tan96]

It is likely that OMG CORBA, Microsoft DCOM, JavaBeans and distributed SQL3 will all contribute to a brave new world of distributed applications. These technologies are all immature; furthermore, it is unlikely that this is the definitive set of technologies. Consider that two years ago CORBA and DCOM seemed the only games in town for distributed component frameworks. Java has seemed to have popped out of a software black hole to take a very significant share of corporate IS.

The properly defensive IS architect must be exploring frameworks for distributed applications and distributed extensible database systems. There is no escaping the observation that these systems will become more integrated over time. One mechanism of integration that is worth exploring is the Virtual Table Interface, applied to distributed objects. There are many benefits to providing SQL access to distributed objects.

References

[BS95] M. Brodie and M. Stonebraker. Migrating Legacy Systems. Morgan Kaufmann, San Mateo, CA, 1995.

[LZ98] S. Lakshmi and S. Zhou. Selectivity Estimation in Extensible Databases – A Neural Network Approach. In *Proc. of VLDB*, New York, USA, August, 1998.

[Tan96] A. Tannenbaum. Computer Networks. Prentice Hall, Englewood Cliffs, NJ, 1996.