Data Access Interoperability in the IBM Database Family

Michael J. Carey Laura M. Haas James Kleewein Berthold Reinwald
{*carey, laura, reinwald*}*@almaden.ibm.com; kleewein@us.ibm.com*

1 Introduction

Business enterprises, and society in general, are becoming increasingly dependent on computer systems. As a result, we are now awash in a sea of data—data of all shapes and sizes—making heterogeneous data management a tremendously relevant challenge today. Moreover, the problem of data heterogeneity is itself varied, with different applications posing a variety of requirements. Some applications need to access and/or manage data in several, possibly many, different database systems—some with different data models. Other applications need to access and/or manage external data, e.g., data stored in file systems or other specialized data repositories, together with data sets residing in one or more databases. Still other applications need to compose business objects from a combination of legacy data and legacy transactions (e.g., travel reservation systems) provided by multiple legacy database management systems. Of course, the systems that contain all this data differ in many ways—they have different data access languages and APIs, different search capabilities, different integrity guarantees, different data types (and even type systems), and so on.

In this paper, we provide a brief and necessarily incomplete overview of what IBM is doing to address some of the aforementioned challenges. In particular, we describe how IBM's DB2 Universal Database product family is responding to these challenges in order to provide *heterogeneous data access* capabilities to DB2 customers. To address the problem of accessing and managing data across multiple databases and database systems, the DB2 family includes *DataJoiner* technology; this technology provides transparent SQL-based access to legacy data that may be managed by any of a number of vendors' database systems. To address problems related to external data access and management, the DB2 family offers several relevant technologies; these include *Table Functions* for customized user-defined access to external data, *DataLinks* for keeping file system data in synch with database data, and various *Extenders* for managing new types of data such as text, imagery, and spatial data. To more naturally model new data being brought into the system, and to extend the performance and transparency benefits of the DataJoiner technology to cover a broader range of sources, these systems are being extended with new *Object-Relational* capabilities and with support for *Wrappers*.

In the remainder of this paper, we provide more information about each of the DB2 extensions mentioned above. We explain what each of the extensions is about, summarizing the capabilities that DB2 currently provides, or will soon provide, in each area. Readers with an interest in the related area of composing business objects from legacy transactions as well as data are encouraged to look at the *Component Broker* technology [IBM98] that IBM is developing to address that problem.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

2 Heterogeneous Relational Data Access

The DB2 UDB product family employs IBM's *DataJoiner* technology [Kle96, IBM97] to provide access to heterogeneous relational data residing in multiple databases. In this section we discuss the goals and concepts underlying this technology; we also describe briefly how data access requests are processed.

2.1 Relational Interoperability Goals

IBM's goal for relational database interoperability is to make the task of accessing, analyzing, and updating business data easier and faster. To do this, we strive to provide users with *transparent* access to heterogeneous data. This transparency can take many forms, including *location transparency* (allowing users to access data without knowing what data source it comes from), *SQL dialect transparency* (enabling users to access data using one SQL dialect regardless of the native dialects of the sources), *error code transparency* (providing a consistent set of error codes), and *data type transparency* (providing a consistent set of data types).

With transparent access to data, users connect to what appears to be a DB2 database and access all their data as though it were locally stored there. They do not need to connect to data sources directly—only indirectly, via DB2—and they submit all queries and updates using DB2's dialect of SQL. For example, suppose one table ACCT.PAYABLE resides on a Sybase server and another table ACCT.CUSTOMER lives on a Oracle server. Thanks to DB2's DataJoiner technology, a user can compose a DB2 query such as

```
SELECT X.AMOUNT, X.PRODUCT, Y.NAME, YEAR(Y.CUSTOMER_SINCE)
  FROM ACCT.PAYABLE X, ACCT.CUSTOMER Y
  WHERE X.CUST_NUMBER = Y.CUST_NUMBER
```

that joins data residing in both sources without being aware of the location of the data sources. Moreover, the user can apply DB2's YEAR scalar function in the select list without knowing the combination of Oracle scalar functions needed to achieve the same results. Also, the result types are all DB2 data types, so the amount column, for example, will be returned as a decimal value despite the fact that it might be stored differently (e.g., as a MONEY value) in the Sybase database.

In addition to providing a high degree of transparency, the IBM DB2 DataJoiner technology also provides very good performance—performance similar to, and in some cases better than, the performance achievable with a single database. In the remainder of this section, we elaborate on some of the details of the technology that makes this level of transparency and performance possible in DB2.

2.2 Naming and Mapping Concepts

To explain how DB2 achieves both transparency and performance, we need to introduce two basic concepts. The first is the concept of a *nickname*, which is a local name for a remote table. Nicknames allow DB2 to resolve name space conflicts without violating location transparency and provide *catalog transparency* to application tools that rely on the availability of catalog entries. They also provide an opportunity to record statistics for use in optimizing access to non-local data. The second concept is the notion of a *mapping*. A mapping represents a relationship between a local (DB2) concept and another (non-local) system's concepts. Two key places where mappings are important are in handling data types and functions.

A *type mapping* indicates how non-local datatypes relate to local datatypes. These relationships may be complex, so an extensible mechanism is necessary to describe them. One example of such complexity involves the date/time datatypes. There is little actual standardization between vendors for these types. IBM offers three date/time datatypes, date, time and timestamp; Sybase offers two, datetime and small datetime, while Oracle offers just one, namely date. Type mappings can also support relationships between non-local data types and local *user-defined* datatypes, which can help provide even greater transparency overall. For example, a user could create a local DB2 data type of YEN based on DECIMAL(20,0), and then define a type mapping that indicates that data of type MONEY or SMALLMONEY from a particular Microsoft SQL Server database located in Japan should be mapped to YEN instead of DECIMAL.

A *function mapping* indicates the equivalence of a local function and a non-local function (or series of functions). As an example, the DB2 function HOUR (which extracts the hour portion of a datatype that has a time component) could be mapped to the Oracle functions TO_NUMBER(TO_CHAR(¡data value¿,'HH')). Function mapping is complicated by limited orthogonality among vendors; e.g., the fact that the DECIMAL scalar function is supported on an integer in some data source does not mean that it will necessarily be supported there on a small integer. As with type mappings, a function mapping can also represent a relationship between a data source function and a local *user-defined function*.

2.3 SQL Query Processing

The DB2 query engine must consider many issues when processing an SQL statement involving multiple sources. To ensure efficiency, the engine must decide how to divide the work involved in the query up among the various sources. To ensure transparency, it must consider many factors. Some are obvious, e.g., how many relations can appear in a single SQL statement, how many levels of subselects are supported, and whether or not correlation is supported in update statements. Other factors are less obvious, such as differences in collating sequences, treatment of NULL values when sorting, comparison semantics for character values, and handling of 0-length versus NULL-valued strings. Still other issues are esoteric, such as whether or not the presence of a scalar function in a predicate involving an indexed column causes the index to become unusable for that query, or whether joins terminate upon finding a match or iterate until end-of-set is reached. Some factors are related to function, i.e., to how the non-local system behaves from an external perspective, while others are performance-related; both kinds are important for processing queries correctly, transparently, and efficiently.

The mechanisms used to process a data access request involving non-local data depend on the source where the data resides and on the portion(s) of the request that the source can process. DB2's DataJoiner technology employs *push-down analysis* to determine which portions of a request can be evaluated at the non-local data source. Portions of the request that can be pushed down may appear in the select list, the from clause, the predicates, in subselects, in subqueries, and elsewhere. In fact, even when the entire predicate cannot be pushed down, it is possible that portions of it may be. For example, let us suppose that ACCT.PAYABLE is a nickname for a table residing on an Oracle server. Suppose a user asks the following query:

```
SELECT X.AMOUNT
  FROM ACCT.PAYABLE X
  WHERE X.CUST_NUMBER = 55234
```

In this case, the user's query can be pushed down in its entirety. However, consider the following slightly different query:

```
SELECT myfunc(X.AMOUNT)
  FROM ACCT.PAYABLE X
  WHERE X.CUST_NUMBER = 55234
```

The latter query cannot be pushed down as a whole—the local user-defined function `myfunc` cannot be evaluated at Oracle, so that portion of the select list cannot be pushed down—but it can be partially pushed down. The portion of a query that cannot be pushed down is processed locally, by DB2, as DB2 performs *functional compensation* to support all of its functionality against all data (even when the owning database cannot perform the function).

It is important to recognize (as DB2 does) that just because a function can be pushed down to the owning database does not mean that doing so is actually a good idea. Pushdown analysis merely determines whether or not a function can be pushed down; it is left up to the query optimizer to determine whether or not it should be. To decide, the optimizer uses nickname statistics maintained in the local DB2 database. Many times the correct decision is not to push the request down (contrary to the heuristic that says the owner can process a request faster because the data is local there). An obvious example of this is a Cartesian product of two tables that reside in the same non-local database—even if the tables only have a cardinality of 1000 rows, the million-row result table would be much more costly to transmit over the network than a pair of 1000-row base tables.

3 Easier Access to External Data

Although databases are a very important data source, they contain just a fraction of the world's data. Thus, in addition to providing seamless access to a multitude of (mostly relational) databases, the IBM DB2 UDB family offers several ways to provide access to more "exotic" data sources [RP98]. In this section, we cover three complementary approaches that provide varying amounts of functionality, transparency, and performance.

3.1 Table Functions

A table function is a user-defined function (UDF) that returns a collection type such as a table. Table functions were first introduced in Starburst [LLPS91] as a convenient way to import data from outside the database and present it as a virtual table without having to first store the data within the database. Products such as IBM's DB2 Universal Database have extensively enriched this concept. Table functions are now a part of the SQL3 standard, and they have been implemented by several object-relational database system vendors.

As a typical example, the following table function, `SALES`, takes a filename (of a spreadsheet) as an input parameter and returns a table containing six columns selectively retrieved from the spreadsheet. The spreadsheet contains data about sales forecasts for products.

```
CREATE FUNCTION SALES (VARCHAR (20)) RETURNS TABLE
  (PRODUCT VARCHAR (10), REGION CHAR(1), STATE VARCHAR (20),
   YEAR INTEGER, MONTH INTEGER, AMOUNT INTEGER)
  LANGUAGE C EXTERNAL NAME 'DB2SMPL!LIST';
```

Once registered, this table function can be used in SQL wherever a table reference is allowed. For example, the following query joins the table function with a database table.

```
SELECT S.REGION, S.STATE, S.AMOUNT, P.NAME
FROM TABLE (SALES('c:\sales.xls')) AS S, PRODUCTS P
WHERE P.NAME = S.PRODUCT AND P.GROUP = 'toasters'
```

While table functions do not supply the transparency of the DataJoiner technology described in the preceding section, they provide SQL-based access to a much broader range of sources. Table functions rely heavily on functional compensation by the DB2 engine, although some optimizations are possible. The query optimizer can indicate to the table function that only a subset of its columns will be needed, thereby allowing projection to be pushed down if the table function supports this optimization. In addition, parameters can be used to support the pushing down of certain predicates into a table function.

Table functions are ideal for accessing external data sources that have limited search capabilities. Such sources expose the bulk of their functionality through application-specific object models such as those defined for MAPI, Excel, XML, Word, or Lotus Notes; interesting sources therefore include spreadsheet files, document databases, mail databases, and HTML/XML data files. In some cases, the external data source exposes its data through a single standard data access API, in which case all of its table functions will basically implement the same algorithm(s) to access the data. Such table function implementations can be eliminated by introducing a generic table function implementation within the query engine. When this is done, the user can create a new table function by simply providing certain schema information and execution properties. We call such table functions *implementation-free table functions* since no implementation is required from the application developer. An example of an implementation-free table function is a generic OLEDB consumer that retrieves data from a variety of OLEDB providers that expose their data in the form of OLEDB rowsets.

3.2 Extenders

Extenders provide a way for the DB2 database engine to interoperate with other search engines [DM97]. Using extenders, DB2's engine can "learn" about a new type of data which it may access with help from an internal or external search engine. Extender data may be stored in the database itself; if the search engine is external, it essentially acts as an external index. In this section, we discuss IBM's text and spatial data extenders. Other extenders (e.g., for image, video, and audio data) are also available.

DB2's text extender adds the power of full-text retrieval to SQL queries by implementing DB2 user-defined types and functions (UDTs and UDFs) that allow unstructured text documents of up to 2GB to be managed by DB2. The text extender can store and retrieve any kind of text document in its original native form, and it offers a rich set of retrieval capabilities including word, phrase, wildcard, and proximity searching. The text data itself is stored in the database, but the text search capabilities are supported by an external text search engine which deploys its own customized text indices that are geared toward high-performance linguistic search. Flexible mechanisms are provided to keep the external indices up to date through DB2 triggers.

Another example is DB2's spatial data extender. This extender augments DB2's datatypes with spatial types such as *point, line*, and *polygon* and functions such as *area* and *distance*. It also includes the capability to *geocode* addresses (i.e., to convert a text address into a point). The spatial data extender gives DB2 the ability to respond to questions such as ”show me customers that live within 10 miles of one of my stores”, or ”show me the average sales to customers who live within 25 miles of highway 101 and have more than two cars”.

The spatial data extender highlights the synergistic combination of heterogeneous data access with object-relational extensions. By using the spatial data extender together with DataJoiner technology, DB2 can support queries that combine access to traditional business data (average sales) with spatial data analysis (distance between my customer and my store) to quickly answer complex questions without regard for the database in which the traditional data is actually stored. This enables the use of new and powerful capabilities without requiring a costly relocation or replication of massive amounts of data.

3.3 DataLinks

The previous approaches seek to exploit external data as searchable object-relational DB2 data. There is another category of data, e.g., CAD/CAM files, that must be viewed and manipulated using complex, file-based, legacy application programs. This data may be closely related to data in relational databases, such as inventory data, making it important to keep the two sorts of data consistent. As a result, the user may wish to maintain database integrity for such external data and/or to be able to use a high level query language to search for it parametrically. DB2 provides DataLinks to support this type of ”linking” of file system data with database data. DataLinks offer full referential integrity, access control, and coordinated backup and recovery for database-related file system data. A new ANSI ”DataLink” data type is provided for use as a column in DB2 tables. Such columns contain URLs of external files, called ”links”. A file manager performs link/unlink operations with transactional semantics and referential integrity within the file system. A filesystem filter (implemented as a virtual file system) intercepts file system requests and controls access to the ”linked” files by enforcing DB2 privileges for update and access requests.

4 Diverse Data from Intelligent Sources

So far, we have seen how IBM's DataJoiner technology provides transparent and efficient access to data residing in a variety of relational databases. We have also seen how table functions, extenders, and DataLinks can be used to incorporate new kinds of data into DB2 (albeit with less transparency) from a broader set of data sources. Ultimately, our goal is to be able to access data from anywhere, exploiting the search capabilities of intelligent external data sources for efficiency, while preserving the transparencies introduced in Section 2. To do this, we must provide a truly unified view of diverse data to the users, and we must also account for the quirkiness and range of search power that can be found in individual data sources. In this section, we will describe our solutions to these two challenges.

4.1 Modeling Diverse Data as Objects

To transparently access external data, that data must be representable inside our data management system in a way that is both natural and convenient to work with. We are using features of the object-relational data model, and the relevant SQL3 language extensions, for this purpose. In this subsection we summarize the most important of these features.

Perhaps the most central object-relational extension is the provision of support for the declaration of *structured types* and for storing and manipulating tables of objects and column values of these types. A structured type definition is not unlike a class definition in any strongly-typed object language—a *structured type* has some number of attributes, and methods can be associated with these types. Once defined, a structured type can be used as the basis for defining a table for holding objects of that type, as an attribute of some other structured type, or as a column of a (regular or typed) table. Because heterogeneity implies variety, DB2's structured type facility includes support for subtyping; a structured type can either be a root type or a subtype of another structured type. A typed table can be prepared to hold instances of a subtype of its declared type via the definition of a *subtable*. To enable direct modeling of complex objects, DB2 also supports strongly-typed *reference types* so that, for example, the type `emp_t` can include an attribute called `dept` of type `ref(dept_t)`. Of course, DB2's dialect of SQL is being extended accordingly, including support for queries and updates over table hierarchies, queries including path expressions, and so on. Finally, work is also underway to extend DB2 to support collection-valued attributes (such as sets, multisets, arrays, and lists).

In addition to these object-relational extensions, DB2 also supports another extension that is very relevant to the topic of this paper: *object views*. While the aforementioned extensions do much to enrich DB2's base table facilities, many applications prevent users from accessing base tables, instead providing different users with access only to views of the database. Thus, the view facilities of DB2 have been extended to provide support for object views, i.e., for hierarchies of typed virtual tables (views). Just like tables, views can now be defined based on a structured type, with the view body specifying how to materialize the relevant set of instances of this type. Object views can be defined over legacy relational tables as well as over table hierarchies, and view objects can contain references to other view objects. DB2's object views are fully functional, supporting inserts, updates, and deletes as well as read access for querying in many cases. Combining object views with the ability to interact with many diverse data sources is the key to providing a truly unified view of diverse data.

4.2 Requirements of Diverse Data Sources

As noted earlier, data today comes from a broad range of sources that have differing characteristics. Data may be stored in file systems that have only primitive search capabilities (e.g., grep or find), in application systems that do sophisticated processing but answer only a fixed set of "queries" (e.g., simulation models), in specialized search engines for particular data types (text, molecular structure, image, etc), or in classical database systems that, depending on their data model, can perform a variety of searches and other operations. In addition, every customer has a different set of sources that they must deal with. Given such diversity, we need a system with the flexibility to easily extend to additional external data sources, as the solutions sketched in Section 3 do, but which provides the transparency and efficiency that we have today in accessing other relational sources (Section 2). In particular, it is essential for the system to be able to learn about and exploit the search capabilities of new data sources. We have prototyped a *wrapper architecture* that enables this.

The wrapper architecture [RS97] provides a set of interfaces for dealing with external data. A *wrapper* is a collection of code, matching these interfaces, that stands between the data management system and a particular source of data. To be sure that wrappers will exist for all the sources we might want to access, wrappers must be easy to write. While wrapper writers need to understand their data source, they should not have to understand (let alone write code for) all the intricacies of the SQL language or a typical SQL query processor. In our architecture, wrapper writers provide an interface to the natural abilities of their source. If the source has limited capabilities, the wrapper will require very little code; wrappers for high-function sources such as relational DBMS's are more complex. This "thin wrapper" approach is in contrast to many existing systems, which essentially require a full SQL front-end to be written for each new source.

The goals of our wrapper architecture are extensibility, evolvability, and performance. Extensibility refers to the ability to add new wrappers to a system (and hence, new sources of data) at any time. Evolvability means that a wrapper can start simple and then grow over time. For example, the wrapper writer can start by exposing only a low level of functionality, such as the ability to iterate over the data in the source, and later expose further abilities, such as the possibility of applying predicates locally, or of doing aggregations. Performance implies that the wrapper must be able to surface the underlying query power of the source, and that the middleware must exploit that power—wisely.

4.3 Flexibility and Efficiency in the Face of Diversity

The wrapper architecture consists of four key interfaces that handle data description, query planning, method dispatch, and query execution [RS97]. We briefly describe the first two here.

The data description interface is used to set up the nicknames and mappings needed by the middleware. Because we need to be able to accurately describe such a broad variety of data, we need the capabilities of the object-relational model. In particular, we model data in the sources as objects, use methods to encapsulate the special searches and other operations that a data source is capable of, and use nested collections (of various collection types) to naturally model the data from the sources. For example, a molecular data source can be modeled naturally as containing objects that have normal attributes, such as *names*, as well as attributes such as *structure* that may be complex types or even collections. The molecular source's ability to look for similar sequences can be modeled as a *similar_to( )* method on molecule objects.

A unique feature of the wrapper architecture is its query planning interface. This interface allows the wrapper to participate in query optimization, providing information about the cost and feasibility of having the wrapper execute various portions of a query. The optimizer builds query plans bottom-up, starting with accesses to single tables, and then uses the plans created in that phase to build plans for two-table joins, then three-table joins, etc. At each phase in this process, as the optimizer notices that the data to be accessed is stored in an external source, it asks the wrapper (using a generic dispatch mechanism) if it can do the work and how much it will cost. Work the wrapper cannot do (for example, it might not be able to do a join, or apply a particular predicate) can be compensated for by the middleware. In this way, transparency is preserved, but the optimizer does not have to know the details of what a source can do; the wrapper can decide on a query-by-query basis how much work to volunteer for. This allows us to meet our extensibility and evolvability goals while still providing the optimizer with enough information to be effective [HKWY97, HKWY96].

Default code is provided for each of the wrapper interfaces, so wrapper writers only need to refine appropriate portions of the default code to expose the functions he or she wishes to provide. For example, the default code for the query planning call to construct a plan for a join operation simply returns null (no plan). Since most data sources do not do joins, the wrapper writer will leave this alone. However, since all wrappers must at least iterate over the objects they expose, the wrapper writer will need to provide code to translate such a request into commands against the underlying data source.

We believe that the combination of the object-relational extensions with the wrapper architecture will enable us to meet our goals of a unified view of data preserving data transparency, coupled with relatively easy extensibility to new sources, all with excellent performance. Both the object-relational extensions and the wrapper architecture have been used in customer shops, and they are now being combined with the features described earlier to create a truly universal DBMS.

5 Conclusions and Future Work

In this paper, we described how IBM's database products provide interoperability across a broad range of data sources. Our goal is completely transparent access to data, whether local to the database engine or residing in external files or applications. We showed how we achieve transparent access today amongst relational sources and how we can access data today in a much broader range of sources. Finally, we described how, in the future, our object-relational extensions and wrapper architecture will allow us to extend our transparent access mechanisms to the broader range of data types and intelligent data sources.

Many challenges remain. One significant issue is the need for data interoperability standards, e.g., for registering external data sources, naming and catalogs, and exchanging information relevant to optimization. A second challenge is benchmarking and improving the performance of heterogeneous queries. Third, for complete transparency we must eventually provide full DBMS functionality—including authorization, constraint enforcement, accounting, and system management—across diverse data sources. Finally, in the future we must investigate other modes of interoperation, including such issues as mobility and publish/subscribe paradigms.

References

[DM97] S. Dessloch and N. Mattos. Integrating SQL databases with content-specific search engines. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 528–537, Athens, Greece, August 1997.

[HKWY96] L. Haas, D. Kossmann, E. Wimmers, and J. Yang. An optimizer for heterogeneous systems with nonstandard data and search capabilities. *IEEE Data Engineering Bulletin*, 19(4):37–44, 1996.

[HKWY97] L. Haas, D. Kossmann, E. Wimmers, and J. Yang. Optimizing queries across diverse data sources. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 276–285, Athens, Greece, August 1997.

[IBM97] IBM. *DB2 DataJoiner Application Programming and SQL Reference Supplement*. IBM Corporation, 1997.

[IBM98] IBM. IBM component broker technical overview. *http://www.software.ibm.com/ad/cb/litp.html*, 1998.

[Kle96] J. Kleewein. Practical issues with commercial use of federated databases. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, page 580, Bombay, India, September 1996.

[LLPS91] G. Lohman, B. Lindsay, H. Pirahesh, and B. Schiefer. Extensions to Starburst: Objects, types, functions, and rules. *Communications of the ACM*, 34(10):94–109, 1991.

[RP98] B. Reinwald and H. Pirahesh. SQL open heterogeneous data access. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 506–507, Seattle, WA, USA, June 1998.

[RS97] M. Tork Roth and P. Schwarz. Don't scrap it, wrap it! a wrapper architecture for legacy data sources. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 266–275, Athens, Greece, August 1997.